ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Datenübertragung mittels MQTT

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Lukas Schulz
lukas.schulz@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Rüdiger Pryss
Karl Herkommer

2017

Fassung 16. Oktober 2017

Kurzfassung

Das Internet der Dinge bezeichnet die Vernetzung unterschiedlicher Geräte über das Internet. Die Wahl eines geeigneten Nachrichtenprotokolls ist aufgrund der hohen Anforderungen nicht einfach. Um eine möglichst breite Vernetzung zu ermöglichen, muss das Protokoll für so viele Gerätetypen wie möglich verwendbar sein. Hierzu gehören auch Geräte mit schlechter Hardware und Geräte in Gebieten mit schlechter Netzanbindung. Da wir uns im Bereich des Internets bewegen, darf natürlich auch die Sicherheit und Zuverlässigkeit des Protokolls nicht vernachlässigt werden.
Ziel dieser Arbeit ist die Präsentation des Nachrichtenprotokolls MQTT, das im Internet der Dinge weite Verbreitung findet. Zunächst wird die genaue Funktionsweise des Protokolls erläutert. Nachdem die Sicherheit des Protokolls genau betrachtet wurde, werden die Vor- und Nachteile des Protokolls herausgearbeitet. Anhand einer Beispielimplementierung wird zudem dessen Verwendung erläutert. Anschließend werden alternative Protokolle des Internets der Dinge vorgestellt und mit dem MQTT Protokoll verglichen. Am Ende der Arbeit wird ein Fazit gezogen, das das MQTT Protokoll für die Verwendung im Internet der Dinge bewertet.

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken die mich bei der Fertigstellung der Arbeit unterstützt haben. In erster Linie bei meinem Betreuer Rüdiger Pryss, der immer gute Ratschläge für mich hatte.
Aber auch bei meinen Kollegen der Firma "habemus! electronic + transfer GmbH¨ in Thannhausen, die mir durch ihre Unterstützung das Ganze erst ermöglicht haben. Ein besonderer Dank gilt hierbei meinem Betreuer Karl Herkommer und meinem Kollegen Bernd Kalwar, die mir jederzeit bei Problemen helfen konnten.
Außerdem möchte ich mich bei meiner Familie und Freunden bedanken, die mir in dieser stressigen Zeit Rückhalt gegeben und sich die Zeit genommen haben, diese Arbeit Probe zu lesen.

Inhaltsverzeichnis

1

Einleitung

Das Internet bietet viele Möglichkeiten zur Kommunikation. Sei es über soziale Netzwerke, Email Verkehr oder sogar Telefonie mittels Voice-over-IP. Eine weitere Alternative liefert die Vernetzung von Geräten ohne direkte Benutzersteuerung. Hier kommunizieren die Geräte untereinander, tauschen Daten aus und reagieren, falls notwendig, auf Meldungen anderer Geräte. Das Internet of Things (IoT) bezeichnet die verbreitete Vernetzung von Geräten ohne direkte Benutzersteuerung. Die Möglichkeiten reichen hier von einfachem Austausch von Messdaten bis hin zum Wecker, der später klingelt, falls die Bahn Verspätung hat. Des Weiteren spielt das IoT in der Industrie eine wichtige Rolle. Sogenannte Smart Factories, Produktionsstandorte, die ohne menschliches Eingreifen weitestgehend funktionieren, sind fester Bestandteil der Industrie 4.0 [1]. Ohne eine entsprechende Vernetzung der einzelnen Geräte untereinander würde das gesamte Konzept der Smart Factories nicht funktionieren.

1.1 Problemstellung

Das Internet of Things stellt hohe Anforderungen dar, damit die Vernetzung unterschiedlicher Endgeräte reibungslos funktionieren kann. Laut einer Vorhersage des Gartner Instituts werden 2020 weltweit bis zu 26 Milliarden Geräte miteinander vernetzt sein [2]. Die Bandbreite an Gerätetypen ist hierbei sehr groß und reicht von High End Geräten bis zu kleinen Haushaltsgegenständen mit nur dürftiger Hardware Ausstattung. Daher wird im IoT eine einfache Implementierung benötigt, die auf möglichst vielen Endgeräten funktioniert. Des Weiteren muss beachtet werden, dass nicht jedes Gerät über eine solide LAN-Anbindung verfügt. Manche haben nur die Möglichkeit ein teils schlechtes WLAN- oder gar schlechtes Mobilfunknetz zu nutzen. Daher sollte eine funktionierende Verbindung auch in Gebieten mit schlechten Netzen und nur geringer Bandbreite gewährleistet sein.

1.2 Zielsetzung

Wie im vorherigen Abschnitt beschrieben wird ein Netzwerkprotokoll benötigt das möglichst alle Geräte des IoT miteinander verbinden und einen zuverlässigen Datentransport gewährleisten kann. Ziel dieser Arbeit ist ein hierfür geeignetes Protokoll vorzustellen. Außerdem werden dessen Vorteile hervorgehoben und in einem Vergleich mit alternativen Protokollen diskutiert. Anhand einer Beispielimplementierung wird die Verwendung des Protokolls veranschaulicht.

1.3 Struktur der Arbeit

Die Arbeit beginnt mit einer allgemeinen Beschreibung des MQTT Protokolls. Im darauffolgenden wird die genaue Funktionsweise des Protokolls dargestellt. Hierbei wird das komplette Kommunikationsmodell als Ganzes betrachtet und danach hierarchisch jede tiefer liegende Ebene beschrieben. Dabei wird unter anderem auf die serverseitige

Organisation der Clients und die verschiedenen Möglichkeiten, die ein Client bei der Verwendung des Protokolls hat, eingegangen. Eine detaillierte Beschreibung des Aufbaus der unterschiedlichen Datenpakete schließt dieses Kapitel ab. Nachdem die genaue Funktionsweise des Protokolls erläutert wurde, wird im darauffolgenden Abschnitt dessen Sicherheit untersucht. Anschließend werden die Vorteile des Protokolls vorgestellt. Der Frage ob diese die in 1.1 beschriebenen Probleme lösen können, wird hier besonders Beachtung geschenkt. Danach werden die Nachteile des Protokolls beschrieben, bevor im darauffolgenden Abschnitt die Verwendung von MQTT Clients am Beispiel der M2Mqtt Bibliothek vorgestellt wird. Hierbei werden alle relevanten Möglichkeiten die ein Client hat beispielhaft demonstriert. Der folgende Abschnitt stellt eine eigene Broker Implementierung vor. Hierbei wird detailliert beschrieben auf welche Aspekte bei einer solchen Programmierung besonders geachtet werden muss. Des Weiteren werden mögliche zusätzliche Funktionen eines Brokers vorgestellt, die zwar nicht zwingend notwendig, in manchen Fällen jedoch sinnvoll sind. Anschließend werden mögliche Alternativen zum beschriebenen MQTT Protokoll vorgestellt und mit diesem verglichen. Abschließend wird die Arbeit kurz zusammengefasst.

2

Was ist MQTT?

Das Message Queue Telemetry Protocol, kurz MQTT, ist ein Transportprotokoll das zuverlässigen Datenaustausch auch mit schlechter Netzanbindung garantiert. Entwickelt wurde das Protokoll 1999 von Dr. Andy Stanford-Clark (IBM) und Arlen Nipper (Arcom). Die ursprüngliche Motivation der Entwicklung war die Überwachung von Öl Pipelines in der Wüste, da die bisherige Kommunikation via Satellitentelefon zu umständlich und zu teuer war [3]. Aufgrund der geografischen Lage der Pipelines lag ein Hauptaugenmerk bei der Entwicklung des MQTT Protokolls auf dem zuverlässigen Datentransport in Netzen mit geringer Bandbreite und hoher Latenz. In der heutigen Zeit wird das MQTT Protokoll nicht mehr nur zum Austausch von Telemetriedaten, sondern für nahezu alle Bereiche der Machine-to-Machine Kommunikation, auf der das Internet of Things basiert, verwendet. Neben dem Eclipse Paho Projekt existieren mittlerweile viele Bibliotheken in verschiedenen Programmiersprachen, die eine einfache Implementierung von MQTT Diensten unterstützen. Seit 2014 gilt das MQTT offiziell als OASIS-Standard für das

Internet der Dinge (IoT) [4]. Seit 2016 ist das Protokoll außerdem ein ISO Standard (ISO/IEC 20922). Wegen der hohen Verfügbarkeit wurde das MQTT Protokoll auch von populären Internetdiensten wie dem Facebook Messenger verwendet, bevor dieser auf XMPP umstellte [5].

3

Funktionsweise

Das MQTT Protokoll ist ein Transportprotokoll, das auf TCP Verbindungen basiert. Das bedeutet, dass miteinander verbundene Geräte neben den Datenpaketen mehrere Kontrollpakete versenden. Diese bestehen sowohl aus Nachrichten die die Verbindung auf- und abbauen, als auch aus Nachrichten, die der Bestätigung einer empfangenen Nachricht dienen. Die Verwendung des TCP Netzwerkprotokolls ist ebenfalls Bestandteil der Garantie einer zuverlässigen Zustellung von Nachrichten.

3.1 Publish/Subscribe Pattern

Das Konzept von MQTT basiert auf dem Publish/Subscribe Pattern. Im Gegensatz zum Request/Response Pattern, bei dem zwischen verschiedenen Clients nicht unterschieden wird, können hier die Clients die Rollen eines Publishers, eines Subscribers oder

sogar beide Rollen einnehmen. Ein Publisher veröffentlicht Daten bzw. Nachrichten, sobald er welche hat. Ein Subscriber meldet sich beim Publisher oder im Fall des MQTT Protokolls beim Broker an und wartet darauf, dass der Publisher Daten veröffentlicht. Sobald dieser Daten sendet, empfangen alle Clients, die sich angemeldet haben, diese Nachricht. Abbildung 3.1 veranschaulicht dieses Modell. Der Vorteil, der aus der Verwen-

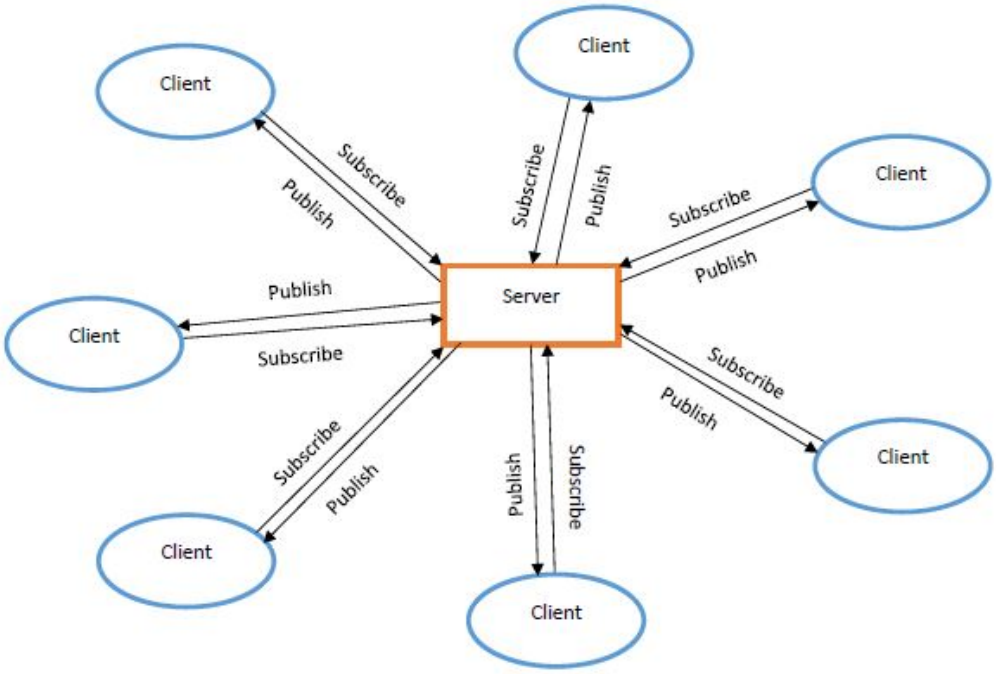

Abbildung 3.1: Publish/Subscribe Modell

dung dieses Patterns resultiert, besteht darin, das mehrere Subscriber die Nachrichten eines Publishers empfangen können. Dies ist beispielsweise für Gruppendiskussionen oder das Versenden von Daten an mehrere Geräte vorteilhaft. Dadurch, dass ein Client auch beide Rollen einnehmen kann, können so alle Parteien einer Kommunikation sowohl selbst Nachrichten versenden, als auch empfangen.

3.2 Aufbau einer MQTT Kommunikationsstruktur

Der grundlegende Aufbau eines MQTT Modells besteht aus mindestens einem Broker mit dem sich, je nach Servergröße, mehrere Clients verbinden können. Der Broker dient als Vermittler und regelt, dass die Clients genau die Nachrichten bekommen, die sie benötigen. Das bedeutet, dass zwei Clients, die kommunizieren möchten, nicht direkt miteinander verbunden sind, sondern sich nur mit demselben Broker verbinden, der die Kommunikation leitet. Bei sehr großen MQTT Netzwerken besteht zudem die Möglichkeit mehrere Broker miteinander zu verbinden. Dadurch werden die einzelnen Broker entlastet und die Arbeit auf mehrere Server verteilt [6]. Einzelne Clients, die miteinander kommunizieren möchten müssen bei dieser Möglichkeit nicht zwingend mit demselben Broker sondern nur mit demselben Netzwerk aus Brokern verbunden sein. Abbildung 3.2 stellt den beispielhaften Aufbau eines einfachen MQTT Netzwerkes dar. Die einzelnen Komponenten des Diagramms stammen von [7], [8] und [9].

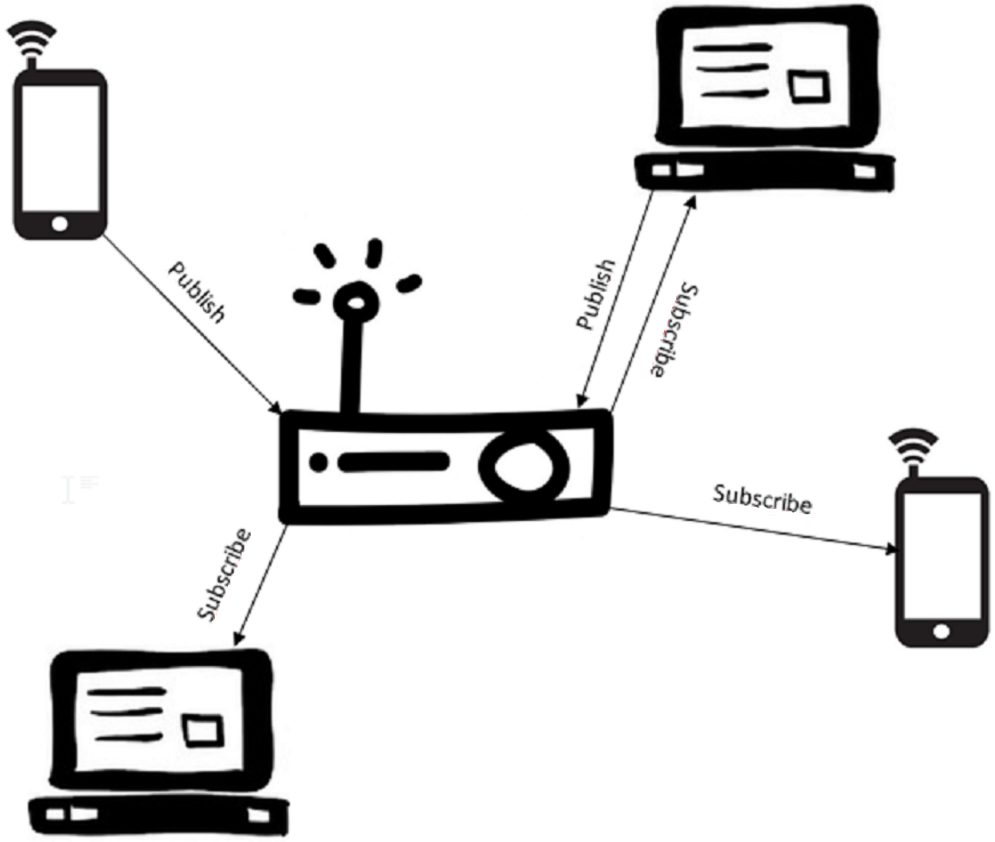

Abbildung 3.2: Architektur einer MQTT Kommunikationsstruktur

3.3 Topics

Topics sind ein wesentlicher Bestandteil in der Funktionsweise des MQTT Protokolls. Sie dienen dem Broker zum Verteilen von Nachrichten. An einem Broker können natürlich mehrere Publisher agieren, andernfalls wäre die Verwendung eines Brokers als Vermittler redundant. Da jedoch nicht alle Subscriber die Nachrichten von allen Publishern empfangen möchten, benötigt der Broker eine Möglichkeit die veröffentlichten Nachrichten richtig zuzuordnen. Die Publisher legen Topics fest, auf denen sie ihre Nachrichten veröffentlichen. Die Subscriber können ebensolche Topics beim Broker abonnieren. Dadurch weiß der Broker nun welche Abonnenten ein Topic hat und an wen er die Nachrichten weiterleiten muss. Topics können Subtopics besitzen. Diese werden hierarchisch aufgebaut, wobei die Trennung der einzelnen Stufen voneinander durch "/" erfolgt. So repräsentiert beispielsweise das Topic "Europa/Deutschland/Köln/Temperatur" die Temperatur in Köln, während das Topic "Europa/Deutschland/Köln/Einwohnerzahl" dessen Einwohnerzahl darstellt. Dadurch kann ein Publisher seine Daten über mehrere Topics verbreiten und die Subscriber können entscheiden welche Informationen sie erhalten wolle. Durch die Verwendung von Wildcards kann das Abonnieren der Topics vereinfacht werden. Mittels "+" kann eine und mit "# " mehrere Hierarchiestufen übersprungen werden. Dabei muss darauf geachtet werden, dass das "# " Zeichen nur am Ende der Topicliste verwendet werden kann. Abonniert ein Subscriber nun "Europa/+/Köln/Temperatur" erhält er dieselben Nachrichten wie ein Subscriber der "Europa/Deutschland/Köln/Temperatur" abonniert hat. Ein Subscriber der das Topic "Europa/# " abonniert erhält hingegen die Nachrichten aller Subtopics von "Europa". Daher muss ein Client, durch die Verwendung der "# " Wildcard, nicht alle Subtopics extra abonnieren. Eine Ausnahme stellen hierbei Topics dar, welche mit einem "$" Zeichen beginnen. Diese Topics sind für interne Statusnachrichten des Brokers reserviert. Der Aufbau und die Verwendung dieser Topics hängt hierbei von der jeweiligen Implementierung ab und ist von Broker zu Broker unterschiedlich. Daher können diese Topics von Clients nicht abonniert werden. Ebenso ist Clients nicht gestattet auf diesen Topics Nachrichten zu veröffentlichen [10].

3.4 Quality of Service

Der Quality of Service, kurz QoS, der bei der Verwendung des MQTT Protokolls angeboten wird, beschreibt den Grad der Zuverlässigkeit, mit der Nachrichten verschickt werden. Der Publisher kann bei jeder Nachricht, die er versendet, entscheiden wie wichtig es ihm ist, dass alle Subscriber diese Nachricht erhalten und den Quality of Service dementsprechend wählen. Ebenso können Subscriber bei jedem abonnierten Topic einen gewünschten QoS angeben, der beschreibt mit welcher Garantie dieser die Nachrichten empfangen möchte. Es existieren die drei Qualitätsstufen 0,1 und 2. Hierbei steht die Stufe 0 für keinerlei Garantie, die Nachricht wird einmal gesendet und im Weiteren nicht mehr betrachtet, wie in Abbildung 3.3 zu sehen ist. Bei den Stufen 1 und

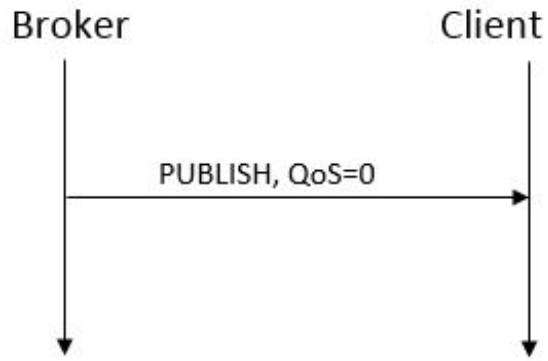

Abbildung 3.3: Quality of Service 0

2 wird hingegen eine erfolgreiche Zustellung gewährleistet, wobei bei Stufe 2 zusätzlich garantiert wird, dass die Nachricht genau einmal empfangen wird. Bei der Verwendung des Quality of Service 1 kann eine Nachricht auch mehrmals bei einem Subscriber ankommen. In Abbildung 3.4 wird das Verfahren dieses Zuverlässigkeitslevels dargestellt. Die Entscheidung ob Zuverlässigkeit oder das schonen der Ressourcen bei Netzen mit geringer Bandbreite wichtiger ist, obliegt dem Client. Zu beachten ist hierbei, dass der gewählte Quality of Service des Publishers nur indirekt mit dem QoS des Subscribers zusammenhängt, diese müssen nicht dieselben sein. Der gewählte Quality of Service eines Publishers bei einer Publish Nachricht beschriebt in erster Linie den Zuverlässigkeitsgrad der Zustellung der Publish Nachricht vom Publisher zum Broker. Sollte der

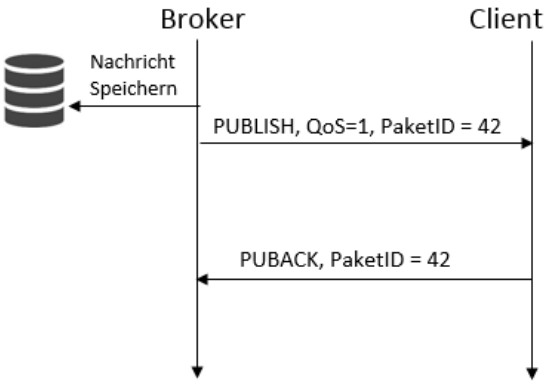

Abbildung 3.4: Quality of Service 1

Publisher das Level 0 gewählt haben, erwartet dieser keine Bestätigung des Brokers. Bei gewähltem Quality of Service 1 wartet der Publisher auf eine PUBACK Bestätigung des Brokers. Sollte diese nicht innerhalb eines bestimmten Zeitlimits beim Publisher ankommen, so sendet dieser den Publish erneut und wartet wieder auf eine Bestätigung durch den Broker. Sollte der Publisher den Zuverlässigkeitsgrad 2 gewählt haben, wartet dieser zunächst auf eine Bestätigung der empfangenen Publish Nachricht durch eine PUBREC Nachricht des Brokers. Der Broker speichert die Paket-ID der empfangenen Publish Nachricht zwischen und wartet auf eine Bestätigung dieser durch eine PUBREL Nachricht des Publishers. Erst nach erfolgreichem Empfang dieser kann der Broker die Publish Nachricht als erfolgreich angekommen ansehen und weiterverarbeiten. Dies bestätigt er noch dem Publisher gegenüber mit einer PUBCOMP Nachricht und beendet somit den Bestätigungsvorgang. Sollten hierbei Nachrichten verloren gehen, werden die Bestätigungsnachrichten erneut gesendet, jedoch mit einem Vermerk das diese öfters versendet wurden. Je nach gewähltem QoS beginnt der Broker erst nach erfolgreicher Bestätigungsprozedur mit der Verteilung der Nachrichten an die Subscriber des Topics. Im MQTT Protokoll wird der QoS der Nachrichten nicht erhöht. Dies bedeutet, dass falls Publisher und Subscriber unterschiedliche Zuverlässigkeitsgrade gewählt haben, der niedrigere QoS von beiden bei der Nachrichtenübermittlung zwischen Broker und Subscriber verwendet wird [11]. Dadurch kann es vorkommen, dass der Subscriber Nachrichten mit einem niedrigeren Quality of Service empfängt als er ursprünglich beim

Subscribe für dieses Topic angegeben hatte. Andernfalls wäre es sehr schwierig für den Broker den QoS 2 eines Subscribers zu gewährleisten, wenn, durch vom Publisher niedriger gewähltem Quality of Service, Nachrichten auch mehrfach beim Broker ankommen können. Sollte der resultierende Zuverlässigkeitsgrad 0 betragen, verschickt der Broker die Nachricht einmal an alle Abonnenten und überprüft nicht ob sie angekommen ist. Andernfalls wartet er auf eine Bestätigung der einzelnen Subscriber und schickt die Nachricht erneut, falls der Broker innerhalb eines bestimmten Zeitlimits keine Antwort erhält. Bei der Verwendung der Quality of Service 2 wird gewährleistet, dass die Nachricht genau einmal ankommt, was den Verkehr an Kontrollpaketen zusätzlich erhöht. Wenn ein Subscriber eine Nachricht mit dieser Zuverlässigkeitsgarantie empfängt, speichert er eine Referenz auf die Paket-ID zwischen und sendet dem Broker eine PUBREC Bestätigung inklusive der ID der empfangenen Nachricht. Sobald der Broker diese empfängt, antwortet er mit einer PUBREL Nachricht, nach dessen Empfang der Subscriber die ursprüngliche Nachricht speichern kann. Anschließend bestätigt er dies dem Broker gegenüber mit einer PUBCOMP Nachricht. Danach kann der Subscriber die zwischengespeicherten Informationen verwerfen. Sollte eine dieser Nachrichten verloren gehen, sendet der Broker die Nachricht des Publishers erneut, der Subscriber verwirft die zwischengespeicherten Informationen und die Bestätigungsprozedur startet erneut. Das Verfahren bei der Verwendung des Quality of Service 2 wird in Abbildung 3.5 dargestellt. Näheres zu den einzelnen Kontrollpaketen wird in 3.9 aufgeführt.

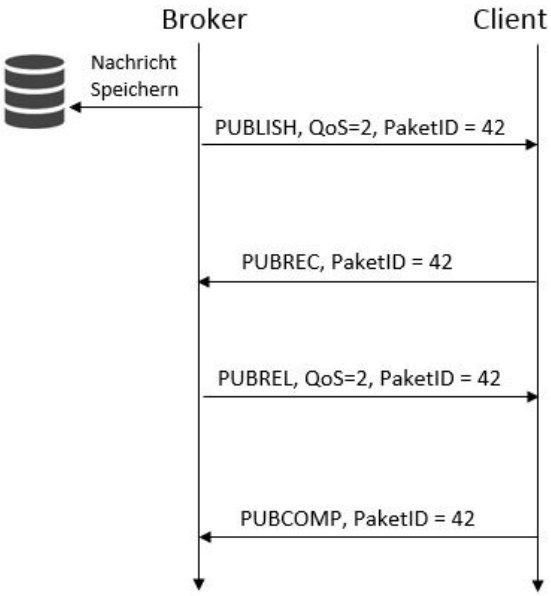

Abbildung 3.5: Quality of Service 2

3.5 Retained Messages

Das MQTT Protokoll verbindet zahlreiche Geräte auf der ganzen Welt miteinander. Durch schlechte Netzanbindung oder technische Probleme einzelner Geräte brechen dabei oft Verbindungen einzelner Clients zum Broker ab. Neu oder erneut verbundene Geräte würden bis zur ersten veröffentlichten Nachricht der abonnierten Publisher entweder keine oder nur veraltete Informationen besitzen. Um diesem Problem zu entgehen bietet das MQTT Protokoll die Möglichkeit retained Messages zu versenden. Jeder Publisher kann seine veröffentlichten Nachrichten zusätzlich mit einem retained flag versehen. Jedoch kann pro Topic nur eine Nachricht als retained gespeichert werden. Wird eine solche Nachricht veröffentlicht, erkennt dies der Broker und speichert diese inklusive ihrer QoS ab. Anschließend wird die Nachricht an alle zur Zeit verbundenen Subscriber, je nach entsprechendem Quality of Service, entsprechend versendet. Auch hier wird bei unterschiedlichen QoS bei Retained Message und Subscriber, der jeweils kleinere verwendet. Verbindet sich ein Client neu, der ein Topic mit aktuell bestehender retained Message abonniert hat, so verschickt der Broker diese sofort an den Client. Dadurch

können Clients auch Nachrichten erhalten die veröffentlicht wurden, während sie offline waren und können so auch direkt nach Verbindungsaufbau mit aktuellen Informationen weiter arbeiten. Veröffentlicht ein Publisher eine Nachricht mit gesetztem retained flag zu einem Topic, für welches der Broker bereits eine retained Message gespeichert hat, so wird die alte retained Message im Broker durch die neue überschrieben. Dieses Verfahren impliziert ebenso die Möglichkeit eine retained Message zu löschen. Um dies zu erreichen muss der Publisher eine Nachricht mit leerem Payload, also eine Nachricht ohne Dateninhalt, mit gesetztem retained flag veröffentlichen [12].

3.6 Persistente Session

Das MQTT Protokoll wurde unter anderem für den Gebrauch in Gebieten mit schlechter Netzanbindung entwickelt. Die Verbindung eines Gerätes, das nur über ein Netz mit geringer Bandbreite und hoher Latenz verfügt, ist meist sehr instabil. Damit ein Client, dessen Verbindung zum Broker regelmäßig abbricht, sich nicht jedes mal erneut anmelden und seine gewünschten Topics abonnieren muss, bietet das MQTT Protokoll den Service der persistent Session an. Ein Broker kann auf Wunsch alle wichtigen Daten einer Session speichern. Um dies zu bewerkstelligen muss der Client beim Verbindungsaufbau das cleanSession flag auf false setzen. Der Broker speichert dann alle abonnierten Topics des Clients, so wie alle Nachrichten mit einem Quality of Service von mindestens 1, von denen der Empfang beim Client noch nicht bestätigt wurde. Neben den Nachrichten, die noch während der letzten Session verschickt und nur noch nicht bestätigt wurden, speichert der Broker auch alle Nachrichten mit einem QoS von mindestens 1, die während der Client offline ist, veröffentlicht werden. Manche Brokerimplementierungen bieten zusätzlich an ankommende Nachrichten mit QoS 0 zu speichern, während der Client offline ist. Dies ist jedoch kein Standardverfahren. Folgende Informationen werden vom Broker für eine kommende Verbindung zum Client gespeichert:

- alle Topics, die der Client bisher abonniert hat
- alle noch nicht bestätigten Nachrichten mit QoS 1 oder QoS 2

- alle Nachrichten mit QoS 1 oder QoS 2, die ankommen während der Client offline ist
- (optional) alle Nachrichten mit QoS 0, die ankommen während der Client offline ist

Wenn das cleanSession flag auf true gesetzt wird, speichert der Broker keinerlei Informationen über den Client. Verbindet sich ein Client, der diese persistent Session in einer vorherigen Sitzung aktiviert hatte, dann erkennt ihn der Broker über die eindeutige Client-ID. Anschließend verschickt der Broker alle persistent Messages an den Client und fährt mit der Bestätigungsprozedur der ausstehenden Nachrichten, die einen Quality of Service von mindestens 1 haben, wie in 3.4 beschrieben fort. Damit die Bestätigung dieser Nachrichten effizient vonstattengehen kann muss der Client ebenfalls vom Broker bereits erhaltene Kontrollnachrichten bis zu einer erneuten Verbindung zwischenspeichern. Andernfalls müssten auch bereits begonnene Bestätigungsabläufe erneut gestartet werden. Da davon ausgegangen wird, dass nur Clients, die sich auch wirklich erneut verbinden, die Möglichkeit der persistent Session in Anspruch nehmen, existiert für diese gespeicherten Daten kein Ablaufdatum. Die Daten werden gespeichert bis der Client sie wieder abruft oder der Speicher des Brokers überläuft [13]. Zusätzlich bietet das MQTT Protokoll die Möglichkeit für Clients ihre Session mit einem Benutzernamen und einem Passwort zu versehen. Diese Verwendung steigert die Sicherheit des Netzwerkes, da hierdurch eine Autorisierung der Clients erforderlich ist. Mehr zur Sicherheit des MQTT Protokolls ist in Abschnitt 4 zu finden.

3.7 Last Will and Testament

Das MQTT Protokoll muss für die Verwendung im Internet of Things vielen Anforderungen gerecht werden. Speziell im industriellen Bereich, wie beispielsweise in den in 1 beschriebenen Smart Factories, in denen komplette Produktionsstandorte ohne menschliches Eingreifen funktionieren sollen, ist eine reibungslose Kommunikation der Geräte untereinander unerlässlich. Durch Software- oder Netzwerkfehler können jedoch immer wieder Verbindungen zwischen Client und Broker unerwartet abbrechen. Damit andere Geräte auf ein Ausfallen dieses Clients reagieren können, wird eine Ausnahme-

behandlung benötigt, die die anderen Clients über den ungeplanten Verbindungsabbruch informiert. Dies ist nicht nur im industriellen Einsatz äußerst wichtig, sondern auch in Bereichen in denen Menschen das Protokoll zur Kommunikation nutzen. Für diesen Zweck bietet das MQTT Protokoll die Möglichkeit eine Last Will and Testament Nachricht für ein bestimmtes Topic zu hinterlegen. Sobald die Verbindung vom Broker zu einem Client, der eine solche Nachricht gesetzt hat, unerwartet abbricht, sendet der Broker diese Nachricht an alle Abonnenten dieses Topics [14]. Dadurch werden alle interessierten Clients darüber informiert und können gegebenenfalls darauf reagieren. Es gibt wie in [15] beschrieben, vier mögliche Szenarien für den Einsatz einer Last Will and Testament Nachricht.

- Ein I/O- oder Netzwerkfehler wird vom Server erkannt.
- Ein Client antwortet nicht innerhalb der in 3.8 beschriebenen Keep Alive Zeit.
- Die Verbindung bricht ohne das Versenden einer DISCONNECT Nachricht ab.
- Der Server schließt die Verbindung aufgrund eines Protokollfehlers.

Neben der eigentlichen Verwendung der Last Will and Testament Nachricht, wird in der Praxis teilweise eine Kombination aus diesem Nachrichtentyp und der in 3.5 vorgestellten retained Messages verwendet. Dies bietet die Möglichkeit den aktuellen Status eines Clients für alle anderen Clients abzurufen. Dabei wird pro Client ein Topic erstellt, das seinen Status repräsentiert. Auf diesem wird eine retained Message hinterlegt, die den Online-Status darstellt. Wenn der Client die Verbindung absichtlich abbricht, kann er diese durch eine neue retained Message ersetzen, die den Offline-Staus beschreibt. Falls die Verbindung ungeplant abbricht, kann die hinterlegte Last Will and Testament Nachricht mit gesetztem Last Will retain Flag die Statusmeldung überschreiben und somit die anderen Clients benachrichtigen.

3.8 Keep Alive

Das MQTT Protokoll verwendet das TCP Netzwerkprotokoll, bei dessen Verwendung halb-offene Verbindungen entstehen können. Dies bedeutet, dass eine Seite abstürzt

bzw. nicht mehr als Kommunikationspartner agieren kann, ohne dass es die andere Seite erfährt. Das heißt die Verbindung vom Broker zum Client bricht ab oder ist fehlerhaft ohne, dass es der Broker bemerkt. Er verschickt also eventuell Nachrichten, die der Client letztlich nicht bekommt. Um dies zu verhindern bietet das MQTT Protokoll die Keep Alive Funktion. Dies bedeutet, dass der Client nach Ablauf des Keep Alive Timers eine PINGREQ Nachricht an den Client sendet, um die Verbindung zu testen. Die Dauer des Timers kann vom Client gesetzt werden. Durch festlegen des Timers auf 0 kann die Keep Alive Funktion auch deaktiviert werden. Wenn der Client eine PINGRESP Antwort vom Client erhält, weiß er, dass die Verbindung noch besteht und startet den Timer von vorne. Selbiges gilt wenn der Client bereits vor Ablauf des Timers eine Nachricht vom Broker empfängt. Wenn der Client jedoch keine Antwort vom Broker erhält, muss er davon ausgehen, dass die Verbindung nicht mehr einwandfrei funktioniert [16]. Daraufhin beendet er die Session und verschickt gegebenenfalls Last Will and Testament Nachrichten. Selbes gilt für den Broker. Dieser besitzt ebenfalls einen Timer mit der angegebenen Keep Alive Zeit. Wenn der Broker nach Ablauf des Timers keine Testnachricht vom Client erhält schließt der Broker die Verbindung. Jede ankommende Nachricht setzt den Timer zurück. Für die Angabe der Keep Alive Zeit sind vom Protokoll zwei Bytes vorgesehen. Diese beschreiben die Zeit in Sekunden. Dadurch ist eine Keep Alive Zeit von bis zu 65535 Sekunden wählbar, was ungefähr 18 Stunden entspricht.

3.9 Kontrollpakete

Das MQTT Protokoll verschickt verschiedene Kontrollpakete. Diese Pakete schaffen eine Grundlage für eine reibungslose Kommunikation. Dazu gehören unter anderem Pakete, die die Verbindung auf-oder abbauen. Auch Pakete die, wie in 3.4 beschrieben, den gewünschten Quality of Service sicherstellen oder wie in 3.8 erwähnt, die Keep Alive Funktion realisieren, sind Teil dieser Kontrollpakete. Der Aufbau der Pakete wird in drei Teile gegliedert. Am Anfang steht der Fixed Header, der in allen Kontrollpaketen enthalten ist. Danach folgt der Variable Header, gefolgt vom Payload Sektor. Die letzten beiden Elemente sind nicht Bestandteil jeder Nachricht.

3.9.1 Fixed Header

Der Fixed Header ist in allen Pakettypen des MQTT Protokolls enthalten. Dieser Sektor ist mindestens zwei Bytes lang. Hierbei spezifizieren die Bits 4 bis 7 den genauen Typ der Nachricht. Die Tabelle 3.1 zeigt eine Liste der unterschiedlichen Pakettypen [17]. Gesetzte Flags werden in den Bits 0 bis 3 dargestellt. Die Bedeutung der Flags ist vom Typ des Paketes abhängig. Eine Auflistung der unterschiedlichen Flags wird in der Tabelle 3.2 dargestellt [18]. Wie in der Tabelle zu sehen ist, ist das letzte Bit bei PUBLISH Nachrichten, das retain flag, gefolgt von zwei Bits, die den Quality of Service darstellen. Das dritte Bit, das in der Tabelle bei PUBLISH Nachrichten mit DUP bezeichnet wird, steht für ein mehrfaches Empfangen eines Kontrollpaketes. Nach dem ersten Byte der Nachricht beginnt die Remaining Length. Diese kann bis zu vier Bytes lang sein und steht für die Länge der restlichen Nachricht, inklusive Variable Header und Payload, in codierter Form. Die Bits, die zur Codierung der Länge benötigt werden, sind nicht Teil der Remaining Length.

3.9.2 Variable Header und Payload

Variable Header und Payload sind Sektoren, die nicht in allen Pakettypen enthalten sind. Der Inhalt des Variable Headers ist abhängig von dem verwendeten Nachrichtentyp. Bei einem Großteil der Nachrichten besteht dieser Teil aus dem Namen des Topics und der eindeutigen Pakets-ID. Die Pakets-ID wird benötigt um eingehende Nachrichten bestätigen zu können. Ein Beispiel für Nachrichten bei denen der Inhalt des Variable Headers von diesem Schema abweicht sind CONNECT Nachrichten. Diese werden aus dem Protokollnamen, der verwendeten Protokollversion, den gesetzten flags und der Keep Alive Information zusammengesetzt. Der Payload einer Nachricht beinhaltet die eigentlichen Daten der Nachricht. Im Falle einer PUBLISH Nachricht steht hier also die eigentliche veröffentlichte Nachricht [19].

Tabelle 3.1: Liste der verschiedenen Pakettypen

Name	Wert des ersten Bytes	Sender und Empfänger	Beschreibung
Reserviert	0	Verboten	Reserviert
CONNECT	1	Von Client zu Server	Verbindungsanfrage vom Client zum Server
CONNACK	2	Von Server zu Client	CONNECT Bestätigung
PUBLISH	3	Beide Richtungen möglich	PUBLISH Nachricht
PUBACK	4	Beide Richtungen möglich	PUBLISH Bestätigung
PUBREC	5	Beide Richtungen möglich	PUBLISH erhalten (bei QoS 2)
PUBREL	6	Beide Richtungen möglich	PUBREC erhalten (bei QoS 2)
PUBCOMP	7	Beide Richtungen möglich	PUBREL erhalten (bei QoS 2)
SUBSCRIBE	8	Von Client zu Server	SUBSCRIBE Nachricht vom Client
SUBACK	9	Von Server zu Client	SUBSCRIBE Bestätigung
UNSUBSCRIBE	10	Von Client zu Server	UNSUBSCRIBE Nachricht vom Client
UNSUBACK	11	Von Server zu Client	UNSUBSCRIBE Bestätigung
PINGREQ	12	Von Client zu Server	PING Anfrage
PINGRESP	13	Von Server zu Client	PING Antwort
DISCONNECT	14	Von Client zu Server	Client schließt Verbindung
Reserviert	15	Verboten	Reserviert

Tabelle 3.2: Darstellung der unterschiedlichen Flags

Pakettyp	Fixed Header Flags	Bit3	Bit2	Bit1	Bit0
CONNECT	Reserviert	0	0	0	0
CONNACK	Reserviert	0	0	0	0
PUBLISH	In MQTT Version 3.1.1 verwendet	DUP	QoS	QoS	Retain
PUBACK	Reserviert	0	0	0	0
PUBREC	Reserviert	0	0	0	0
PUBREL	Reserviert	0	0	1	0
PUBCOMP	Reserviert	0	0	0	0
SUBSCRIBE	Reserviert	0	0	1	0
SUBACK	Reserviert	0	0	0	0
UNSUBSCRIBE	Reserviert	0	0	1	0
UNSUBACK	Reserviert	0	0	0	0
PINGREQ	Reserviert	0	0	0	0
PINGRESP	Reserviert	0	0	0	0
DISCONNECT	Reserviert	0	0	0	0

4

Sicherheit

Die Sicherheit ist ein enorm wichtiges Thema in Netzwerken und somit auch im Internet der Dinge. Dieses Kapitel beschäftigt sich mit der Sicherheit bei der Verwendung des MQTT Protokolls. Das Protokoll versendet seine Daten in Form von Bytearrays und nicht in Klarschrift, wie beispielsweise das HTTP Protokoll [20]. Diese sind für dritte schwerer zu interpretieren als Nachrichten in Klarschrift, sofern diese unverschlüsselt sind. Bei der Verschlüsselung ergibt sich jedoch bereits ein Nachteil, der aus den Bytearrays resultiert. Ein Großteil der öffentlichen Broker verschlüsseln ihre Verbindungen mittels dem TLS bzw. SSL Verschlüsselungsverfahren. Bei diesem erhalten Client und Server einen gemeinsamen Schlüssel, welcher zur Verwendung eines symmetrischen Verschlüsselungsverfahrens verwendet wird. Diese Verfahren verschlüsseln jedoch nur Nachrichten in Klartext und keine Bytearrays. Somit ist eine Verschlüsselung dieser Nachrichten mittels TLS ohne Zusatzaufwand auf Client und Serverseite nicht durchführbar [20]. Jedoch bietet das Verfahren durch den initialen Austausch von Zertifikaten eine

Möglichkeit der Authentifizierung zwischen Client und Server. Somit ist die Verwendung einer TLS Verschlüsselung beim MQTT Protokoll zwar durchaus empfehlenswert, bietet aber in Protokollen, die ihre Nachrichten in Klartext verschicken mehr Sicherheit.
Das MQTT Protokoll bietet die Möglichkeit die persistente Session eines Clients mit einem Usernamen und einem Passwort zu versehen. Dadurch wird verhindert das andere User in seinem Namen agieren. Jedoch werden die Zugangsdaten standardgemäß unverschlüsselt verschickt. Eine zusätzliche Verschlüsselung der Zugangsdaten ist durchaus ratsam. Ein Nachteil ist, dass einzelne Topics nicht mit einem Passwort versehen werden können. Speziell für Topics auf denen sensible Daten veröffentlicht werden, können theoretisch von jedem beliebigen Client abonniert werden.
Letztlich ist die Sicherheit des MQTT Protokolls im Vergleich zu anderen Protokollen eher dürftig. Die Verwendung von TLS/SSL unterstützt zwar die Authentifizierung zwischen Client und Server, jedoch können die Nachrichten aufgrund deren Aufbau in Form von Bytearrays nicht ohne Zusatzaufwand verschlüsselt werden. Ein weiteres Sicherheitsproblem bietet die Tatsache, dass Topics nicht mit Passwörtern geschützt werden können. Dadurch ist für die Übertragung sensibler Daten ein eigener Broker notwendig, bei dem der Administrator den Clients beispielsweise Rechte vergibt oder nur bestimmten Clients eine Verbindung erlaubt. Somit ist das MQTT Protokoll an sich nicht wirklich sicher, kann aber durch entsprechende zusätzliche Funktionen und eigene Verschlüsselungen relativ sicher gemacht werden.

5

Vorteile

Wie in 1 erwähnt, wurde das MQTT Protokoll in erster Linie für den zuverlässigen Transport von Messdaten in Gebieten mit schlechten Netzen entwickelt. Das MQTT Protokoll ist fester Bestandteil des IoT und muss daher in der Lage sein möglichst viele Geräte miteinander zu verbinden. Dazu gehören auch Geräte mit eingeschränkten Ressourcen und nur dürftiger Hardware. Um dies zu realisieren ist die Implementierung von MQTT Clients einfach und schlank gehalten.

5.1 Auch für schlechte Netze geeignet

Mit dem MQTT Protokoll wurde ein sehr schlankes Transportprotokoll entwickelt. Durch das kompakte Nachrichtenformat mit nur geringer Nachrichtengröße und minimalem Protokolloverhead, ist nur eine geringe Bandbreite für die Nutzung des Protokolls von

Nöten. So beträgt beim MQTT Protokoll die Größe der Nachrichten ohne Payload und eindeutiger Packet-ID, wie PINGREQ, PINGRESP und DISCONNECT Nachrichten, nur 2 Bytes, während im Vergleich dazu ein einfacher HTTP Request aus 41 und eine HTTP Response Nachricht sogar schon aus mindestens 65 Bytes besteht [20]. Durch die Nutzung des Publish/Subscribe Patterns verwendet das MQTT Protokoll im Gegensatz zu anderen Protokollen kein Polling. Ankommende Nachrichten werden sofort vom Broker weiterverarbeitet und an die entsprechenden Clients weitergeleitet. Die Clients müssen nicht regelmäßig beim Server nachfragen ob neue Daten vorliegen [21]. Das reduziert ebenfalls den Datenverkehr, schont die Bandbreite und kommt somit der Verfügbarkeit des Protokolls, in Netzen unterschiedlicher Qualität, zu Gute.

5.2 Hohe Zuverlässigkeit

Ein wichtiger Aspekt eines Transportprotokolls ist die Zuverlässigkeit. Vor allem, wenn das Protokoll auch in Gebieten mit schlechter Netzanbindung verfügbar sein soll, muss mit gelegentlichem Nachrichtenverlust gerechnet werden. Jedoch wird das Protokoll auch für andere Kommunikationsmodelle verwendet, in denen den Beteiligten eine zuverlässige Datenübermittlung wichtiger als geringer Datenverkehr ist. Um diese unterschiedlichen Ansprüche unter einen Hut zu bekommen, bietet das Protokoll die Möglichkeit für jeden Client einen für sich geeigneten Quality of Service zu wählen. Dadurch können Clients mit guter Netzanbindung einen hohen QoS verwenden und so eine zuverlässige Nachrichtenübermittlung gewährleisten. Auf der anderen Seite können Clients, welche nur Netze mit geringer Bandbreite und hoher Latenz zur Verfügung haben, durch einen niedrigen Quality of Service ihre Ressourcen schützen, was der Verfügbarkeit in schlechten Netzen zu Gute kommt. Jedoch können Clients mit schlechter Netzanbindung ebenfalls einen hohen Quality of Service wählen. Trotz erhöhtem Datenverkehr kommen die Nachrichten irgendwann zuverlässig beim Client an. Die Entscheidung, ob ein zuverlässiger Datenaustausch oder die Schonung vorhandener Ressourcen wichtiger ist, liegt bei jedem Client selbst.

5.3 Sessionawareness

Mit der Sessionawareness bzw. der Verwendung einer persistent Session bietet das MQTT Protokoll eine Option, die vor allem Clients, die sich oft neu verbinden, hilft. Bei der Verwendung einer persistent Session werden, wie in Abschnitt 3.6 beschrieben, alle Daten eines Clients vom Broker gespeichert, wenn die Verbindung zu diesem abbricht. Dadurch müssen Clients bei einer erneut hergestellten Verbindung die gewünschten Topics nicht erneut abonnieren und erhalten vom Broker alle in der abgemeldeten Zeit verpassten Nachrichten. Neben der Reduzierung des Aufwands des Clients hat dieses Vorgehen zusätzlich einen entscheidenden Vorteil. Der Client verpasst keine Nachrichten in der Zeit, in der er offline ist. Speziell bei schlechter Netzanbindung, wenn die Verbindung oft abbricht, unterstützt das die Zuverlässigkeit des Protokolls. Nach erneutem Verbindungsaufbau erhält der Client die verpassten Nachrichten vom Broker. Zudem wird durch dieses Vorgehen der Datenaustausch verringert, da erneute SUBSCRIBE und UNSUBSCRIBE Nachricht nicht mehr benötigt werden. Dadurch steigt wiederum die Verfügbarkeit in schlechten Netzen.

6

Nachteile

Nachdem in Abschnitt 5 die Vorteile des Protokolls vorgestellt wurden, bezieht sich dieses Kapitel auf dessen Nachteile. Die Zuverlässigkeit des Protokolls ist nicht optimal. Wie in Abschnitt 3.4 beschrieben kann der Quality of Service einer Nachricht nicht angehoben werden. Das bedeutet, dass bei unterschiedlich gewählten QoS einer Nachricht immer der niedrigere verwendet wird. Dadurch kann der vom Subscriber gewünschte Quality of Service nicht immer eingehalten werden. Sollte der Subscriber beispielsweise ein Topic mit einem QoS von 2 abonnieren, da für ihn der Empfang der Nachricht sehr wichtig ist, der Publisher aber nur einen Quality of Service von 0 verwendet, kann es passieren, dass die Nachricht verloren geht. In diesem Fall erhält der Subscriber die Nachricht nicht, obwohl er extra einen hohen QoS gewählt hat.
Der größte Nachteil des Protokolls liegt jedoch in dessen Sicherheit, wie in Abschnitt 4 beschrieben. Die Nachrichten können nicht mittels SSL/TLS verschlüsselt werden, da sie in Form von Bytearrays und nicht als Klartext versendet werden. Das MQTT

Protokoll bietet zwar die Möglichkeit der Client-Authentifizierung mittels Usernamen und Passwort, jedoch können einzelne Topics nicht mit einem Passwort geschützt werden. Da die Nachrichten im Normalfall unverschlüsselt verschickt werden und somit auch Username, Passwort und Client-ID unverschlüsselt sind, besteht die Gefahr das dritte diese Zugangsdaten bekommen. Das Sicherheitslevel von MQTT Netzwerken hängt stark von deren Implementierung ab, da spezielle Funktionen, die die Sicherheit erhöhen, zusätzlich implementiert werden müssen und nicht Teil des Protokolls sind.

7

Implementierung von Clients

Dieses Kapitel befasst sich mit der Implementierung von MQTT Clients. Im Gegensatz zur, in Abschnitt 8 vorgestellten, Implementierung eines Brokers wird hier keine eigene Implementierung vorgestellt, sondern auf öffentliche Bibliotheken zurückgegriffen. Sollte Interesse an einer eigenen Clientimplementierung bestehen, müssen die MQTT Richtlinien und der korrekte Aufbau der Nachrichten eingehalten werden, wie in [19] beschrieben. Die Implementierung der Clients erfolgt in der Programmiersprache C#, basierend auf dem .Net Framework. Zur Erstellung der Clients habe ich die Version 4.3.0 der M2Mqtt Bibliothek von Pablo Patierno verwendet [22].
Zunächst muss das passende NuGet Paket installiert und die zugehörige using Directive im Projekt hinzugefügt werden. Als erstes wird mit Hilfe des Hostnamens oder alternativ dessen IP Adresse ein neues MqttClient-Objekt erstellt. Im nächsten Schritt kann eine Verbindung zum Broker hergestellt werden. Falls keine persistente Session verwendet wird, muss zunächst eine Client-ID erzeugt werden. Andernfalls kann auch eine statische

Client-ID übergeben werden. Folgendes Codebeispiel zeigt den Verbindungsaufbau ohne persistente Session.

```
1  string clientID = Guid.NewGuid().ToString();
2  byte code = client.Connect(clientID, // ID
3                 "username", // Username
4                 "passwort", //Passwort
5                  false, // will retain flag
6                  MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE, // will QoS
7                  true, // will flag
8                 "Maschinenstatus", // will topic
9                 "Offline", // will message
10                 true, //cleansession
11                 60); //keep alive
```

Die einzelnen Parameter der Verbindung werden bei diesem Vorgang festgelegt. Falls kein Passwortschutz der eigenen Session gewünscht wird, kann als Username und Passwort alternativ der Parameter NULL übergeben werden. Sollte ein Passwort übergeben werden, darf der Username Parameter jedoch nicht NULL sein. Der Parameter des WILL QoS kann außerdem alternativ als Integer übergeben werden.
Wenn der Client nicht nur Nachrichten veröffentlichen, sondern auch Nachrichten anderer Publisher empfangen will, muss er die gewünschten Topics, inklusive zugehörigem Quality of Service, abonnieren. Das folgende Codebeispiel stellt einen beispielhaften Subscribe dar.

```
1  client.Subscribe(
2          new string[]{"Deutschland/Ulm/Temperatur","Europa/#"},
3          new byte[]{2,0});
```

Hierbei können mehrere Topics auf einmal abonniert werden. Der Client muss nicht für jedes Topic eine eigene Subscribe Nachricht verschicken. Um empfangene Nachrichten verarbeiten zu können, muss dem Client eine Methode hinzugefügt werden, die die Daten aus der Nachricht extrahiert. Zunächst muss in der Main Methode angegeben werden welche Methode die empfangene Nachricht erhalten soll. Dies geschieht mit folgendem Befehl:

```
1  client.MqttMsgPublishReceived += client_MqttMsgPublishReceived;
```

Anschließend kann die Methode deklariert werden, welche den selben Methodennamen besitzen muss, wie zuvor in der Main Methode angegeben.

```
1 static void client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e){
2         Console.WriteLine("Subscribed for Topic = " + e.Topic + " Received Message: " +
3         Encoding.UTF8.GetString(e.Message) + "  von Sender:" + sender.ToString());}
```

Um selbst als Publisher zu fungieren und Nachrichten zu veröffentlichen, müssen die dafür notwendigen Parameter der Publish Methode übergeben werden. Das Retain Flag bezeichnet hierbei ob die Nachricht als Retained Message des Topics abgespeichert werden soll. Hierbei muss beachtet werden das Wildcards in einer Publish Nachricht nicht zulässig sind.

```
1 client.Publish("Europa/Einwohnerzahl", // topic
2                 Encoding.UTF8.GetBytes("812000000"), // message body
3                 0, // QoS level
4                 true); // retained
```

Um Topics deabonnieren zu können, müssen lediglich die gewünschten Topics der Unsubscribe Methode übergeben werden.

```
1 client.Unsubscribe(new string[] { "Europa/Einwohnerzahl" });
```

8

Implementierung eines Brokers

In diesem Kapitel werden die wichtigsten Aspekte einer Broker Implementierung anhand meiner eigenen Implementierung gezeigt. Dieser unterstützt die derzeit aktuellste Protokollversion 3.1.1. Ich habe meinen Broker in der Programmiersprache C#, basierend auf der .NET Core Plattform implementiert. Als Entwicklungsumgebung habe ich Microsoft Visual Studio 2017 verwendet. Der Broker speichert seine Daten zudem in einer lokalen MySQL Datenbank. Codebeispiele für wichtige Funktionen sind im Anhang zu finden.

8.1 Datenbankmodell

In diesem Abschnitt wird der Aufbau der verwendeten MySQL Datenbank beschrieben. Die Clients werden in der Relation "clients¨ gespeichert, in der das Feld "ClientID¨ den Primärschlüssel darstellt. Anhand dieser kann jeder Client eindeutig bestimmt werden.

Eine Liste der Parameter des Clients, wie beispielsweise Last Will Nachrichten oder die Keep Alive Zeit, wird im Feld "Parameter¨ der "clients¨ Relation hinterlegt. Im Feld ¨SubscribedTopics¨ befindet sich eine Liste der abonnierten Topics inklusive QoS. Alle Nachrichten mit einem Quality of Service 1, welche dem Client noch übermittelt werden müssen, befinden sich in der Liste des Feldes "QoS1". Selbiges gilt für Nachrichten mit einem QoS von 2 und dem zugehörigen Feld "QoS2". Im Letzten Feld "RecievedQoS2¨ der "clients¨ Relation, wird eine Liste gespeichert, welche alle vom Client empfangenen Nachrichten mit Quality of Service von 2 beinhalten und deren Erhalt dem Client gegenüber noch bestätigt werden muss. Alle vorhandenen Topics werden in der Relation "topics¨ gespeichert. Da der Name eines Topics eindeutig ist, wird dieser als Primärschlüssel im Feld "TopicName¨ gespeichert. Zu jedem in der Datenbank angelegtem Topic wird eine Liste seiner Subtopics in dem Feld ¨SubTopics¨ hinterlegt. Außerdem wird das Parenttopic, das Topic in dessen Subtopicliste sich das aktuelle Topic befindet, in dem Feld "Parent¨ abgelegt. Als oberste Topicebene wird hierbei in der Datenbank das Topic "Root¨ verwendet, dadurch werden alle Topics in der Form "Root/...¨ in der Datenbank abgelegt. Topics ohne Subtopics besitzen das Parenttopic "Root". Durch das Speichern der Parent- und Subtopics wird u.a. die Verwendung von Wildcards erleichtert. Eine Retained Message eines Topics wird in dem Feld "RetainedMessages¨ gespeichert. Das Feld ¨Subscribers¨ enthält eine Liste, welche mit Tupel aus der Client-ID und dem entsprechendem Quality of Service befüllt wird. In beiden Relationen werden die Listen in allen Feldern im Json-Format abgespeichert und müssen nach dem Auslesen aus der Datenbank deserialisiert werden. In Abbildung 8.1 ist das zugehörige Entity Relationship Diagramm zu sehen.

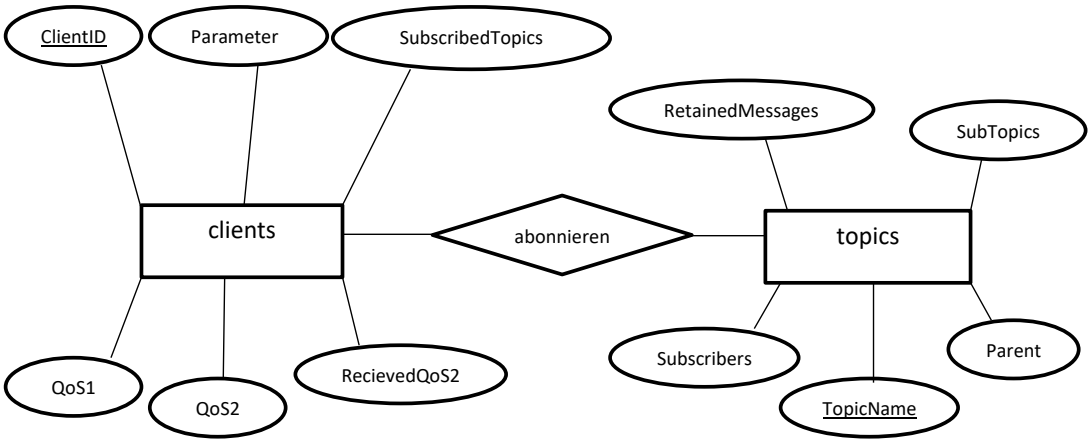

Abbildung 8.1: ER-Diagramm der Datenbank

8.2 Verbindung zu den Clients und Registrierung in der Datenbank

Nach dem der Broker gestartet wurde, wartet er mit Hilfe eines TcpListeners permanent auf eingehende TCP Verbindungen der Clients. Für jeden Client wird ein eigener Thread erstellt um Nebenläufigkeit zu ermöglichen und die Effizienz des Brokers zu steigern. Des Weiteren wird für jeden Client eine eigene Datenbankverbindung hergestellt und dem Thread mit übergeben. Während sich der neue Thread ausschließlich dem neu verbundenen Client widmet, wartet der Broker weiterhin auf neue TCP Verbindungen. Nach erfolgreichem Verbindungsaufbau empfängt der Broker eine initiale CONNECT Nachricht vom Client. Dieser interpretiert die erhaltenen Informationen, indem er die einzelnen Bytes der Nachricht betrachtet. Sollte die in der CONNECT Nachricht enthaltene Protokollversion nicht der vom Broker unterstützten Version entsprechen, so wird dies in der CONNACK Nachricht vermerkt und daraufhin die Verbindung geschlossen. Selbes gilt für nicht protokollkonforme CONNECT Nachrichten und Clients, welche versuchen sich mit falschen Zugangsdaten zu verbinden. Da die Client-ID eindeutig sein muss, wird im nächsten Schritt überprüft ob diese bereits in der Datenbank vorhanden ist. Sollte dies der Fall sein, so verwendet der Client eine persistente Session. Genaueres zu persistente Sessions wird in 3.6 beschrieben. Ein neues Client Objekt wird erzeugt, welches entweder mit neuen Clientdaten gefüllt oder im Falle einer bereits vorhandenen persistenten Session aus dieser übernommen werden. Sollte die persistent Session des Clients mit einem Usernamen und Passwort versehen sein, so wird überprüft, ob der neu verbundene Client dieselben Zugangsdaten verwendet hat. Ist dies nicht der Fall, wird die Verbindung abgebrochen und dies in der CONNACK Nachricht vermerkt. Sollte die Session bisher nicht mit Usernamen und Passwort geschützt sein, so kann der Client dies nun im Nachhinein seiner Session hinzufügen. Hierbei ist eine Verwendung von Usernamen ohne Passwort ebenso zulässig; Passwort ohne Usernamen jedoch nicht. Eine Kopie des Client Objektes inklusive TCP-Verbindung wird in der Dictionary des Brokers abgespeichert. Hierfür wird die Client-ID als Schlüssel verwendet. Anschließend wird der Client in der Datenbank hinterlegt, wobei das Client Objekt im Json-Format vorliegt. Falls der CONNECT des Clients fehlerhaft war, wird dies dem Client mitgeteilt

und die Verbindung abgebrochen. Andernfalls wird der erfolgreiche Verbindungsaufbau dem Client gegenüber durch eine CONNACK Nachricht bestätigt. Sollte der Client eine persistente Session verwenden, so werden nun die verpassten Nachrichten, sowie die retained Messages zu den abonnierten Topics versendet. Näheres dazu folgt in Abschnitt 8.7. Nach erfolgreichem Verbindungsvorgang wartet der Thread auf weitere Nachrichten des Clients. Dies geschieht solange der Client mit dem Broker verbunden ist. Ist das nicht mehr der Fall, so beendet der Broker den Thread. Wenn der Broker eine neue Nachricht vom Client erhält, betrachtet er das erste Byte der Nachricht. Dieses beschreibt den Typ der Nachricht. Je nachdem was für eine Nachricht der Broker erhalten hat ruft er die entsprechende Methode auf.

8.3 Subscribe Nachrichten

Die Durchführung eines Subscribes stellt den komplexesten Teil des Brokers dar. Nach erhaltener Subscribe Nachricht wird diese genauer betrachtet und aus den einzelnen Bytes die unterschiedlichen Informationen extrahiert. Da innerhalb einer Subscribe Nachricht mehrere Topics abonniert werden können, werden die in der Nachricht enthaltenen Topics samt zugehörigen Quality of Service in einer Liste gespeichert. Anschließend wird der Subscribe in der Datenbank durchgeführt. Jedes einzelne Topic wird separat betrachtet. Wenn das gewünschte Topic bereits in der Datenbank enthalten ist, wird der Client, inklusive gewähltem Quality of Service, in dessen Subscriber Liste aufgenommen. Andernfalls wird das Topic in der Datenbank neu angelegt. Gegebenenfalls müssen die Sub-Topics der anderen Topics aktualisiert werden. Nachdem der Client bei einem Topic als Subscriber eingetragen wurde, wird dieses Topic in seine SubscribedTopic Liste aufgenommen. Falls das Topic Wildcards enthält, wird das Topic vom "Root¨-Topic aus sukzessive aufgebaut, wobei bei jeder Wildcard alle anderen Topics mit gleichem Parenttopic ebenfalls betrachtet werden. Dadurch werden weitere Topics gefunden, die der Client über die Wildcard Funktion abonniert hat. Falls ein Topic in der Datenbank neu hinzugefügt wurde, muss überprüft werden, ob vorher Clients Topics mit Wildcards abonniert haben, welche zu dem neuen Topic passen. Hierzu wird das neue Topic ebenfalls vom Parenttopic her Stück für Stück aufgebaut. Im Gegensatz zum vorherigen

Schritt werden nun gezielt Topics gesucht, deren Wildcards die Sub-Topics des neuen Topics ersetzen können. Wurden Topics mit passenden Wildcards gefunden, so werden die Subscriber dieses Topics der SubscriberListe des neuen Topics hinzugefügt. Dieses Verfahren wird für jedes Topic durchgeführt.

8.4 Publish Nachrichten

Die empfangene PUBLISH Nachricht wird byteweise vom Broker betrachtet. Dabei werden die Packet-ID, der Nachrichteninhalt, der gewählte Quality of Service und das Topic auf dem gepublished wurde aus dem Bytearray extrahiert. Nachrichten mit einem QoS von mindestens 1 müssen dem Client gegenüber bestätigt werden, bevor der Broker sie an alle Abonnenten verteilen kann. Sollte der Zuverlässigkeitsgrad 1 gewählt worden sein, reicht es, wenn der Broker eine SUBACK Nachricht mit der Paket-ID der PUBLISH Nachricht verschickt. Im Falle eines Quality of Service von 2 muss der Broker erst eine PUBREC Nachricht verschicken, auf eine PUBREL Antwort warten und diese per PUBCOMP Nachricht bestätigen. Näheres zu dem Verfahren der jeweiligen Quality of Services ist in Abschnitt 3.4 beschrieben. Anschließend wird aus der Datenbank die Liste der Subscriber des Topics geholt. Mit Hilfe der eindeutigen Client-ID wird für jeden darin enthaltenen Client die zugehörige TCP-Verbindung aus der Dictionary des Brokers geholt. Anschließend wird der Quality of Service der PUBLISH Nachricht mit dem gewählten des Clients verglichen. Für das weitere Vorgehen wird das Minimum der beiden verwendet. Sollte aktuell keine aktive Verbindung zwischen Broker und Client bestehen, so wird die Nachricht in der entsprechenden Nachrichtenliste des Clients in der Datenbank übernommen. Diese Nachricht wird übermittelt sobald sich der Client erneut verbunden hat. Näheres dazu wird in Abschnitt 8.7 beschrieben. Im Falle einer aktuell aktiven Verbindung zwischen Broker und Client wird die Nachricht sofort versendet. Dies gilt insbesonders für einen QoS von 0. Bei einem höheren Quality of Service wird ein neuer Thread erstellt. Die verwendete Paket-ID wird in der zugehörigen Liste des Clients eingetragen. Außerdem wird die Nachricht mit Topic und verwendeter Paket-ID der jeweiligen Liste des Clients in der Datenbank hinzugefügt. Im Falle eines QoS von 1 wird die Nachricht versendet und anschließend auf eine SUBACK Bestätigung des

Clients gewartet. Nach Erhalt dieser wird die Paket-ID aus der Liste entfernt und der Thread beendet. Sollte der Broker keine Antwort erhalten, dann schickt er die Publish Nachricht mit gesetztem DUP-Flag erneut. Bei einer Nachricht mit QoS 2 wird neben Topic, Nachricht und Paket-ID noch der Status der QoS Abwicklung gespeichert. Zu Beginn befindet sich der Status auf 1, nach erhaltener PUBREC Nachricht wird dieser auf 2 erhöht. Somit weiß der Thread, der für diesen Publish zuständig ist, in welcher Phase der QoS Bestätigung er sich gerade befindet. Nach empfangener PUBCOMP Nachricht wird die Nachricht aus der Datenbank entfernt, die Paket-ID aus der Liste des Clients entfernt und der Thread beendet. Da der Versand der PUBLISH Nachrichten in einem separaten Thread durchgeführt wird, kann es passieren, dass die Verbindung zum Client abbricht, ohne dass es der Thread merkt. Daher wurde die maximale Anzahl pro Nachrichtentyp einer QoS Abwicklung auf 5 beschränkt. Wenn der Broker eine Nachricht fünfmal versendet ohne eine Reaktion des Clients zu erhalten wird der Thread ebenfalls beendet. Die Nachricht befindet sich weiterhin in der zugehörigen Liste des Clients und wird versendet sobald dieser sich erneut verbindet. Für den Fall, dass auf ein neues Topic gepublished wurde, das bisher noch nicht in der Datenbank vorhanden ist, wird überprüft ob Clients Topics inklusive Wildcards abonniert haben, die dieses Topic beinhalten. Dieses Vorgehen wird für alle Clients durchgeführt, die das Topic abonniert haben. Des Weiteren besitzt der Client eine Variable deren Wert besagt ob an diesen Client gerade eine Publish Nachricht versendet wird. Weitere Nachrichten können den Versandvorgang des jeweiligen Quality of Services erst beginnen, wenn kein anderer Thread gerade Nachrichten an diesen Client versendet. Dadurch wird bei vielen Nachrichten gleichzeitig der Datenverkehr minimiert und der Client hat genug Zeit die ankommenden Nachrichten zu verarbeiten.

8.5 Unsubscribe Nachrichten

Nach Erhalt einer UNSUBSCRIBE Nachricht beginnt der Broker die einzelnen Bytes dieser zu betrachten. Anschließend wird die Client-ID aus der Subscriber Liste von jedem in der Unsubscribe enthaltenen Topic entfernt. Der Broker erstellt eine UNSUB-ACK Nachricht um das erfolgreiche deabonnieren der Topics zu bestätigen. Um die

Bestätigungsnachricht der zuvor verschickten UNSUBSCRIBE Nachricht zuordnen zu können, muss diese dieselbe Packet-ID wie die empfangene UNSUBSCRIBE Nachricht enthalten.

8.6 Keep Alive und Ping Nachrichten

Ein in der CONNECT Nachricht mitgegebener Parameter ist die Keep Alive Zeit. Diese beschreibt die Größe des Zeitintervalls, nach dessen Ablauf der Client dem Broker ein Lebenszeichen schicken muss. Andernfalls schließt dieser die Verbindung. Nach erfolgreichem Verbindungsaufbau startet der Broker einen Timer welcher zwei Sekunden länger läuft als die angegebene keep Alive Zeit. Dieser Timer wird bei jeder empfangenen Nachricht zurückgesetzt. Sollte der Broker innerhalb des Keep Alive Intervalls keine Nachrichten erhalten, rechnet er mit einer PINGREQ Nachricht des Clients. Auf den Erhalt dieser antwortet der Broker mit einer PINGRESP Nachricht. Anschließend startet der Timer von vorne. Sollte der Broker keine PINGREQ Nachricht erhalten, muss er davon ausgehen das die Verbindung zum Client nicht mehr besteht. Anschließend schließt er die Verbindung und entfernt den Client gegebenenfalls aus der Datenbank. Sollte der Client die Last Will and Testament Funktion verwenden, so wird die Last Will Nachricht an alle Subscriber des Last Will Topics versendet. Gleiches gilt für jede andere Situation in der die Verbindung zum Client abbricht oder geschlossen wird.

8.7 Persistente Session und Retained Messages

Nachdem sich ein Client mit aktivierter Persistent Session neu verbunden hat wird überprüft ob ausstehende Nachrichten für ihn vorliegen. Anschließend werden zuerst alle Nachrichten mit Quality of Service 1 versendet. Danach folgt die Übermittlung der Nachrichten mit QoS 2. Der Versand der Nachrichten erfolgt auf dieselbe Weise wie gewöhnliche PUBLISH Nachrichten. Der genaue Vorgang ist in 8.4 beschrieben. Danach werden die retained Messages der vom Client abonnierten Topics versendet. Die Übermittlung dieser erfolgt ebenfalls mittels der in Abschnitt 8.4 vorgestellten Methode.

8.8 Potential des Brokers

In diesem Abschnitt werden Möglichkeiten vorgestellt mit denen man den Broker im Nachhinein optimieren könnte. Das in Abschnitt 8.1 vorgestellte Datenbankmodell könnte effizienter aufgebaut werden. Eine zusätzliche Relation "Messages¨ welche alle zu verschickenden Nachrichten beinhaltet, würde das Ganze übersichtlicher gestalten und die Relation "clients¨ entlasten. Mit der Verwendung der Client-ID als Primärschlüssel könnten effizient alle Nachrichten eines Clients ermittelt werden. Das Entity Relationship Diagramm dieses Datenbankmodells ist in Abbildung 8.2 zu sehen. Wie in Abschnitt 8.4 beschrieben wird der Versand einer PUBLISH Nachricht abgebrochen wenn der Client auf fünf Nachrichten des Clients nicht geantwortet hat. Für den Fall, dass die Verbindung nicht abgebrochen ist und der Client aus einem anderen Grund nicht antwortet, wäre eine Funktion von Vorteil, die in regelmäßigen Abständen testet, ob sich noch Nachrichten in der Liste des Clients befinden und diese dann versendet. Bei großem Nachrichtenandrang können neue Nachrichten ältere überholen. Durch Hinzufügen eines Zeitstempels könnte die Einhaltung der richtigen Reihenfolge auch bei vielen zu versendenden Nachrichten gewährleistet werden. Des Weiteren wäre es ratsam die Sicherheit des Netzwerkes zu erhöhen. Hierfür wäre die Verwendung von SSL Zertifikaten, zur Authentifizierung, in Verbindung mit einer, für Bytearrays geeigneten, Verschlüsselung ratsam. Wie in Abschnitt 4 erwähnt ist ebenso ein Rechtesystem für Clients denkbar um Topics mit wichtigen Daten zu schützen.

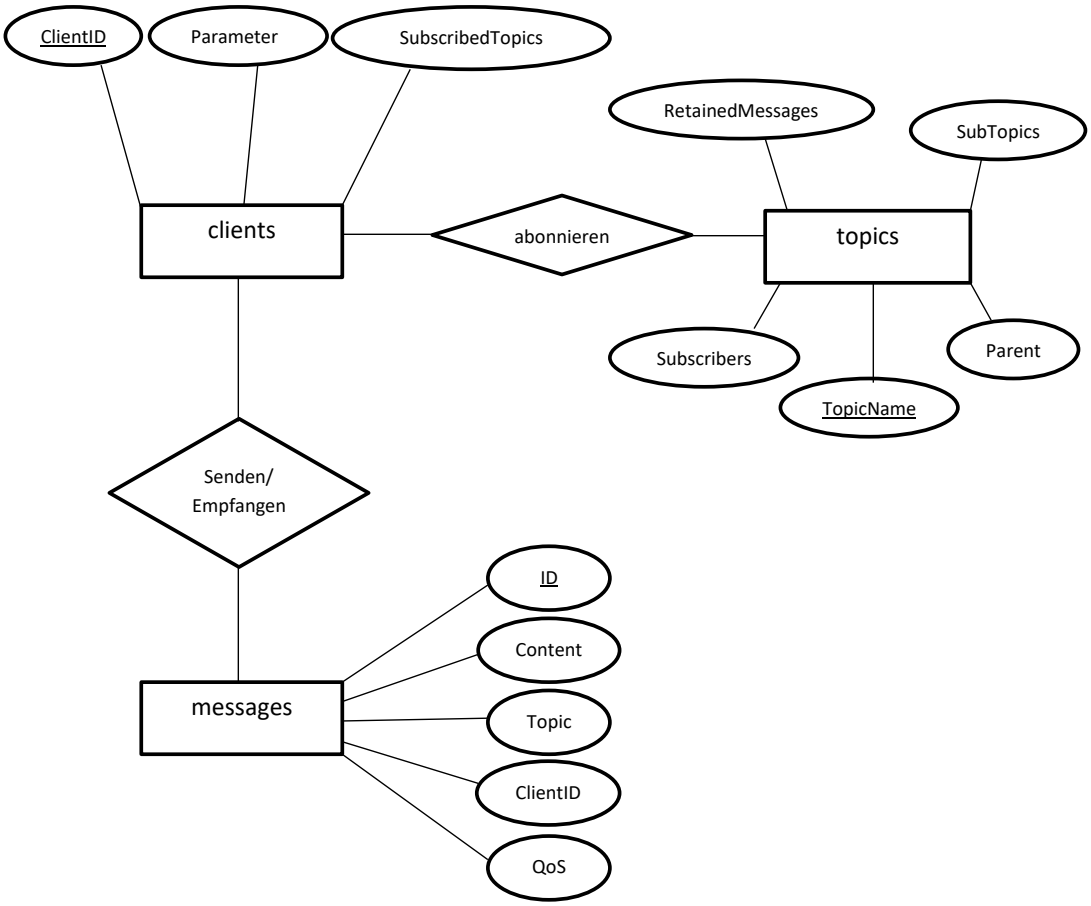

Abbildung 8.2: ER-Diagramm des effizienteren Datenbankmodells

9

Alternative Protokolle

Das MQTT Protokoll wurde für eine zuverlässige Datenübertragung in schlechten Netzen entwickelt und ist mittlerweile fester Bestandteil des Internet der Dinge. In diesem Abschnitt werden alternative IoT Protokolle vorgestellt und mit dem MQTT Protokoll verglichen.

9.1 CoAP

Das Constrained Application Protokoll CoAP ist eine resourcenschonende und effiziente Variante des HTTP Protokolls. Wie beim HTTP Protokoll wird das Request/Response Pattern verwendet. Das CoAP Protokoll basiert, im Gegensatz zum MQTT Protokoll, auf einer UDP Verbindung mit DTLS Verschlüsselung [23]. Die Observe Funktion des CoAP

Protokolls benachrichtigt Clients über Änderungen spezieller Ressourcen des Servers und ist somit grob mit der Last Will Funktion des MQTT Protokolls vergleichbar.

9.1.1 Vorteile gegenüber MQTT

Im Gegensatz zu MQTT ist CoAP mit HTTP kompatibel. Durch den Einsatz von Proxys können auch HTTP Clients mit einem CoAP Server kommunizieren. Für die umgekehrte Konstellation gilt dasselbe. Dadurch müssen, wie beim MQTT Protokoll, keine zusätzlichen Clients implementiert werden [24].

9.1.2 Nachteile gegenüber MQTT

Die Verwendung des Request/Response Patterns ist ein Nachteil gegenüber des MQTT Protokolls. Es steigt der Aufwand und Datenverkehr des Clients, da dieser durch Polling regelmäßig nachfragen muss ob neue Daten für ihn vorliegen. Beim MQTT Protokoll hingegen empfängt der Client alle relevanten Nachrichten sofort. Durch die UDP Verbindung des CoAP Protokolls ist zudem die Datenübertragung unzuverlässiger als beim MQTT Protokoll, welches auf TCP Verbindungen basiert.

9.2 XMPP

Das Extensible Messaging and Presence Protocol (XMPP), früher auch Jabber genannt, ist ein IoT Protokoll auf XML Basis. Für das Protokoll existieren zahlreiche unterschiedliche Implementierungen, die verschiedene Erweiterungen beinhalten [24]. Hauptsächlich wird das Protokoll für Chatprogramme verwendet, da es neben einer hohen Sicherheit auch verschiedene Funktionen wie etwa Videoanrufe zur Verfügung stellt.

9.2.1 Vorteile gegenüber MQTT

Wie in Abschnitt 6 beschrieben ist der größte Nachteil des MQTT Protokolls dessen Sicherheit. Das XMPP Protokoll hingegen bietet seinen Benutzern einen sehr hohen

Sicherheitsstandard und wird sogar als sicherster Nachrichtenstandard beworben [25]. Theoretisch ist der Versand von Bild und Videodateien über MQTT zwar auch möglich, jedoch ist das nicht empfehlenswert, da das Protokoll dafür nicht ausgelegt ist. Es wurde entwickelt um effizient mit möglichst geringer Bandbreite zu arbeiten. Daher ist das XMPP Protokoll für Chatprogramme besser geeignet als das MQTT Protokoll.

9.2.2 Nachteile gegenüber MQTT

Im Punkt Nachrichtengröße kann das MQTT Protokoll gegenüber XMPP mit seinem schlanken Aufbau Punkten. Da XMPP auf XML basiert besitzt dieses Protokoll einen relativ großen Protokolloverhead. Durch die zahlreichen unterschiedlichen Implementierungen, welche die unterschiedlichsten Protokollerweiterungen beinhalten, werden bei der Benutzung des XMPP Protokolls auch Clients benötigt, die dieselben Erweiterungen unterstützen. Beim MQTT Protokoll hingegen kann im Normalfall jeder Client mit jedem Broker kommunizieren.

9.3 AMQP

AMQP ist ein IoT Protokoll, das dem MQTT Protokoll relativ ähnlich ist. Auch hier existiert ein zentraler Broker, der den korrekten Nachrichtenaustausch regelt. Die sogenannte Börse im Broker fügt die Nachrichten zu den Warteschlangen hinzu. Die Zuteilung erfolgt über Regeln und der Wahl eines Austauschformates. Das AMQP bietet vier verschiedene Austauschformate an. Neben dem Topic Austauschformat, welches der Funktionsweise des MQTT Protokoll ähnelt, gibt es ein Broadcast Format, das alle existierenden Warteschlangen anspricht und zwei Formate, die nur eine bestimmte Warteschlange ansprechen, welche sich nur bedingt in Zusatzmöglichkeiten unterscheiden. Wie beim MQTT Protokoll können Warteschlangen abonniert werden. Alternativ kann ein Client auch Nachrichten einer Warteschlange anfordern, wenn er diese benötigt [26].

9.3.1 Vorteile gegenüber MQTT

Während das MQTT Protokoll nur das Publish/Subscribe Pattern unterstützt, bietet das AMQP Protokoll dem Client mehrere Möglichkeiten zur Nachrichtenübermittlung. Vor allem die Möglichkeit das Clients Warteschlangen nicht abonnieren müssen, sondern selbst entscheiden können, wann sie Nachrichten abrufen möchten, ist im MQTT Protokoll nicht gegeben.

9.3.2 Nachteile gegenüber MQTT

Während das MQTT Protokoll relativ einfach aufgebaut ist, ist das AMQP Protokoll recht komplex. Auch beim Vergleich der minimalen Nachrichtengröße der beiden Protokolle schneidet das MQTT Protokoll mit 2 Bytes gegenüber den 60 Bytes des AMQP Protokolls deutlich besser ab. Dadurch ist das MQTT Protokoll für den Einsatz in schlechten Netzen mehr zu empfehlen, da es deutlich effizienter und ressourcenschonender als das AMQP Protokoll ist.

10 Fazit

Das MQTT Protokoll wurde entwickelt um einen zuverlässigen Datentransport in unzuverlässigen Netzen zu ermöglichen. Durch seine geringe Nachrichtengröße benötigt das Protokoll nur wenig Bandbreite. Mit zusätzlichen Funktionen wie dem Quality of Service und der Option einer persistenten Session wird die Zuverlässigkeit ebenfalls unterstützt. Zudem werden durch die relativ einfache Implementierung nur sehr wenig Ressourcen benötigt, was die Verwendung auf möglichst vielen Gerätetypen ermöglicht. Somit wird das Protokoll seinen Anforderungen gerecht. Für ein Protokoll des Internet der Dinge liefert MQTT leider nur eine geringe Sicherheit. Durch den Nachrichtenaufbau in Form von Bytearrays sind diese leider nicht mittels SSL/TLS verschlüsselbar. Dadurch ist ein sicheres MQTT Netzwerk nur durch Zusatzaufwand des Brokers und der Clients realisierbar. Viele Implementierungen bieten die Möglichkeit, die Sicherheit durch zusätzliche Optionen zu erhöhen. So beinhaltet die Implementierung von Mosquitto beispielsweise eine Möglichkeit, den Benutzern Rechte zuzuweisen [27]. Aufgrund der

höheren Sicherheit und den zusätzlichen Möglichkeiten ist das in Abschnitt 9.2 vorgestellte XMPP Protokoll für Chatprogramme besser geeignet. Abschließend kann gesagt werden das das MQTT Protokoll wohl das effizienteste und ressourcenschonendste IoT Protokoll darstellt. Daher ist es für den Einsatz in Netzen mit geringer Bandbreite und hoher Latenz durchaus zu empfehlen und erfüllt damit seine Anforderungen.

Literaturverzeichnis

[1] Weisbecker, A., Burmester, M., Schmidt, A.: Mensch und Computer 2015– Workshopband. Walter de Gruyter GmbH & Co KG (2015)

[2] : Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. (http://www.gartner.com/newsroom/id/2636073) Aufrufdatum: 20.05.2017.

[3] : MQTT, Das M2M und IoT Protokoll. (https://www.predic8.de/mqtt.htm) Aufrufdatum: 10.05.2017.

[4] : MQTT Version 3.1.1 becomes an OASIS Standard. (https://www.oasis-open.org/news/announcements/mqtt-version-3-1-1-becomes-an-oasis-standard) Aufrufdatum: 10.05.2017.

[5] : MQTT-Protokoll: IoT-Kommunikation von Reaktoren und Gefängnissen öffentlich einsehbar. (https://www.heise.de/security/meldung/MQTT-Protokoll-IoT-Kommunikation-von-Reaktoren-und-Gefaengnissen-oeffentlich-einsehbar-3629650.html) Aufrufdatum: 04.10.2017.

[6] : Bridges. (https://www.cloudmqtt.com/docs-bridge.html) Aufrufdatum: 10.10.2017.

[7] : Who's Leading in the Smartphone Race? (http://socialwebqanda.com/2013/08/whos-leading-in-the-smartphone-race/) Aufrufdatum: 13.05.2017.

[8] : Computer Hand gezeichnet Werkzeug Kostenlose Icons. (http://de.freepik.com/freie-ikonen/computer-hand-gezeichnet-werkzeug_785226.htm) Aufrufdatum: 13.05.2017.

[9] : Radio flachen Hand gezeichnet Werkzeug mit einer Antenne Kostenlose Icons. (http://de.freepik.com/freie-ikonen/radio-flachen-hand-gezeichnet-werkzeug-mit-einer-antenne_734709.htm) Aufrufdatum: 13.05.2017.

[10] : MQTT Essentials Part 5: MQTT Topics und Best Practices. (https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices) Aufrufdatum: 10.06.2017.

[11] : MQTT Essentials Part 6: Quality of Service 0, 1 und 2. (https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels) Aufrufdatum: 10.06.2017.

[12] : Understanding Retained Messages -MQTT by Example. (http://www.steves-internet-guide.com/mqtt-retained-messages-example/) Aufrufdatum: 13.06.2017.

[13] : MQTT Essentials Part 7: Persistent Session and Queuing Messages. (https://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages) Aufrufdatum: 15.06.2017.

[14] : MQTT Last Will and Testament. (http://dev.iachieved.it/iachievedit/mqtt-last-will-and-testament/) Aufrufdatum: 23.06.2017.

[15] : MQTT Essentials Part 9: Last Will and Testament. (https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament) Aufrufdatum: 15.06.2017.

[16] : MQTT Keep Alive -By Example. (http://www.steves-internet-guide.com/mqtt-keep-alive-by-example/) Aufrufdatum: 24.06.2017.

[17] : MQTT Version 3.1.1 Plus Errata 01. (http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html#_Toc385349209) Aufrufdatum: 03.07.2017.

[18] : MQTT Version 3.1.1 Plus Errata 01. (http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html#_Toc385349211) Aufrufdatum: 03.07.2017.

[19] : MQTT Version 3.1.1 Plus Errata 01. (http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html) Aufrufdatum: 03.07.2017.

[20] : MQTT Sicherheit. (http://androegg.de/?page_id=1429) Aufrufdatum: 17.05.2017.

[21] : IoT-Allrounder. (https://jaxenter.de/iot-allrounder-27208) Aufrufdatum: 26.09.2017.

[22] : M2Mqtt 4.3.0 M2Mqtt - MQTT Client Library for .Net and WinRT. (https://www.nuget.org/packages/M2Mqtt/) Aufrufdatum: 12.08.2017.

[23] : CoAP – Das zukünftige Standard-Protokoll für das Internet of Things? (https://blog.doubleslash.de/coap-das-zukuenftige-standard-protokoll-fuer-das-internet-of-things/) Aufrufdatum: 04.10.2017.

[24] : IoT-Protokolldschungel – Ein Wegweiser. (https://www.informatik-aktuell.de/betrieb/netzwerke/iot-protokolldschungel-ein-wegweiser.html) Aufrufdatum: 04.10.2017.

[25] : The most secure messaging standard. (https://xmpp.org/) Aufrufdatum: 04.10.2017.

[26] : IoT Protokolle – MQTT vs. AMQP . (https://www.sic-software.com/iot-protokolle-mqtt-vs-amqp/) Aufrufdatum: 05.10.2017.

[27] : Verschlüsseltes MQTT vom und zum Mosquitto-Server. (https://www.auxnet.de/verschluesseltes-mqtt-vom-und-zum-mosquitto-server/) Aufrufdatum: 12.05.2017.

A

Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```
1  private static void Main(string[] args)
2    {
3        //Broker startet und wartet auf TCP Verbindungen
4        var broker = new Broker();
5        var listener = new TcpListener(IPAddress.Any, 1883);
6        listener.Start();
7        var data = new Database();
8        while (true)
9        {
10           Console.WriteLine("Waiting for client...");
11           try
12           {
13               var clientTask = listener.AcceptTcpClientAsync();
14               var client = clientTask.Result;
15               if (clientTask.Result != null)
16               {
```

```
17                  //Pro Client eigener Thread mit eigener Datenbankverbindung
18                  Console.WriteLine("Client connected. Waiting for data.");
19                  var messages = new Message();
20                  var writer = new BinaryWriter(client.GetStream());
21                  var reader = new BinaryReader(client.GetStream());
22                  var connection = data.ConnectToDatabase();
23                  client = clientTask.Result;
24                  var thread = new Thread(() =>
25                  broker.Listen(client, messages, writer, reader, connection));
26                  thread.Start();
27              }
28          }
29          catch (Exception e)
30          {
31              Console.WriteLine(e.StackTrace);
32          }
33      }
```

Listing A.1: Verbindungsaufbau

```
1   public Client ClientConnect(string []clientParam, IPAddress ip,
2   MySqlConnection con, TcpClient tcpclient,Broker broker,
3   Message messageObj,BinaryWriter writer)
4      {
5          var clientId = clientParam[0];
6          var willFlag = 0;
7          var willQoS = 0;
8          var willRetain = 0;
9          var willTopic = "";
10         var willMessage = "";
11         var cleanSession = 0;
12         var keepAlive = 0;
13         var username = "";
14         var password = "";
15         var stm = "SELECT Parameter FROM clients WHERE ClientID='" + clientId + "';";
16         var comdand = new MySqlCommand(stm, con);
17         var read = comdand.ExecuteReader();
18         if (read.HasRows)
19         {
20             //client wird aus db uebernommen falls persistentSession aktiv ist
21             Client tempClient = null;
22             try
23             {
```

```
                while (read.Read())
                {
                    tempClient = JsonConvert.DeserializeObject
                    <Client>(read.GetString(0));
                }
                read.Dispose();
                comdand.Dispose();
                if (!tempClient.username.Equals("False"))
                {
                    if (!tempClient.username.Equals(clientParam[8])
                    || (!tempClient.password.Equals(clientParam[9])
                    &&!tempClient.password.Equals("False")))
                    {
                        var connack = messageObj.Connack(false, false, false, true);
                        Send(connack, tcpclient, writer);
                        return null;
                    }
                }
                if(tempClient.username.Equals("False")
                ||(tempClient.password.Equals("False")
                && tempClient.username.Equals(clientParam[8])))
                {
                    tempClient.username = clientParam[8];
                    tempClient.password = clientParam[9];
                    var clientJson = JsonConvert.SerializeObject(tempClient);
                    var cmd = new MySqlCommand();
                    cmd.Connection = con;
                    //username/password wird in Datenbank aktualisiert
                    cmd.CommandText = "UPDATE clients SET Parameter =’"
                    + clientJson + "’ WHERE ClientID=’"
                    + tempClient.id + "’;";
                    cmd.Prepare();
                    cmd.ExecuteNonQuery();
                }
                dictionary.Remove(tempClient.id);
                var dictionaryClient = new Client(tempClient.id,
                tempClient.willMessage, tempClient.willTopic,
                tempClient.willRetain, tempClient.willQoS, tempClient.willFlag,
                tempClient.cleanSession, tempClient.keepAlive,
                clientParam[8], clientParam[9], tcpclient,
                tempClient.retainedList, tempClient.packetIdList, true);
                dictionary.Add(tempClient.id,dictionaryClient);
```

```
66                  if (clientParam[6].Equals("True")){
67                      cleanSession = 1;
68                  }
69                  else{
70                      cleanSession = 0;
71                  }
72                  if (tempClient.cleanSession != cleanSession){
73                      dictionaryClient.cleanSession = cleanSession;
74                      tempClient.cleanSession = cleanSession;
75                      tempClient.username = clientParam[8];
76                      tempClient.password = clientParam[9];
77                      var clientJson = JsonConvert.SerializeObject(tempClient);
78                      var cmd = new MySqlCommand();
79                      cmd.Connection = con;
80                      //Client wird in Datenbank aktualisiert
81                      cmd.CommandText = "UPDATE clients SET Parameter ='"
82                      +clientJson+"' WHERE ClientID='"+tempClient.id+"';";
83                      cmd.Prepare();
84                      cmd.ExecuteNonQuery();
85                  }
86                  return dictionaryClient;
87              }
88              catch (KeyNotFoundException){
89                  return null;
90              }
91          }
92              read.Dispose();
93              comdand.Dispose();
94          if (clientParam[3].Equals("True")){
95              willFlag = 1;
96          }
97                  if (clientParam[1].Equals("True")){
98                      willRetain = 1;
99                  }
100                 if (clientParam[2].Equals("1"))
101                 {
102                     willQoS = 1;
103                 }else if (clientParam[2].Equals("2")){
104                     willQoS = 2;
105                 }
106                 willTopic = clientParam[4];
107                 willMessage = clientParam[5];
```

```
108                 if (clientParam[6].Equals("True")){
109                     cleanSession = 1;
110                 }
111                 keepAlive = Int32.Parse(clientParam[7]);
112                 username = clientParam[8];
113                 password = clientParam[9];
114                 var subtopics = new List<Tuple<string,int>>();
115                 var retainList = new List<string>();
116                 var packetIdList = new List<int>();
117                 var tempclient = new Client(clientId, willMessage,
118                 willTopic, willRetain, willQoS, willFlag,
119                 cleanSession, keepAlive, username,password,tcpclient,
120                 retainList,packetIdList,true);
121             try{
122               //Client mit Tcp Verbindung wird in dictionary gespeichert
123               broker.dictionary.Add(clientId, tempclient);
124               //Client ohne verbindung wird in db gespeichert
125               var client = new Client(clientId, willMessage, willTopic,
126               willRetain, willQoS, willFlag, cleanSession, keepAlive,
127               username, password, null,retainList,packetIdList,true);
128               var parameterJson = JsonConvert.SerializeObject(client);
129               var topicsJson = JsonConvert.SerializeObject(subtopics);
130               var cmd = new MySqlCommand();
131               cmd.Connection = con;
132               //Client wird in Db eingefuegt
133               cmd.CommandText = "INSERT INTO clients
134               (ClientID,Parameter,SubscribedTopics) VALUES('"
135               + clientId + "','" + parameterJson + "','"
136               + topicsJson + "');";
137               cmd.Prepare();
138               cmd.ExecuteNonQuery();
139               Console.WriteLine("IP: " + ip);
140               return client;
141             }
142             catch (Exception){
143               return null;
144             }
145         }
```

Listing A.2: Clientregistrierung in der Datenbank

```
1   public string GetMessageType(byte[] message)
2         {
```

```
3             var firstByte = Convert.ToString(message[0], 2);
4             //Byte mit fuehrenden Nullbits auffuellen
5             while (firstByte.Length < 8)
6             {
7                 firstByte="0"+firstByte;
8             }
9             Console.WriteLine(firstByte);
10            if (firstByte.StartsWith("1000"))
11            {
12                if (firstByte.Equals("10000010"))
13                {
14                    return "Subscribe";
15                }
16                else return "ERROR";
17            }
18            else if(firstByte.Equals("11000000")){
19                return "Pingreq";
20            }
21            else if(firstByte.StartsWith("0011")){
22                return "Publish";
23            }else if (firstByte.Equals("01000000"))
24            {
25                return "Puback";
26            }else if (firstByte.Equals("01010000"))
27            {
28                return "Pubrec";
29            }else if (firstByte.Equals("01110000"))
30            {
31                return "Pubcomp";
32            }
33            else if (firstByte.Equals("01100010"))
34            {
35                return "Pubrel";
36            }
37            else if (firstByte.Equals("10100010"))
38            {
39                return "Unsubscribe";
40            }else if (firstByte.Equals("11100000"))
41            {
42                return "Disconnect";
43            }
44            else
```

```
45              {
46                  return "ERROR";
47              }
48          }
```

Listing A.3: Nachrichtentyp wird ermittelt

```
1  while (message != null && tcpClient.Connected && clientOnline)
2        {
3            try
4            {
5                var buffer = new byte[1024];
6                reader.Read(buffer, 0, buffer.Length);
7                var messageType = messagesObj.GetMessageType(buffer);
8                Console.WriteLine("Messagetype: " + messageType);
9                switch (messageType)
10               {
11                   case "Subscribe":
12                       timer.Change((client.keepAlive * 1000) + 2000, 0);
13                       if (messagesObj.Subscribe(buffer, tcpClient, broker,
14                       writer,reader,con,client,messagesObj)) { }
15                       else
16                       {
17                           Error(reader, writer, tcpClient, "Ungueltiger Subscribe,
18                           Connection closed",con,client, messagesObj);
19                           clientOnline = false;
20                        }
21                        break;
22                    case "Pingreq":
23                       timer.Change((client.keepAlive * 1000) + 2000, 0);
24                       var ping = messagesObj.Pingresp();
25                       Send(ping, tcpClient, writer);
26                       break;
27                   case "Disconnect":
28                       Error(reader, writer, tcpClient,
29                       "Client hat die Verbindung geschlossen",con,client,messagesObj);
30                       clientOnline = false;
31                       break;
32                   case "Publish":
33                       timer.Change((client.keepAlive * 1000) + 2000, 0);
34                       messagesObj.Publish(buffer,tcpClient,broker,
35                       writer,messagesObj,con,client);
36                       break;
```

```
37                  case "Pubrel":
38                      timer.Change((client.keepAlive * 1000) + 2000, 0);
39                      Console.WriteLine("PUBREL!");
40                      if (PubrelReceived(buffer, client, con))
41                      {
42                          Send(messagesObj.Pubcomp(buffer), tcpClient, writer);
43                      }
44                      break;
45                  case "Pubrec":
46                      timer.Change((client.keepAlive * 1000) + 2000, 0);
47                      Console.WriteLine("PUBREC!");
48                      PubrecReceived(buffer, client, con);
49                      break;
50                  case "Puback":
51                      timer.Change((client.keepAlive * 1000) + 2000, 0);
52                      Console.WriteLine("PUBACK!");
53                      PubackReceived(client, con,buffer);
54                      break;
55                  case "Pubcomp":
56                      timer.Change((client.keepAlive * 1000) + 2000, 0);
57                      PubcompReceived(buffer, client, con);
58                      break;
59                  case "Unsubscribe":
60                      timer.Change((client.keepAlive * 1000) + 2000, 0);
61                      var unsuback = messagesObj.Unsubscribe(buffer,con,client,broker);
62                      if (unsuback != null)
63                      {
64                          Send(unsuback, tcpClient, writer);
65                      }
66                      break;
67                  case "ERROR":
68                      if (keepAliveSemaphore)
69                      {
70                          while (keepAliveSemaphore)
71                          {
72                          }
73                          break;
74                      }
75                      else{
76                          keepAliveSemaphore = true;
77                          timer.Dispose();
78                          Error(reader, writer, tcpClient, "Fehlerhafte Clientnachricht,
```

```
                        Connection closed", con, client, messagesObj);
                        clientOnline = false;
                        keepAliveSemaphore = false;
                        break;
                        }
                }
                message = Encoding.ASCII.GetString(buffer);
                Console.WriteLine(message);
            }
            catch (Exception)
            {
               if (keepAliveSemaphore)
               {
                   while (keepAliveSemaphore)
                   {
                   }
                   break;
               }
               else
               {
                   keepAliveSemaphore = true;
                   timer.Dispose();
                   if (clientOnline)
                   {
                       Error(reader, writer, tcpClient, "Clientverbindung abgebrochen",
                       con, client, messagesObj);
                       clientOnline = false;
                   }
               }
           }
           keepAliveSemaphore = false;
       }
```

Listing A.4: Warten und Reagieren auf eingehende Nachrichten

```
public bool Subscribe(List<Tuple<string, int>> topicList, MySqlConnection con,
Client client,Message messageObj,BinaryWriter writer)
{
   try
   {
      //topics werden nacheinander betrachtet
      for (var i = 0; i < topicList.Count; i++)
      {
```

```
9           Console.WriteLine("TOPIC " + topicList[i].Item1);
10          var topics = topicList[i].Item1;
11          var bytes = Encoding.ASCII.GetBytes(topics);
12          var counter = 0;
13          for (var lengthCounter = 0; lengthCounter < bytes.Length; lengthCounter++)
14          {
15              Console.WriteLine(bytes[lengthCounter]);
16              if (bytes[lengthCounter] == 0)
17               {
18                 break;
19               }
20              counter++;
21          }
22          var newTopicByte = new byte[counter];
23          for (var update = 0; update < counter; update++)
24          {
25              newTopicByte[update] = bytes[update];
26          }
27          topics = Encoding.ASCII.GetString(newTopicByte);
28          var qos = topicList[i].Item2;
29          var splittedTopics = topics.Split('/');
30          var parentTopic = "Root";
31          //topic wird in subtopics aufgeteilt
32          for (var j = 0; j < splittedTopics.Length; j++)
33          {
34              //topic wird in datenbank gesucht
35              Console.WriteLine("SUB " + splittedTopics[j]);
36              var topicName = parentTopic + "/" + splittedTopics[j];
37              if (topicName.Equals("Root/#"))
38              {
39                 WildcardHelper(con, "Root", client, qos,messageObj,writer);
40              }
41              WildcardUpdater(con, topicName, client, qos,messageObj,writer);
42              Console.WriteLine("PARENTTOPIC " + topicName);
43              var stm = "SELECT TopicName FROM topics
44              WHERE TopicName='" + topicName + "';";
45              var comd = new MySqlCommand(stm, con);
46              var reader = comd.ExecuteReader();
47              if (reader.HasRows)
48             {
49                  while (reader.Read())
50                  {
```

```
51                      parentTopic = reader.GetString(0);
52                  }
53                  reader.Dispose();
54                  comd.Dispose();
55                  if (j == splittedTopics.Length - 1)
56                  {
57                  //falls topic bereits vorhanden und # wildcard subscribed wurde,
58                   subscriber wird in allen relevanten subtopics eingetragen
59                   if (parentTopic.EndsWith("#"))
60                   {
61                      var subTopicSplit = parentTopic.Split('/');
62                      var topic = "";
63                      for(var subCount = 0;subCount < subTopicSplit.Length;subCount++)
64                      {
65                          if (subCount == subTopicSplit.Length - 1)
66                          {
67                          }
68                          else
69                          {
70                              if (topic.Contains("Root"))
71                               {
72                                  topic = topic + "/" + subTopicSplit[subCount];
73                               }
74                               else
75                               {
76                                  topic = topic + subTopicSplit[subCount];
77                              }
78                          }
79                      }
80                      WildcardHelper(con, topic, client, qos, messageObj, writer);
81                   }
82                    //topic bereits vorhanden, kein wildcard subscribe
83                      var statement = "SELECT Subscribers FROM topics
84                      WHERE TopicName='" + parentTopic +"';";
85                      var cmd = new MySqlCommand(statement, con);
86                      var read = cmd.ExecuteReader();
87                      var subscriberList = new List<Tuple<string, int>>();
88                      while (read.Read())
89                      {
90                         try
91                          {
92                             subscriberList =JsonConvert.DeserializeObject
```

```
93                              <List<Tuple<string, int>>>(read.GetString(0));
94                             }
95                            catch(Exception){}
96                            }
97                        read.Dispose();
98                        cmd.Dispose();
99                        var alreadySubscribed = false;
100                       //test ob client bereits als subscriber eingetargen ist
101                       for (var t = 0; t < subscriberList.Count; t++)
102                       {
103                         alreadySubscribed= subscriberList[t].Item1.Equals(client.id);
104                       }
105                     if (alreadySubscribed)
106                     {
107                        Console.WriteLine("Already Subscribed");
108                        RetainedPublish(client, con, messageObj, parentTopic, writer);
109                     }
110                    else
111                     {
112                        //Client wird in subscribe liste aufgenommen
113                        var tupel = new Tuple<string, int>(client.id, qos);
114                        subscriberList.Add(tupel);
115                        var subJson = JsonConvert.SerializeObject(subscriberList);
116                        statement = "UPDATE topics SET Subscribers='" + subJson +
117                                    "' WHERE TopicName='" +
118                                    parentTopic + "';";
119                        var command = new MySqlCommand(statement, con);
120                        command.Prepare();
121                        command.ExecuteNonQuery();
122                        PreviousSingleWildcardUpdater(con,client,parentTopic,
123                        qos,messageObj,writer);
124                        updateSingleWildcardAtEnd(con, client, parentTopic,
125                        qos, messageObj,writer);
126                        UpdateClientSubList(con,client,parentTopic,qos);
127                        RetainedPublish(client, con, messageObj,
128                        parentTopic,writer);
129                    }
130                       if (reader.IsClosed == false)
131                       {
132                            reader.Dispose();
133                       }
134                   }
```

```
135                 }
136                 else
137                 {
138                     //falls topic noch nicht vorhanden, topic wird angelegt
139                     reader.Dispose();
140                     comd.Dispose();
141                     var subscribers = new List<Tuple<string, int>>();
142                     var breaked = false;
143                     for (var l = j; l < splittedTopics.Length; l++)
144                     {
145                         var subTopics = new List<string>();
146                         var cmd = new MySqlCommand();
147                         cmd.Connection = con;
148                         if (l != j)
149                         {
150                             topicName = topicName + "/" + splittedTopics[l];
151                         }
152                         if (l == splittedTopics.Length - 1)
153                         {
154                           //Subscribe
155                           var tupel = new Tuple<string, int>(client.id, qos);
156                           subscribers.Add(tupel);
157                           var subscriberJson = JsonConvert.SerializeObject(subscribers);
158                           Console.WriteLine("Topicname" + topicName);
159                           Console.WriteLine("Laenge: " + topicName.Length);
160                           cmd.CommandText = "INSERT INTO topics
161                           (TopicName,Subscribers,Parent)
162                           VALUES('" +topicName + "','" + subscriberJson + "',
163                           '" + parentTopic + "');";
164                           cmd.Prepare();
165                           cmd.ExecuteNonQuery();
166                           cmd.Dispose();
167                           UpdateClientSubList(con,client,topicName,qos);
168                           PreviousSingleWildcardUpdater(con,client,topicName,qos,
169                           messageObj,writer);
170                           updateSingleWildcardAtEnd(con, client, topicName,
171                           qos, messageObj,writer);
172                           RetainedPublish(client, con, messageObj, topicName,writer);
173                          //subtopics gegebenenfalls aktualisieren
174                           var subTopicSplit = topicName.Split('/');
175                           var sub = "";
176                           var topic = "";
```

```
177                         for(var subCount = 0;subCount < subTopicSplit.Length;subCount++)
178                         {
179                              if (subCount == subTopicSplit.Length - 1)
180                             {
181                              sub = subTopicSplit[subCount];
182                             }
183                             else
184                             {
185                                 if (topic.Contains("Root"))
186                                 {
187                                    topic = topic + "/" + subTopicSplit[subCount];
188                                 }
189                                 else
190                                 {
191                                     topic = topic + subTopicSplit[subCount];
192                                 }
193                             }
194                         }
195                         if (sub.Equals("#"))
196                          {
197                             WildcardHelper(con, topic, client, qos,messageObj,writer);
198                          }
199                          else
200                          {
201                             var subList = new List<string>();
202                             var statement = "SELECT SubTopics FROM topics
203                             WHERE TopicName='" + topic + "';";
204                             var commd = new MySqlCommand(statement, con);
205                             var subRead = commd.ExecuteReader();
206                             if (subRead.HasRows)
207                             {
208                                 while (subRead.Read())
209                                 {
210                                     try
211                                     {
212                                         subList =JsonConvert.DeserializeObject
213                                             <List<string>>(subRead.GetString(0));
214                                     }
215                                     catch(Exception)
216                                     {
217                                     }
218                                 }
```

```
219                     }
220                     subRead.Dispose();
221                     commd.Dispose();
222                     if (subList.Contains(sub))
223                      {
224                         WildcardUpdater(con, topic, client,
225                         qos,messageObj,writer);
226                      }
227                       else
228                      {
229                        subList.Add(sub);
230                        var subListJson = JsonConvert.SerializeObject(subList);
231                        var subcmd = new MySqlCommand(statement, con);
232                        subcmd.CommandText =
233                                      "UPDATE topics SET SubTopics='"
234                                      + subListJson + "' WHERE TopicName='"
235                                      + topic + "';";
236                         subcmd.Prepare();
237                         subcmd.ExecuteNonQuery();
238                         subcmd.Dispose();
239                         WildcardUpdater(con, topic, client,
240                         qos,messageObj,writer);
241                      }
242                  }
243                  updateSingleWildcardAtEnd(con, client, topicName,
244                  qos, messageObj,writer);
245               }
246               else
247               {
248                 //falls Subtopics vorhanden, werden diese ergaenzt
249                 subTopics.Add(splittedTopics[l + 1]);
250                 var subtopicJson = JsonConvert.SerializeObject(subTopics);
251                 cmd.CommandText = "INSERT INTO topics
252                 (TopicName,SubTopics,Parent)
253                 VALUES('" + topicName + "','" + subtopicJson + "','"
254                 + parentTopic + "');";
255                 cmd.Prepare();
256                 cmd.ExecuteNonQuery();
257                 cmd.Dispose();
258                 PreviousSingleWildcardUpdater(con, client,
259                 topicName, qos,messageObj,writer);
260                 var statement = "SELECT SubTopics FROM topics
```

```
261                         WHERE TopicName='" + parentTopic + "';";
262                         var commd = new MySqlCommand(statement, con);
263                         var subRead = commd.ExecuteReader();
264                         var subList = new List<string>();
265                         if (subRead.HasRows)
266                         {
267                           while (subRead.Read())
268                           {
269                               try
270                               {
271                                   subList =JsonConvert.DeserializeObject
272                                       <List<string>>(subRead.GetString(0));
273                               }
274                               catch(Exception)
275                               {
276                               }
277                           }
278                           commd.Dispose();
279                           subRead.Dispose();
280                           var subTopicSplit = topicName.Split('/');
281                           var sub = "";
282                           for(var subCount= 0;subCount<subTopicSplit.Length;subCount++)
283                           {
284                               if (subCount == subTopicSplit.Length - 1)
285                               {
286                                   sub = subTopicSplit[subCount];
287                               }
288                           }
289                           if (subList.Contains(sub) == false)
290                           {
291                               subList.Add(sub);
292                           }
293                           var subListJson = JsonConvert.SerializeObject(subList);
294                           var subcmd = new MySqlCommand(statement, con);
295                           subcmd.CommandText =
296                               "UPDATE topics SET SubTopics='"
297                               + subListJson + "' WHERE TopicName='" +
298                                parentTopic + "';";
299                           subcmd.Prepare();
300                           subcmd.ExecuteNonQuery();
301                           subcmd.Dispose();
302                           WildcardUpdater(con, parentTopic, client,
```

```
                        qos,messageObj,writer);
                    }
                    if (subRead.IsClosed == false)
                    {
                        subRead.Dispose();
                    }
                    if (j == splittedTopics.Length - 1)
                    {
                        parentTopic = "Root";
                    }
                    else
                    {
                        parentTopic = parentTopic + "/" + splittedTopics[l];
                    }
                    breaked = true;
                    break;
                }
            }
            if (!breaked)
            {
                updateSingleWildcardAtEnd(con, client, topicName,
                qos,messageObj,writer);
            }
            PreviousSingleWildcardUpdater(con, client, topicName,
            qos,messageObj,writer);
        }
    }
}
return true;
}
catch (Exception e)
{
    Console.WriteLine(e.StackTrace);
    return false;
}
}
```

Listing A.5: Durchführung eines Subscribes

```
public byte[] Puback(byte[] id)
    {
        var puback = new byte[4];
        const byte bit0 = (byte)(1 << 6);
```

```
            const byte bit1 = (byte)(1 << 1);
            puback[0] |= bit0;
            puback[1] |= bit1;
            puback[3] = id[0];
            return puback;
        }
```

Listing A.6: Erzeugen einer Nachricht, Beispiel PUBACK

```
while (!pubackReceived&&tcpClient.Connected&&publishCounter<5)
      {
       try {
            System.Threading.Thread.Sleep(5000);
            stm = "SELECT QoS1 FROM clients WHERE ClientID='" + client.id + "';";
            var commd = new MySqlCommand(stm, con);
            var dataReader = commd.ExecuteReader();
            if (dataReader.HasRows)
            {
               try
               {
                    while (dataReader.Read())
                    {
                        qosList = JsonConvert.DeserializeObject
                        <List<Tuple<string, string, int, int>>>(dataReader.GetString(0));
                    }
                    dataReader.Dispose();
                    if (qosList.Count == 0)
                    {
                      client.packetIdList.Remove(packetId);
                      pubackReceived = true;
                    }
                    for (var counter = 0; counter < qosList.Count; counter++)
                    {
                       if (qosList[counter].Item1.Equals(contentString)
                       && qosList[counter].Item2.Equals(topicString))
                       {
                          var tempQoSList = new List<Tuple<string, string, int, int>>();
                          foreach (Tuple<string, string, int, int> tuple in qosList)
                          {
                            if (tuple.Item4 != packetId)
                            {
                              tempQoSList.Add(tuple);
                            }
```

```
                }
                if (tcpClient.Connected)
                {
                  var updaetTuple = new Tuple<string, string, int, int>
                  (qosList[counter].Item1,qosList[counter].Item2,
                  qosList[counter].Item3,packetId);
                  tempQoSList.Add(updaetTuple);
                  var qosListJson = JsonConvert.SerializeObject(tempQoSList);
                  var qoscmd = new MySqlCommand(stm, con);
                  qoscmd.CommandText = "UPDATE clients SET QoS1='"
                  + qosListJson + "' WHERE ClientID='"
                  + client.id + "';";
                  qoscmd.Prepare();
                  qoscmd.ExecuteNonQuery();
                  qoscmd.Dispose();
                  writer.Write(messageObj.SendPublish
                  (content, topic, 1, packetId,true), 0, message.Length);
                  writer.Flush();
                  publishCounter++;
               }
               else
               {
                 client.packetIdList.Remove(packetId);
                 con.Close();
                 client.qosrecieved = true;
                 return true; ;
               }
          }
          else if (counter == qosList.Count - 1)
               {
                   pubackReceived = true;
                   client.packetIdList.Remove(packetId);
                   break;
                }
          }
     }
     catch (Exception) {
       client.packetIdList.Remove(packetId);
       pubackReceived = true;                        }
       }
       else
       {
```

```
77                  dataReader.Dispose();
78                  commd.Dispose();
79                  client.packetIdList.Remove(packetId);
80                  pubackReceived = true;
81                  break;
82                  }
83              }
84              catch (Exception) {
85                  client.packetIdList.Remove(packetId);
86                  pubackReceived = true;
87              };
88      }
```

Listing A.7: Senden einer Nachricht mit QoS 1

```
1    public void Send (byte[] message, TcpClient tcpClient, BinaryWriter writer)
2         {
3             writer.Write(message, 0, message.Length);
4             writer.Flush();
5         }
```

Listing A.8: Senden einer Nachricht

```
1  public void Error(BinaryReader reader, BinaryWriter writer, TcpClient tcpClient,
2  string message, MySqlConnection con,
3  Client client,Message messageObj)
4      {
5         if (message.Equals("Keep Alive TimeOut"))
6         {
7             if (client.willFlag == 1)
8             {
9                 Console.WriteLine(client.willTopic);
10                ASCIIEncoding enc = new ASCIIEncoding();
11                var content = enc.GetBytes(client.willMessage);
12                var topic = enc.GetBytes(client.willTopic);
13                Publish(content, topic, con, messageObj, writer, false,client.willQoS);
14                reader.Dispose();
15                writer.Dispose();
16                if (client.willRetain==1)
17                {
18                    var command = new MySqlCommand();
19                    command.Connection = con;
20                    var updateTuple = new Tuple<string, int>
```

```
21                  (client.willMessage,client.willQoS);
22                  var updateJson = JsonConvert.SerializeObject(updateTuple);
23                  command.CommandText = "UPDATE topics SET RetainedMessage='"
24                  +updateJson+"' WHERE TopicName='Root/"
25                  +client.willTopic+"';";
26                  command.Prepare();
27                  command.ExecuteNonQuery();
28                  command.Dispose();
29              }
30          }
31          if (client.cleanSession == 1)
32          {
33              UnsubscribeAll(con, client);
34              var cmd = new MySqlCommand();
35              cmd.Connection = con;
36              //client loeschen
37              cmd.CommandText = "DELETE FROM clients WHERE ClientID = '"
38              + client.id + "';";
39              cmd.Prepare();
40              cmd.ExecuteNonQuery();
41              dictionary.Remove(client.id);
42          }
43          try
44          {
45              tcpClient.GetStream().Dispose();
46              tcpClient.Dispose();
47          }
48          catch (Exception) { }
49          reader.Dispose();
50          writer.Dispose();
51          con.Close();
52          Console.WriteLine(message);
53      }
54      else
55      {
56          if (client.willFlag == 1)
57          {
58              Console.WriteLine(client.willTopic);
59              ASCIIEncoding enc = new ASCIIEncoding();
60              var content = enc.GetBytes(client.willMessage);
61              var topic = enc.GetBytes(client.willTopic);
62              Publish(content, topic, con, messageObj, writer,
```

```
63                  false,client.willQoS);
64                  reader.Dispose();
65                  writer.Dispose();
66              }
67              if (client.cleanSession == 1)
68              {
69                  UnsubscribeAll(con, client);
70                  var cmd = new MySqlCommand();
71                  cmd.Connection = con;
72                  //client loeschen
73                  cmd.CommandText = "DELETE FROM clients WHERE ClientID = '"
74                  + client.id + "';";
75                  cmd.Prepare();
76                  cmd.ExecuteNonQuery();
77                  dictionary.Remove(client.id);
78              }
79              try
80              {
81                  tcpClient.GetStream().Dispose();
82                  reader.Dispose();
83                  writer.Dispose();
84                  Console.WriteLine(message);
85              }
86              catch (Exception)
87              {
88                  reader.Dispose();
89                  writer.Dispose();
90                  Console.WriteLine(message);
91              }
92              try
93              {
94                  tcpClient.GetStream().Dispose();
95                  tcpClient.Dispose();
96              }
97              catch (Exception) { }
98              con.Close();
99      }
100 }
```

Listing A.9: Beenden einer Verbindung

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Lukas Schulz Matrikelnummer: 830589

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Lukas Schulz