Smart Load Balancing MQTT Traffic an HiveMQ Cluster mit Envoy

Abschlussarbeit zur Erlangung des Bachelor of Science
in der Fachrichtung Technischer Informatik
mit dem Schwerpunkt Software Systeme

Author:	Felix Breuer
Matrikelnummer:	111 217 51
Adresse:	Mutzer Straße 72
	51467 Bergisch Gladbach
	felix.breuer@smail.th-koeln.de
Erstgutachter:	Prof. Dr. René Wörzberger
Zweitgutachter:	Dipl.-Inform. (TH) Christian Rohmann
Abgabefrist:	07.05.2021

Zusammenfassung

Das *Internet of Things* revolutioniert die Interaktion zwischen Mensch und Umwelt. Dabei werden physische Objekte mit Sensoren und Mikrochips ausgestattet, um Daten an dedizierte IoT-Plattformen zu senden. Um einen effizienten Nachrichtenaustausch zu ermöglichen, wurde das Message Queuing Telemetry Transport (MQTT) Protokoll entworfen. Dieses verwendet das *Publish und Subscribe* Kommunikationsschema, das Nachrichten zwischen Clients über einen Broker austauscht. Bei dem clusterfähigen MQTT Broker von HiveMQ benötigt man einen Load-Balancer, um die einzelnen Nodes für die MQTT Clients zu abstrahieren. Envoy ist ein OSI-Layer sieben Proxy, der für große, moderne und serviceorientierte Architekturen entwickelt wurde. Durch Features, wie das Einbinden von WASM-Modulen und das dynamische Konfigurieren via Control-Plane, ist Envoy flexibel einsetz- und erweiterbar. Damit Millionen von MQTT Clients Nachrichten über eine IoT-Plattform austauschen können, wird in der vorliegenden Arbeit untersucht, wie man MQTT Clientverbindungen *klug* über einen Envoy Proxy an Nodes eines HiveMQ Clusters verteilen kann.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Zielsetzung

Das Internet der Dinge (engl. *Internet of Things (IoT)*) verändert die Art der Interaktion von Menschen mit ihrer Umwelt. Physische Objekte werden mit Sensoren und Mikrochips ausgestattet, um Daten an dedizierte IoT-Plattformen zu übermitteln. Diese kategorisieren, filtern und werten die Daten aus. IoT verschafft ein besseres Verständnis von Prozessen und deren Interaktion zueinander.

Durch das Internet of Things können beispielsweise Menschen ihre Gesundheit mit Armbändern beobachten und frühzeitig bewusst auf Veränderungen ihres Körpers reagieren. Bauteile von Maschinen werden mit Sensoren ausgestattet, um kontinuierlich die Funktionalität zu überprüfen. Bei Anzeichen, die auf einen möglichen Ausfall des Bauteils hinweisen, werden automatisch Techniker:innen für eine präventive Wartung angefordert.

Aufgrund des stetigen Ausbaus mobiler Netzwerke steigt die Anzahl potenzieller IoT-Geräte nahezu täglich. Damit die Kommunikation der Geräte untereinander sowie mit den datenverarbeitenden Services reibungslos stattfindet, müssen die IoT-Plattformen empfangene Daten möglichst in Echtzeit verarbeiten. Um einen effizienten Nachrichtenaustausch zu ermöglichen, wurde das MQTT Protokoll entworfen. Es ist eventgesteuert und verwendet das *Publish und Subscribe* Kommunikationsschema. Bei diesem Protokoll können IoT-Geräte und Services, sogenannte Clients, Nachrichten über einen MQTT Broker austauschen. Damit ein Broker auch bei mehreren Millionen Geräten alle Nachrichten schnell und ausfallsicher verarbeiten kann, formen Broker, wie zum Beispiel der HiveMQ Broker, ein Cluster. Dabei bilden mehrere HiveMQ Broker ein HiveMQ Cluster, das eine logische Einheit für Clients bildet.

Load-Balancer abstrahieren die einzelnen Nodes eines Clusters für Clients. Dabei wird den Nodes ein Load-Balancer vorgeschaltet, der eingehende Clientverbindungen empfängt, um diese basierend auf unterschiedlichen load balancing Algorithmen an die Nodes weiterzuleiten. Bei der Wahl des Zielnodes für einen bestimmten Client kann der Load-Balancer diverse Merkmale in Betracht ziehen und somit beispielsweise die Servicequalität für Clients oder die gleichmäßige Verteilung der Last auf alle Nodes des Clusters begünstigen.

Das Hypertext-Transfer-Protocol (HTTP) wird verwendet, sobald ein Webbrowser eine Internetseite aufruft. Aufgrund der Popularität des Protokolls können viele Load-Balancer HTTP optimierte load balancing Entscheidungen treffen. Für das MQTT Protokoll gibt es trotz des IoT Booms noch keine protokollbezogenen Optimierungen. In der vorliegenden Arbeit wird untersucht, wie ein Load-Balancer *kluge* load balancing Entscheidungen für eingehende MQTT Clientverbindungen an HiveMQ Nodes treffen kann. Bei der stetig steigenden Anzahl von IoT-Geräten muss der Load-Balancer reaktiv mit neuen Lastsituationen umgehen können, um einen effizienten Nachrichtenaustausch zu ermöglichen.

1.2 Unternehmen

Die vorliegende Arbeit wird bei der Firma inovex GmbH im Team IT-Engineering & Operations angefertigt. Die inovex GmbH ist ein innovations- und qualitätsgetriebenes IT-Projekthaus mit dem Leistungsschwerpunkt *Digitale Transformation*. Es unterstützt Unternehmen bei der Digitalisierung und Agilisierung ihres Kerngeschäfts sowie bei der Realisierung von neuen digitalen *Use Cases*. Das aktuelle Lösungsangebot umfasst Application Development (Web Platforms, Mobile Apps, Edge & Embedded), Data Management & Analytics (Business Intelligence, Big Data, Search, Data Science, Artificial Intelligence) und die Entwicklung von skalierbaren IT Infrastrukturen (IT Engineering, Cloud Services), auf denen die digitalen Lösungen im DevOps-Modus betrieben werden. Zudem modernisiert inovex vorhandene Lösungen (Replatforming), härtet Systeme gegen Angriffe von außen (Security) und vermittelt ihr Wissen durch Trainings und Workshops (inovex Academy). Im Jahr 2021 beschäftigte die inovex GmbH 395 Mitarbeitende an sieben Standorten: Karslruhe, Köln, München, Hamburg, Stuttgart, Pforzheim und Münster.

Im Team *IT-Engineering & Operations (ITO)* entstehen Cloud-Plattformen, die entweder komplett als private Cloud oder als hybride Cloud im Zusammenspiel mit den großen Public Clouds (Amazon, Google, Microsoft) realisiert werden. Durch die wachsende Nachfrage von IoT-Projekten befasst sich das Team ITO zunehmend mit dem Internet of Things Ökosystem, dessen Infrastruktur sowie mit betrieblichen Aspekten. Durch die im Jahr 2020 entstandene Partnerschaft mit HiveMQ und der Einführung von IoT als strategisches Thema bei inovex, wurde ein Raum für Forschung, Weiterbildung und die Durchführung von Arbeiten in diesem Ökosystem geschaffen.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit beginnt mit einem ausführlichen Grundlagenteil in Kapitel 3. Dabei werden zunächst die Funktionsweise und die Eigenheiten des MQTT Protokolls erläutert. Grundlagen wie das *Publish und Subscribe* Kommunikationsschema aus Kapitel 3.1.1 sind Voraussetzungen für das Verständnis der in Kapitel 3.2.1 beschriebenen Clusterfähigkeit des HiveMQ MQTT Brokers. Die technischen Grundlagen werden komplettiert mit einem Überblick über Load-Balancer. Kapitel 4 beschreibt Problemstellungen beim Verteilen von MQTT Clients an ein Broker-Cluster durch einen Load-Balancer. Diese werden in Kapitel 5 erläutert und mit Hinblick auf einen dynamischen load balancing Algorithmus analysiert. Dabei wird besonders auf die gleichmäßige Verteilung der Last auf alle Nodes eines HiveMQ Clusters und die Sicherstellung der Servicequalität für die MQTT Clients geachtet. Abschließend wird der Aufbau einer Control-Plane für das *kluge* Verteilen von MQTT Clients in Kapitel 6 dargestellt.

2 Anwendungsfeld Internet of Things

Das Internet hat die Welt der Computer und der Kommunikation revolutioniert. Es ist ein Medium für eine geografisch unabhängige Kollaboration und Interaktion zwischen Mensch und Computer. [1]

Das Internet der Dinge nutzt dieses Medium, um nicht nur Mensch und Computer zu vernetzen, sondern auch Objekte. Es bildet eine Grundlage, um Sensoren, Aktoren, Smartphones, Maschinen, Lampen, Autos oder andere elektronischen Geräte mit dem Internet zu verbinden [2]. Dadurch wird eine Kommunikation zwischen Mensch und Objekt oder Objekt und Objekt ermöglicht [3, S. 2]. Solche Objekte werden auch *Smart-Devices* genannt. Gartner [4] und statista [5] erwarten bis 2025 rund 30 - 75 Milliarden Geräte, die mit dem Internet verbunden sind.

Durch die Vernetzung alltäglicher Objekte wird unsere Umgebung klüger und reaktiv. Die Idee, verschiedene Objekte mit dem Internet zu verbinden, wurde bereits um 1980 und 1990 behandelt, jedoch waren zu dieser Zeit die Mikrocontroller und Mikrochips zu groß und zu langsam, um effektiv eingesetzt werden zu können. Mit der stetigen Optimierung der Chips können immer mehr Objekte mit solchen ausgestattet und somit in die Lage versetzt werden, mit ihrer Umwelt zu kommunizieren. [6]

Die stetige Zunahme von Smart-Devices stellt jedoch aktuelle Kommunikationsprotokolle und Infrastrukturen vor neue Herausforderungen. Diese wurden ursprünglich für statische Geräte mit einer stetigen Strom- und Internetanbindung entwickelt. Millionen Geräte, die unter anderem auch durch ein instabiles mobiles Netzwerk mit dem Internet verbunden sind, erfordern neue Protokolle und Technologien, die ihnen eine optimale Integration in die bestehende Infrastruktur ermöglichen. [3, S. 7f]

3 Technische Grundlagen

Das folgende Kapitel erläutert die erforderlichen technologischen Grundlagen für das tiefere Verständnis des Themas der Thesis. Das MQTT Protokoll wird in Kapitel 3.1 beschrieben. Kapitel 3.2 handelt von den Besonderheiten eines HiveMQ Brokers und geht insbesondere auf die Cluster-Mechanik ein. Abschließend wird die Funktionalität eines Load-Balancers am Beispiel von Envoy in Kapitel 3.3 dargestellt.

3.1 Message Queuing Telemetry Transport

Message Queuing Telemetry Transport (MQTT) wurde ursprünglich von Doktor Andy Stanford-Clark und Arlen Nipper im Jahr 1999 entworfen, um Gas- und Ölpipelines zu überwachen. Durch die oftmals abgelegene Lage, wie zum Beispiel auf Übersee, konnten sie nicht mit Radiowellen oder einem Kabel vom Festland erreicht werden. Seinerzeit war die einzige Option, um Sensordaten auf einen Server zu übertragen, eine auf Datendurchsatz abgerechnete Satellitenkommunikation. Bei mehreren tausend Sensoren wurde ein Protokoll benötigt, das die Daten zuverlässig mit minimaler Bandbreite an die Server auf dem Festland übermittelt. MQTT wurde im Jahr 2013 von der Organization for the Advancement of Structured Information Standards (OASIS) als offener Standard veröffentlicht und wird heutzutage von vielen großen IoT-Plattformen wie AWS IoT und Sparkplug unterstützt. [7]

Es gibt derzeit zwei verfügbare Versionen der MQTT Spezifikation: 3.1.1 und 5. Alle Referenzen, falls nicht explizit gekennzeichnet, beziehen sich auf die aktuelle Version 5 des Protokolls.

MQTT ist ein OSI-Layer sieben *Publish und Subscribe* Protokoll, das auf TCP / IP aufsetzt. Anders als das Request / Response Paradigma bei HTTP ist MQTT eventgesteuert und erlaubt Servern, Nachrichten direkt an Clients zu schicken. Somit müssen Clients neue Nachrichten nicht periodisch abfragen. Zusammen mit einem fixen Paket-Header von nur zwei Byte sorgen diese Eigenschaften für einen geringen Overhead im Netzwerk.[7]

3.1.1 Publish und Subscribe

MQTT nutzt das *Publish und Subscribe* Kommunikationsschema, das eine Struktur bietet, um Nachrichten entkoppelt zwischen Clients auszutauschen. In diesem Schema gibt es zwei unterschiedliche Systeme:

- Clients
- Broker

Clients sind alle Teilnehmer dieses Systems, die Nachrichten empfangen (*Subscriber*) oder veröffentlichen (*Publisher*). Um Nachrichten erfolgreich zu übermitteln, wird ein Broker als Mittelsmann eingesetzt [8]. Jede Nachricht wird durch den Publisher in bestimmte Klassen kategorisiert. Der Publisher weiß nicht, ob oder wer an dieser Nachricht interessiert ist und schickt die klassifizierte Nachricht an den Broker. Clients, die an bestimmten

Nachrichten interessiert sind, müssen bei dem Broker eine Subscription erstellen und eine Kategorie angeben. So kann der Broker eingehende kategorisierte Nachrichten an die entsprechenden Clients weiterleiten. Die Kategorie funktioniert wie ein Filter auf allen Nachrichten. Der Broker kann Nachrichten auch aufbewahren, sodass Clients, die zu einem späteren Zeitpunkt eine Kategorie abonnieren, ebenfalls die vergangenen Nachrichten erhalten [9, S. 295]. Durch dieses System entsteht eine Entkopplung der einzelnen Clients auf mehreren Ebenen [10].

- **Kommunikative Entkopplung:** Publisher und Subscriber der Nachricht müssen sich nicht kennen.
- **Zeitliche Entkopplung:** Publisher und Subscriber müssen nicht zur selben Zeit aktiv sein.
- **Synchronisierende Entkopplung:** Publisher und Subscriber müssen ihre Operationen beim Publizieren und Konsumieren nicht unterbrechen.

In verteilten Systemen bietet ein solches Konzept eine einheitliche Abstraktion der Kommunikationsebene.

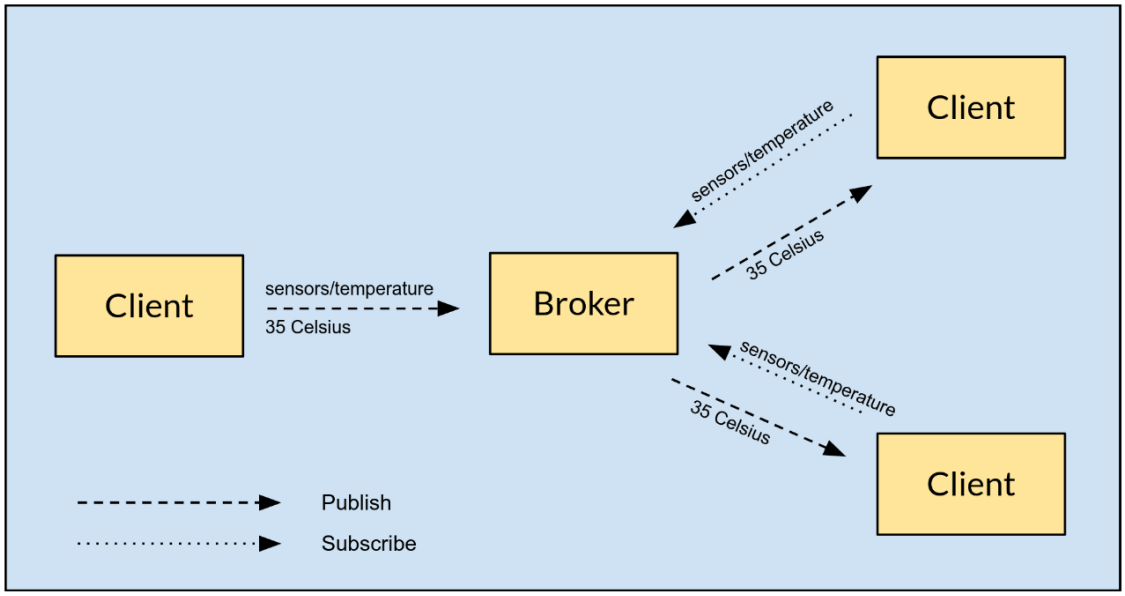

Abbildung 1: Es wird eine Publish und Subscribe Architektur dargestellt. Nachrichten werden über einen Broker auf dem Topic *sensors/temperature* ausgetauscht.

Im Kontext MQTT werden Nachrichten in ein hierarchisch aufgebautes Topic klassifiziert, das mit einem oder mehreren *Topic-levels* adressiert wird. Bei der Notation werden Topic-levels mit einem Schrägstrich (`/`) getrennt: `sensors/temperature/celcius`. Abbildung 1 zeigt einen Client, der auf das Topic `sensors/temperature` Nachrichten veröffentlicht. Diese wird an zwei weitere Clients, die das Topic `sensors/temperature` abonniert haben, durch den Broker weitergeleitet. Topics müssen bei einem Broker nicht explizit angelegt werden. Sobald ein Client auf einem Topic publiziert oder es abonniert, wird dieses automatisch angelegt.[7]

Beim Abonnieren können Clients ein exaktes Topic angeben oder mehrere Topics gleichzeitig abonnieren, indem sie Wildcards als Topic-levels verwenden. Bei Wildcards werden zwischen den folgenden zwei unterschieden [11]:

- Single-level '`+`': Ersetzt ein Topic-level und abonniert alle Topics, die an der Stelle der Wildcard ein beliebiges Topic-level verwenden.
- Multi-level '`#`': Muss als letztes Topic-level verwendet werden und abonniert alle Topics, die an der Stelle der Wildcard und allen nachfolgenden Stellen ein beliebiges Topic-level verwenden.

Bei der folgenden Topic-Struktur

- `sensors`
- `sensors/temperature`
- `sensors/temperature/celcius`
- `sensors/temperature/kelvin`
- `sensors/fuel`
- `sensors/fuel/tank1`
- `sensors/fuel/tank2`

schließt eine Subscription auf `sensors/#` alle Topics mit ein. Bei einem Abonnement auf `sensors/+` sind hingegen nur diese Topics mit eingeschlossen:

- `sensors/temperature`
- `sensors/fuel`

3.1.2 Quality-of-Service

Je zuverlässiger eine Nachricht übermittelt werden soll, desto mehr Kommunikationsaufwand verursacht sie im gesamten System. Wenn ein Client wissen möchte, ob seine Nachricht im Broker eingetroffen ist, muss der Broker dem Client eine entsprechende Rückmeldung geben. Andernfalls kann der Client seine Nachricht an den Broker schicken ohne eine Rückmeldung zu erwarten.

Bei MQTT wird die Zuverlässigkeit der Übermittlung einer Nachricht mit einem *Quality-of-Service (QoS)* Level festgelegt. Eine publizierte Nachricht muss einem der drei QoS Level zugeordnet sein:

- QoS 0: Maximal eine Zustellung der Nachricht.
- QoS 1: Mindestens eine Zustellung der Nachricht.
- QoS 2: Genau eine Zustellung der Nachricht.

Bei QoS Level 1 und 2 werden Handshakes zur Verifizierung der Nachrichtenübermittlung eingesetzt.[12]

3.1.3 Retained-Message

In MQTT kann pro Topic die zuletzt veröffentlichte Nachricht aufbewahrt werden. Dazu muss im `PUBLISH` Paket die `RETAIN` Kennzeichnung gesetzt werden (siehe Kapitel 3.1.4). In diesem Fall speichert der Broker die Nachricht mit dem dazugehörigen Topic und QoS Level ab. Falls bereits eine aufbewahrte Nachricht (engl. *Retained-Message*) auf diesem Topic vorhanden ist, wird sie überschrieben. Jeder Client, der ein Topic abonniert das eine Retained-Message besitzt, bekommt diese Nachricht direkt nach dem Abonnieren zugeschickt. Dies trifft auch zu, falls das Topic mit der Retained-Message durch einen Wildcard Filter abonniert wurde. [13]

Um eine Retained-Message zu löschen, muss eine neue Nachricht mit der `RETAIN` Kennzeichnung und einem leeren Payload auf dem Topic veröffentlicht werden. Der Broker löscht die aktuelle Retained-Message und neue Subscriber werden erst Nachrichten von diesem Topic erhalten, sobald eine neue Nachricht dort veröffentlicht wird. Wenn ein Topic eine Retained-Message (im Folgenden *M1* genannt) aufweist und eine neue Nachricht ohne `RETAIN` Kennzeichnung auf diesem Topic veröffentlicht wird, wird die aktuelle Retained-Message M1 nicht überschrieben. Neue Subscriber erhalten weiterhin die Nachricht *M1*. [12]

Durch Retained-Messages können neue Subscriber direkt einen Wert erhalten, nachdem sie ein Topic abonniert haben. Daher wird eine Retained-Message auch *last known good value* genannt. [13]

3.1.4 Paket Struktur

MQTT hat die Absicht, ein leichtgewichtiges Protokoll zu sein. Tabelle 1 zeigt den Aufbau eines MQTT Paketes. Der fixe Header ist zwei Byte groß und muss in jedem Paket vorhanden sein. Basierend auf der Art des Paketes, das in dem fixen Header angegeben ist, sind zusätzlich ein variabler Header und weitere Daten möglich [12]. Tabelle 2 zeigt den

Fixer Header, muss in jedem MQTT Paket vorhanden sein
Variabler Header, optional
Daten des Pakets, optional

Tabelle 1: Zeigt die Struktur eines MQTT Paketes. Quelle: [12]

detaillierten Aufbau des fixen Headers. Im ersten Byte werden Bit sieben bis vier für die spezifische Art des Paketes verwendet. Tabelle 3 enthält eine Liste mit allen Pakettypen und deren Wert. Byte zwei gibt die restliche Paketlänge enkodiert als *Variable-Byte-Integer* an [12]. Die Enkodierung eines *Variable-Byte-Integer* wird in Kapitel 1.5.5 der MQTT 5 Spezifikation [12] detailliert erklärt.

Bit	7	6	5	4	3	2	1	0
Byte 1	MQTT Paketart				Paketart spezifische Kennzeichnung			
Byte 2	Restliche Paketlänge							

Tabelle 2: Zeigt den Aufbau und die Belegung der Bits des fixen MQTT Headers. Quelle: [12]

Name	**Wert**
Reserved	0
CONNECT	1
CONNACK	2
PUBLISH	3
PUBACK	4
PUBREC	5
PUBREL	6
PUBCOMP	7
SUBSCRIBE	8
SUBACK	9
UNSUBSCRIBE	10
UNSUBACK	11
PINGREQ	12
PINGRESP	13
DISCONNECT	14
AUTH	15

Tabelle 3: Listet alle verfügbaren MQTT Paketarten auf. Der Wert ist die Identifikation der Paketart im fixen Header. Quelle: [12]

3.1.5 MQTT CONNECT

Nachdem eine Transmission-Control-Protocol (TCP) Verbindung zwischen einem Client und einem Broker aufgebaut wurde, muss das erste Paket des Clients ein `CONNECT` Paket sein. Tabelle 4 zeigt den Aufbau und Inhalt eines beispielhaften variablen Headers eines `CONNECT` Paketes. Die ersten sechs Byte geben den Namen des Protokolls als UTF-8 enkodierten String an. Dieser wird sich in zukünftigen Versionen nicht ändern und dient als Identifikationsmerkmal des MQTT Protokolls für Server, die mehrere Protokolle implementiert haben. Byte sieben gibt die verwendete MQTT Version des Clients an. In Byte acht kann der Client Kennzeichnungen für die Präsenz der optionalen Daten setzen. Die Länge der Eigenschaften wird mit Byte elf bestimmt. In den Eigenschaften kann der Client zum Beispiel eine maximale Paketgröße oder Topic Aliasse setzen.[12]

Auf den variablen Header folgen die Daten (engl. *Payload*) des `CONNECT` Paketes. Diese beinhalten ein oder mehrere Felder mit ihrer Länge als Präfix. Byte acht des variablen Headers bestimmt das Vorkommen der Felder, die - falls vorhanden - eine strikte Reihenfolge einhalten müssen. [12]

1. Clientkennung
2. *Will* Eigenschaften
3. *Will* Topic
4. *Will* Payload
5. Benutzername
6. Passwort

Die Clientkennung ist als einziges Feld nicht optional. Sie muss ein UTF-8 enkodierter String sein und aus folgenden Zeichen bestehen: 0-9, a-z, A-Z. Eine Clientkennung dient zur Identifikation des aktuellen Zustands des Clients und muss daher einzigartig vergeben sein. Ist bereits ein Client mit der gleichen Kennung mit dem Broker verbunden, wird die Verbindung des existierenden Clients getrennt und es findet ein *Client-Takeover* statt (siehe 3.2.2.1). Der Broker muss eine `CONNECT` Nachricht von einem Client mit einem `CONNACK` Paket beantworten. [12]

	Beschreibung	**Wert**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Byte 1	Länge MSB	0	0	0	0	0	0	0	0	0
Byte 2	Länge LSB	4	0	0	0	0	0	1	0	0
Byte 3	UTF-8 Char	'M'	0	1	0	0	1	1	0	1
Byte 4	UTF-8 Char	'Q'	0	1	0	1	0	0	0	1
Byte 5	UTF-8 Char	'T'	0	1	0	1	0	1	0	0
Byte 6	UTF-8 Char	'T'	0	1	0	1	0	1	0	0
Byte 7	Protokoll Version	5	0	0	0	0	0	1	0	1
Byte 8	Benutzername Kennzeichnung Passwort Kennzeichnung *Will* speichern *Will* QoS Level *Will* Kennzeichnung Frische Session Reserviert	1 1 0 01 1 1 0	1	1	0	0	1	1	1	0
Byte 9	Keep Alive MSB	0	0	0	0	0	0	0	0	0
Byte 10	Keep Alive LSB	10	0	0	0	0	1	0	1	0
Byte 11	Länge der Eigenschaften	0	0	0	0	0	0	0	0	0

Tabelle 4: Zeigt ein beispielhaftes MQTT CONNECT Paket. Quelle: [12]

3.1.6 Last-Will Nachricht

Ein Client kann bei der Verbindung mit einem Broker einen letzten Willen (engl. *Last-Will*) als Nachricht definieren. Diese Nachricht wird veröffentlicht, sobald die Verbindung zum Client unterbricht und dieser nicht innerhalb eines definierten Zeitraums seine Verbindung zum Broker wiederaufbauen kann. Im Fall einer protokollkonformen Terminierung der Verbindung wird die Last-Will Nachricht nicht veröffentlicht. [14]

Die Last-Will Nachricht kann in sicherheitsrelevanten Applikationen eingesetzt werden, um auf ungeplante Ausfälle von Clients zu reagieren.

3.1.7 Shared-Subscriptions

Mit der fünften Version von MQTT wurden *Shared-Subscriptions* eingeführt, die das Teilen von Subscriptions unter Clients erlauben. Diese Clients werden in einer *Shared-Subscription-Group* zusammengefasst. Bei einer normalen Subscription erhält jeder Client, der das gleiche Topic abonniert hat, eine Kopie der Nachricht. Bei einer Shared-Subscription erhält immer nur ein Client der Shared-Subscription-Group die Nachricht in abwechselnder Reihenfolge. Dieser Mechanismus wird auch als *client load balancing* bezeichnet, da die Last eines Topics auf mehrere Subscriber verteilt wird [15]. Shared-Subscriptions verwenden Standard MQTT Mechanismen. Jeder Client kann Teil einer Shared-Subscription-Group werden, indem er folgende Topic Struktur einhält: `$share/GROUPID/TOPIC`.

- `$share`: Statische Shared-Subscription Kennung
- `GROUPID`: Identifikation einer Shared-Subscription-Group. Darf `/`, `#` und `+` nicht enthalten
- `TOPIC`: Das zu abonnierende Topic

Ein konkretes Beispiel für das Abonnieren einer Shared-Subscription ist:
`$share/group1/sensors/temperature`

3.2 HiveMQ Broker

Der HiveMQ Broker ist ein Produkt der in 2012 gegründeten Firma HiveMQ GmbH. Mit über 130 internationalen Kunden und einem Sitz in Landshut, Deutschland, erleichtert HiveMQ durch ihren MQTT Broker die Kommunikation von Geräten und Services untereinander. [16]

Der HiveMQ Broker ist ein auf Java basierter MQTT Broker, der auf das Anwendungsfeld Internet of Things spezialisiert ist. Der Broker unterstützt MQTT Version 3 und 5 und ist Eclipse Sparkplug kompatibel. Eclipse Sparkplug ist eine Open-Source Spezifikation, die ein Framework für MQTT Anwendungen bietet, indem sie Definitionen für MQTT Topics und MQTT Payload Strukturen definiert. Die Kompatibilität des HiveMQ Brokers mit Sparkplug und die Clusterfähigkeit (siehe Kapitel 3.2.1) ermöglichen den Einsatz des Brokers in Internet of Things Projekten mit mehreren Millionen Clients. [17]

Neben dem Anwendungsfeld IoT kann der HiveMQ Broker auch für *Service-to-Service* Kommunikation im Backend eingesetzt werden. Eine Alternative für eventgesteuerte Kommunikation zwischen Services bieten Open-Source Broker wie Apache Kafka, RabbitMQ oder ZeroMQ.

Bis 2019 war der Quellcode des HiveMQ Brokers nicht frei zugänglich. Im April 2019 wurde eine Community Edition des HiveMQ Brokers unter der Apache 2.0 Lizenz auf GitHub veröffentlicht. Für die Enterprise Edition mit Features wie Clustering und Control-Center ist eine HiveMQ Lizenz notwendig.

3.2.1 HiveMQ Cluster

Ein Merkmal des HiveMQ Brokers ist die Möglichkeit, ein Broker-Cluster zu bilden. Ein Broker-Cluster (folgend Cluster genannt) ist ein verteiltes System, in dem sich mehrere HiveMQ Broker zusammenschließen und eine logische Einheit für Clients bilden. Demnach macht es für Clients keinen Unterschied, ob sie mit einem einzigen HiveMQ Broker oder mit einem Cluster verbunden sind. Um dies zu erreichen, tauschen alle Nodes eines Clusters Informationen über Clients und eingehende Nachrichten aus. Das geschieht über eine dedizierte Verbindung, die auch Cluster-Transport genannt wird. [18]

Durch das Zusammenschließen der HiveMQ Broker entsteht eine horizontale Skalierung. Wie in Kapitel 2 bereits erläutert, können sich im Anwendungsfall IoT mehrere Millionen Clients mit einem MQTT Broker verbinden. Hierbei kann eine vertikale Skalierung an ihre Grenzen stoßen. [19]

- **Vertikale Skalierung:** Fügt mehr Rechen- oder Speichereinheiten zu einem Server hinzu.
- **Horizontale Skalierung:** Fügt mehrere Server zu einer logischen Recheneinheit hinzu. Dabei kann die Arbeitslast auf die einzelnen Server verteilt werden.

Ein HiveMQ Broker in einem HiveMQ Cluster wird als ein HiveMQ Node (im Folgenden Node genannt) bezeichnet. Damit sich mehrere Nodes zu einem Cluster zusammenschließen, muss jeder Node:

- ein HiveMQ Enterprise Broker sein.
- die anderen Nodes über ein Netzwerk erreichen können.
- das *Clustering* in seiner Konfiguration eingeschaltet haben.
- die anderen Nodes mit der *Cluster-Discovery* auffinden können.

Die Cluster-Discovery beschreibt den Mechanismus, wie sich individuelle Nodes gegenseitig finden können, um ein Cluster zu bilden. Es stehen folgende Möglichkeiten zur Verfügung:

- **static:** Jeder Node wird mit einer statischen Liste von allen Nodes und deren IP-Adresse / Port konfiguriert.
- **multicast:** Nodes finden dynamisch andere Nodes, die dieselbe multicast Adresse und Port benutzen.
- **broadcast:** Nodes finden dynamisch andere Nodes im selben Subnetzwerk, indem sie die Broadcast IP-Adresse benutzen.
- **extension:** Es kann eine HiveMQ Erweiterung genutzt werden, um andere Nodes zu entdecken.

HiveMQ ist in der Lage, die Clustergröße zur Laufzeit anzupassen. Diesen Mechanismus muss der gewählte Cluster-Discovery Modus unterstützen. [18]

3.2.1.1 Überlastschutz

HiveMQ stellt einen eingebauten Überlastschutz (engl. *Overload-Protection*) für ein HiveMQ Cluster bereit. Ein HiveMQ Node befindet sich in einer Überlastsituation, wenn in einem bestimmten Zeitraum mehr neue Aufgaben entstehen, als dieser abarbeiten kann. Aufgaben entstehen beispielsweise durch neue MQTT Clients, ein- und ausgehende Nachrichten oder die Vergrößerung des Clusters. Eine Ursache, warum ein Node anstehende Aufgaben nicht rechtzeitig abarbeiten kann, ist die volle Auslastung ein oder mehrerer Ressourcen. Darunter zählen unter anderem die Central-Processing-Unit (CPU), der Arbeitsspeicher und die Speicherkapazität.

Jeder Node eines Clusters ist in der Lage, die Rate der eingehenden Nachrichten zu reduzieren. Dieser Mechanismus erhöht die Stabilität des Clusters, indem gezielt Publisher, die zu einer Überlast beitragen, gedrosselt werden [20]. Dadurch kann sich das Cluster von einer Überlastsituation erholen ohne die Servicequalität für Clients, die nicht zu der Überlastsituation beitragen, zu beeinträchtigen. Jeder Node eines Clusters berechnet durchgehend einen individuellen *Overload-Protection Level* auf Basis der Menge aller:

- eingehenden `PUBLISH` Nachrichten
- Retained-Messages
- verbundenen Clients
- Queued-Messages

Das Drosseln von Clients basiert auf einem Credit-System. Jeder Client erhält nach dem Aufbau einer Verbindung ein Guthaben von 50,000 Credits. Für jede Veröffentlichung einer Nachricht werden die Credits des Clients reduziert. Diese werden alle 200 Millisekunden solange um einen bestimmten Wert regeneriert, bis das maximale Guthaben von 50,000 wieder erreicht wurde. Der Regenerationswert richtet sich nach dem derzeitigen Overload-Protection Level des Nodes, mit dem der Client verbunden ist. [20]

Hat ein Client alle seine Credits verbraucht, wird TCP *Back pressure* auf dem TCP-Socket des Clients angewandt, woraufhin dieser keine Nachrichten mehr veröffentlichen kann. Überschreitet der Zeitraum, in dem TCP *Back pressure* für einen Client angewandt wird, den konfigurierten *keep-alive* Wert der Verbindung, wird diese unterbrochen [20]. Um TCP *Back pressure* zu erzeugen, stoppt der HiveMQ Node das Lesen der Daten aus dem TCP *receive buffer* des Clients, was ein Verkleinern des TCP *receive windows (rwin)* zur Folge hat. Wird das *receive window* null, kann der Client aufgrund des TCP *flow controls* keine neuen Daten mehr an den HiveMQ Node schicken [21, S. 219ff]. Das MQTT Protokoll hat für QoS 0 Nachrichten keine *flow control* implementiert, da diese nach dem Prinzip *fire and forget* abgeschickt werden. HiveMQ greift für das Drosseln der Nachrichten auf TCP Mechaniken zurück, damit unter anderem auch keine neuen QoS 0 Nachrichten an den HiveMQ Node geschickt werden.

3.2.2 Persistente Session

Um Nachrichten von einem MQTT Broker zu erhalten, muss sich ein Client mit diesem verbinden und eine Subscription auf einem Topic erstellen. Alle Subscriptions die ein Client erstellt, werden mit der TCP-Verbindung des Clients assoziiert. Jedes Mal, wenn die Verbindung zum Client unterbrochen wird, muss die Verbindung erneut aufgebaut und alle Subscriptions müssen erneut erstellt werden. Für einen Client mit limitierten Ressourcen ist dies eine Belastung. Um diese Belastung zu reduzieren, kann der Client beim Aufbau der Verbindung eine persistente Session anfordern, indem das entsprechende Bit eins von Byte acht des variablen Headers der `CONNECT` Nachricht (siehe Tabelle 4) auf null gesetzt wird. Ist dieses Bit hingegen auf eins gesetzt, werden alle Nachrichten und Informationen von vorherigen Sessions gelöscht. Eine persistente Session speichert folgende Daten auf dem Broker:

- Die Existenz einer persistenten Session. Als Identifikation wird die Clientkennung verwendet.
- Alle Subscriptions des Clients.
- Alle Nachrichten mit einem QoS Level 1 oder 2, die der Client noch nicht bestätigt hat.
- Alle Nachrichten mit einem QoS Level 1 oder 2, die der Client aufgrund der unterbrochenen Verbindung nicht erhalten konnte.
- Alle Nachrichten mit einem QoS Level 2, die der Client veröffentlicht hat, und die noch nicht bestätigt wurden.

Bei einer persistenten Session muss neben dem Broker auch der Client Informationen speichern. Er ist für die Persistierung folgender Daten verantwortlich:

- Seine eigene Clientkennung. Diese wird als Identifikationsmerkmal einer persistenten Session auf dem Broker genutzt und muss bei einer erneuten Verbindung immer gleich sein.
- Alle QoS Level 1 oder 2 Nachrichten, die noch nicht vom Broker bestätigt wurden.
- Alle QoS Level 2 Nachrichten, die der Client vom Broker erhalten hat, und die noch nicht vollständig bestätigt wurden.

Sobald ein Client die Verbindung zu einem Broker verliert und eine persistente Session aktiv ist, versucht der Broker die Daten des Clients so lange wie möglich zu persistieren, bis sich dieser erneut verbindet. Falls der Broker während der nicht verbundenen Zeit keine verfügbaren Ressourcen mehr hat, besteht die Möglichkeit, dass die persistente Session vom Broker gelöscht wird. [22]

3.2.2.1 Client-Takeover

Wie bereits im Kapitel 3.2.2 beschrieben, kann ein Client eine persistente Session bei einem Broker anfordern. Dabei werden zum Beispiel Nachrichten mit einem QoS Level von 1 oder 2 auf dem Broker nachgehalten, falls der Client seine Verbindung verliert.

Ein Client-Takeover findet statt, wenn der Broker eine neue Clientverbindung erhält, aber noch eine offene Verbindung mit derselben Clientkennung hat. Dies kann bei *halb-offenen TCP-Verbindungen* vorkommen, die, wie Andy Stanford-Clark beschreibt, bei TCP auftreten können:
“Although TCP/IP in theory notifies you when a socket breaks, in practice, particularly on things like mobile and satellite links, which often ’fake’ TCP over the air and put headers back on at each end, its quite possible for a TCP session to ’black hole’, i.e. it appears to be open still, but in fact is just dumping anything you write to it onto the floor.”[23]
Der Broker beendet die existierende Verbindung und assoziiert die persistente Session - falls vorhanden - nun mit dem neuen Client. [24]

3.3 Load Balancing

Load balancing wurde um 1990 als Unterstützung für eine horizontale Skalierung eingeführt, um Traffic auf Server-Infrastrukturen zu verteilen. Dadurch sollte eine erhöhte Verfügbarkeit der Services für Endnutzer erzielt werden [25, S. 2]. Bei einer horizontalen Skalierung werden nicht wie bei der vertikalen Skalierung die Ressourcen eines einzelnen Servers erweitert, sondern weitere Server bereitgestellt. Es gibt mehrere Ursachen, wann eine vertikale Skalierung an ihre Grenzen stößt [25]:

- Der Server ist nicht mehr erweiterbar an CPU, RAM oder Speicherplatz.
- Ein Neustart des Servers zur Durchführung einer vertikalen Skalierung ist nicht möglich.
- Es wird eine Hochverfügbarkeit erzielt.

Ein Load-Balancer wird bei der horizontalen Skalierung den einzelnen Servern vorgeschaltet, um eingehenden Traffic basierend auf diversern load balancing Algorithmen an die entsprechenden Server zu verteilen. Je nach Algorithmus sorgt der Load-Balancer (LB) dadurch für eine gleichmäßige Verteilung der Last auf das gesamte System. Es ist ebenfalls möglich, ausgefallene Server zu detektieren und die Last auf verbleibende Server umzuleiten. [25, S. 8]

Load balancing kann software- und hardwareseitig betrieben werden. Als Hardware-Load-Balancer (HLB) wird oftmals ein OSI-Layer vier oder sieben Switch verwendet. Software load balancing braucht, anders als Hardware load balancing, keine dedizierte Hardware und kann in jedem Server, Container oder jeder virtuellen Maschine eingesetzt werden [26]. Dabei kann jegliche Funktionalität eines HLBs vollständig ersetzt und erweitert werden. Ein HLB hat typischerweise Einfluss auf die physische Architektur der Komponenten und ist nicht nur eine reine logische Einheit wie ein Software LB. Software load balancing bietet mehr Flexibilität gegenüber Hardware load balancing auf Kosten von Performance. [27]

Load balancing kann zwischen Layer vier und sieben des Open-System-Interconnection (OSI) Modells stattfinden [27]. Das OSI-Modell standardisiert die Kommunikation zwischen verschiedenen Computer-Systemen in sieben Layern [25, S. 13ff].

1. Bitübertragungsschicht (engl. *Physical-Layer*)
2. Sicherrungsschicht (engl. *Datalink-Layer*)
3. Vermittlungsschicht (engl. *Network-Layer*)
4. Transportschicht (engl. *Transport-Layer*)
5. Sitzungsschicht (engl. *Session-Layer*)
6. Darstellungsschicht (engl. *Presentation-Layer*)
7. Anwendungsschicht (engl. *Application-Layer*)

Layer 4 Load Balancing Der LB lenkt den Traffic basierend auf Informationen des Network- und Transport-Layers. Diese beinhalten die TCP und User-Datagram-Protocol (UDP) Ports der Pakete sowie die Internet-Protocol (IP) Adressen der Empfänger und Absender. Layer vier Load-Balancer inspizieren für die Weiterleitung nicht die Daten des Paketes. [25, S. 15]

Layer 7 Load Balancing Ein Layer sieben LB inspiziert die Daten des gesendeten Paketes. Dafür muss der LB in der Lage sein, das verwendete Protokoll zu parsen. Es können komplexere Weiterleitungsregeln als bei einem Layer vier LB implementiert werden. Bei HTTP können beispielsweise Weiterleitungsentscheidungen basierend auf den HTTP-Headern oder den HTTP-Pfaden getroffen werden. [27, S. 15]

In einem verteilten System, zum Beispiel einem HiveMQ Cluster, bieten Load-Balancer eine transparente Abstraktion für Clients. Der Client verbindet sich mit einem LB und wird an einen HiveMQ Node weitergeleitet, ohne die Topologie des HiveMQ Clusters zu kennen. Für den Client macht es den Anschein, als würde der LB eine einziger HiveMQ Broker sein, der seinen Service anbietet.

3.3.1 Load Balancing Algorithmen

Der Load-Balancer bestimmt anhand eines load balancing Algorithmus, an welchen Server eine neue Anfrage verteilt wird. Welcher Algorithmus verwendet wird, hängt von dem Einsatz des Load-Balancers ab. Das Protokoll, die Server-Infrastruktur und das Verhalten der Clients sind wichtige Faktoren für die Auswahl des Algorithmus.

Im Folgenden werden die weit verbreitetsten load balancing Algorithmen erläutert.

Round-Robin Alle Server sind in einer Liste aufgeführt. Der round-robin Algorithmus leitet den Traffic immer an den ersten Server in der Liste weiter und verschiebt ihn an die letzte Stelle. Dieser Algorithmus teilt alle Anfragen gleichmäßig auf die Server auf und findet oft Anwendung, wenn alle Server über die gleichen Ressourcen verfügen. [27]

Least-Connection Der least-connection Algorithmus hält alle aktiven Clientverbindungen pro Server nach und verteilt neue Anfragen an den Server, der die wenigsten aktiven Verbindungen hat. [27]

Consistent-Hashing Der LB bestimmt den Server anhand eines Hash-Wertes von diversen Daten des eingehenden Paketes vom Client. Mögliche Daten sind typischerweise Quell-, Ziel IP-Adresse, Port oder Domain-Name. Zudem können auch Informationen aus OSI-Layer sieben verwendet werden. Durch diesen Algorithmus werden wiederkehrende Clients mit denselben Hash-Werten immer mit demselben Server verbunden. [27]

3.3.2 Envoy

Envoy ist ein OSI-Layer sieben Proxy, der für große, moderne und serviceorientierte Architekturen von Lyft entwickelt wurde. Im September 2017 wurde Envoy an die *Cloud Native Computing Foundation* gestiftet, die Cloud Projekte wie Kubernetes, Prometheus, containerd und Jaeger verwaltet. Der Envoy Quellcode wird öffentlich auf GitHub versioniert und verzeichnet mit Stand April 2021 bereits 670 Kontributoren und 16.500 Sterne auf GitHub [28]. Envoy wird unter anderem bei Slack [29] als Load-Balancer für Websockets oder in *service-meshes* als Edge-Proxy bei Istio [30] und Google Traffic-Director [31] eingesetzt.

Der in C++ programmierte und hochperformante Envoy Proxy stellt folgende abstrakte Features bereit [32]:

- **OSI-Layer 3/4 Filter:** Envoy erlaubt das Dazuschalten von TCP- oder UDP-Filtern in die Netzwerk-Pipeline. Filter wie zum Beispiel für MongoDB, Redis, Transport-Layer-Security (TLS) Authentifizierung wurden bereits in Envoy implementiert. Benutzerdefinierte Filter können direkt in C++ implementiert und eingebunden oder als WebAssembly (WASM) Modul integriert werden.
- **HTTP OSI-Layer 7 Filter:** Aufgrund der Popularität von HTTP wurden diverse OSI-Layer sieben HTTP-Filter in Envoy implementiert. Filter wie *buffering*, *routing* und *rate-limiting* sind bereits in Envoy integriert und können über einen HTTP-Connection-Manager konfiguriert werden.
- **Dynamische Konfiguration:** Envoy kann dynamisch über eine *Data-Plane API* provisioniert werden. Dabei können Daten wie Cluster, HTTP-Routen, Listener-Sockets oder TLS-Zertifikate von einer externen Quelle bereitgestellt werden.
- **Observeability:** Envoy verfolgt das Ziel, transparente Netzwerke zu schaffen. Um dies zu erreichen, werden umfangreiche Statistiken bereitgestellt, die zum Beispiel mit *statsd* aggregiert werden können.

Ein Proxy-Server, wie beispielsweise Envoy, dient als Gateway zwischen Clients und Servern. Clients verbinden sich mit einem Proxy-Server, der die Anfragen der Clients an einen Server weiterleitet, der diese bearbeiten kann. Es wird zwischen einem *forward* und einem *reverse* Proxy-Server unterschieden [33]:

- **Forward Proxy:** Der Proxy-Server versteckt die Informationen des Clients vor dem Server, der die Anfragen bearbeiten kann.
- **Reverse Proxy:** Der Proxy-Server versteckt den Server, der die Anfragen bearbeiten kann, vor dem Client.

Envoy kann als *reverse proxy* Load-Balancer betrieben werden und verteilt Anfragen transparent an Nodes eines Clusters.

4 Problembeschreibung

Die Clusterfähigkeit des HiveMQ Brokers ermöglicht über einen zentralen und ausfallsicheren HiveMQ Cluster Millionen von IoT-Geräten die Kommunikation untereinander. Um ein HiveMQ Cluster für MQTT Clients zu einem virtuellen Broker mit einer einzigen IP-Adresse oder Domain zu abstrahieren, wird wie in Kapitel 3.3 beschrieben, ein Load-Balancer benötigt. In diesem Kapitel werden Anforderungen für einen Load-Balancer definiert, um MQTT Traffic *klug* an alle Nodes eines HiveMQ Clusters zu verteilen.

4.1 Cluster-Discovery

Ein HiveMQ Cluster ist, wie in Kapitel 3.2.1 beschrieben, in der Lage, zur Laufzeit die Größe des Clusters anzupassen. Der Load-Balancer muss daher fähig sein, neue HiveMQ Nodes zur Laufzeit zu entdecken und neue Clients mit den neuen Nodes zu verbinden. Falls sich die Anzahl der Nodes verringert, darf der Load-Balancer keine neuen Verbindungen mehr mit den alten Nodes aufbauen. Dies würde zu Fehlern bei den Clients führen. Eine statische Konfiguration der HiveMQ Nodes ist somit nicht möglich.

4.2 Langlebige TCP-Verbindungen

In Kapitel 3.1 wurde erläutert, dass MQTT langlebige TCP-Verbindungen benutzt, um den Kommunikationsaufwand gering zu halten. Dies ermöglicht das *Push-Push* Prinzip, bei dem der Broker Nachrichten zum Client schicken kann, ohne dass dieser periodisch nach neuen Nachrichten fragen muss. Der Load-Balancer hält aktive Clientverbindungen aufrecht, bis Clients diese terminieren. Aktualisierungen des Load-Balancers oder dessen Konfiguration dürfen nicht zu einer Unterbrechung der aktiven Verbindungen führen.

4.3 Persistente Client-Sessions

Wie in Kapitel 3.2.2 beschrieben können Clients eine persistente Session bei einem Broker anfordern. Für den Fall eines Verbindungsabbruchs werden Nachrichten und Session relevante Daten gespeichert. Sobald der Client seine Verbindung wieder aufbaut, findet ein Client-Takeover (3.2.2.1) statt. Dieser Prozess findet bei einem Broker im Arbeitsspeicher statt. Dort wird zu den existierenden persistierten Daten der dazugehörige TCP-Socket ausgetauscht. Wenn sich der Client mit einem anderen Node des HiveMQ Clusters wieder verbindet, ist ein Client-Takeover komplexer. In diesem Szenario müssen die persistierten Daten des Clients im Cluster synchronisiert werden. Je mehr Daten für die Synchronisation anfallen, desto aufwändiger ist der Client-Takeover.

Im Optimalfall wird der neue Client mit demselben Node verbunden, mit dem er zuvor verbunden war, um interne Kommunikation und Latenzen beim Verbindungsaufbau zu verringern.

4.4 Überlastschutz

In Kapitel 3.2.1.1 wurde der Überlastschutz eines HiveMQ Brokers erläutert. Dies ermöglicht dem Broker die Verbindung individueller Clients, die viel Arbeitslast auf dem Broker erzeugen, zu unterbrechen. Falls sich der MQTT Client MQTT konform verhält, versucht dieser automatisch die Verbindung wiederherzustellen. In diesem Fall darf der Load-Balancer den Client nicht wieder mit demselben Node verbinden, da dieser Node mit dem Client erneut überlastet sein wird. Es muss ein Node mit einer geringeren Auslastung für diesen Client gefunden werden.

Das Overload-Protection Level gibt an, ob sich der HiveMQ Node in einer Überlast befindet. Der Load-Balancer sollte den Node im Fall einer Überlast unterstützen, indem er keine neuen Clients mit diesem Node verbindet.

4.5 Ungleiche Lastverteilung

Es gibt viele verschiedene MQTT Clients im Anwendungsfeld IoT. Manche sind beispielsweise Temperatursensoren, die alle zehn Minuten einen Wert auf ein Topic veröffentlichen. Andere sind Drehzahlsensoren, die alle 500 Millisekunden die aktuelle Drehzahl eines Motors zur Geschwindigkeitsermittlung veröffentlichen. Die Drehzahlsensoren verursachen durch die erhöhte Frequenz der Nachrichten mehr Traffic und Arbeitslast auf einem Broker als die Temperatursensoren.

Da MQTT ein zustandsbehaftetes und verbindungsorientiertes Protokoll ist, kann die load balancing Entscheidung nicht auf Paket-, sondern nur auf Verbindungsebene getroffen werden. Sobald eine Clientverbindung aufgebaut ist, werden alle Nachrichten des Clients immer an denselben Node weitergeleitet. Aus diesem Grund kann bei einem *round-robin* oder *least-connection* load balancing Algorithmus das zuvor beschriebene Verhalten zu einer ungleichen Lastverteilung im Cluster führen. Clientseitiges load balancing, wie zum Beispiel mit *Shared-Subscriptions* (siehe Kapitel 3.1.7) oder dem Verteilen von ausgehenden Nachrichten des Clients auf mehrere TCP-Verbindungen zu einem HiveMQ Cluster, ist nicht immer möglich. Wenn ein HiveMQ Cluster als *Software-as-a-Service* angeboten wird, hat der Betreiber keinen direkten Einfluss auf die MQTT Clients, die sich mit dem Cluster verbinden.

Ein Load-Balancer für MQTT muss in der Lage sein, die eingehenden Verbindungen basierend auf der derzeitigen Arbeitslast der einzelnen Nodes zu gewichten.

5 Lösungskonzept

Im folgenden Kapitel werden alle Problematiken aus Kapitel 4 detailliert analysiert und konzeptuelle Lösungsansätze entwickelt. In Kapitel 5.1 wird die Integration einer dynamischen HiveMQ Cluster-Discovery in Envoy beschrieben. Kapitel 5.2 untersucht die Lastverteilung in einem HiveMQ Cluster mit Fokus auf die gleichmäßige Verteilung von MQTT Clients anhand einer Envoy Control-Plane (siehe Kapitel 5.2.3). Abschließend wird in Kapitel 5.3 die Problemstellung der persistenten Client-Sessions (siehe Kapitel 4.3) behandelt.

5.1 Cluster-Discovery

HiveMQ verfügt über mehrere Mechanismen zur Identifikation von individuellen Nodes, die in das Cluster aufgenommen werden sollen (siehe 3.2.1). Ein Load-Balancer muss fähig sein, das HiveMQ Cluster auf derselben Datenbasis wie HiveMQ zu formen. Wenn der LB eine andere Datenbasis benutzt, könnte die Topologie des HiveMQ Clusters im LB von der tatsächlichen Topologie abweichen.

Envoy verfügt über folgende Möglichkeiten, Nodes eines Clusters zu bestimmen:

- **Static:** Alle Nodes eines Clusters werden statisch in die Envoy Konfiguration eingetragen.
- **Strict Domain-Name-System (DNS):** Envoy löst periodisch und asynchron einen konfigurierten DNS-Namen auf. Jede eingetragene IP-Adresse wird zu einem Node des Clusters. Falls ein Node entfernt wird, werden keine neuen Clients mehr mit diesem Node verbunden. Mit der Variable `dns_refresh_rate` kann die Frequenz, in welcher der DNS-Eintrag abgefragt wird, bestimmt werden.
- **Logical DNS:** Ähnlich wie bei Strict DNS löst Envoy einen konfigurierten DNS-Namen auf. Bei jeder neuen eingehenden Verbindung wird der DNS-Name erneut aufgelöst und die erste IP-Adresse als Ziel der neuen Verbindung genommen.
- **Original Destination:** Envoy leitet eingehende Verbindungen anhand der *Redirect-Metadata* weiter. Eingehende Verbindungen müssen dafür mit einem *iptables REDIRECT*, *TPROXY target* oder *Proxy-Protocol* an Envoy weitergeleitet werden.
- **Endpoint-Discovery Service:** Envoy ruft die Nodes eines Clusters bei einem *xDS Management-Server* ab. Es werden Java und Golang Bibliotheken angeboten, um einen Management-Server für Envoy zu programmieren und bereitzustellen. Somit ist es möglich, eine komplexe Service-Discovery zu implementieren.

[34] In Kapitel 3.2.1 wurden folgende Methoden der HiveMQ Cluster-Discovery erläutert:

- static
- multicast
- broadcast
- extension

```yaml
clusters:
- name: hivemq
  connect_timeout: 0.25s
  type: STRICT_DNS
  dns_lookup_family: v4_only
  lb_policy: round_robin
  load_assignment:
    cluster_name: hivemq
    endpoints:
    - lb_endpoints:
      - endpoint:
          address:
            socket_address:
              address: example.cluster.internal
              port_value: 1883
```

Abbildung 2: Es wird eine statische Envoy Cluster Konfiguration gezeigt, die alle Nodes mit der Strict DNS Methode ermittelt.

Envoy und HiveMQ stellen eine statische Cluster-Discovery zur Verfügung. Diese Methode ist für Cloud- oder Containerumgebungen suboptimal, da sie keine dynamische Veränderung der Cluster-Topologie zulässt. Container-Management-Umgebungen, wie zum Beispiel Kubernetes, erlauben dynamische Vorgänge wie das Skalieren der Replika-Sets oder Rolling-Updates. Eine statische Clusterkonfiguration schließt den Einsatz dieser oder ähnlichen dynamischen Vorgängen aus.

Eine weitere gemeinsame Cluster-Discovery Methode ist Strict DNS. HiveMQ hat diese Methode nicht in der Standardversion implementiert, stellt für diesen Anwendungsfall aber eine frei zugängliche Erweiterung bereit. Bei Strict DNS werden periodisch alle Einträge zu einem gegebenen DNS Eintrag abgefragt und als Nodes des Clusters anerkannt. Die Frequenz der Abfrage wird in Envoy mit der Variable `dns_refresh_rate` bestimmt. In der HiveMQ Erweiterung ist die Einstellung der Frequenz derzeit noch nicht möglich. Ein Issue [35] und ein Pull-Request [36] wurden bereits zu diesem Feature auf GitHub erstellt.

Anders als bei der statischen Clusterkonfiguration ist bei der Strict DNS Methode eine dynamische Änderung der Nodes möglich. In Cloudumgebungen wie Kubernetes werden DNS-Zonen dynamisch von Plugins wie *CoreDNS* verwaltet.[37] Die Strict DNS Methode kann auf Änderungen dieser Zonen reagieren und dynamisch die Topologie des HiveMQ Clusters anpassen.

Der Quellcodeauszug 2 zeigt eine Envoy Clusterkonfiguration, die den DNS-Namen `example.cluster.internal` auflöst und neue Verbindungen auf alle Einträge an den Port `1883` verteilt. Die *HiveMQ DNS Cluster-Discovery Extension* [38] muss auf allen Nodes installiert sein. Der Quellcodeauszug 3 zeigt eine Konfigurationsdatei der Erweiterung, die das Cluster aus allen Einträgen des DNS-Namens `example.cluster.internal` bildet.

```
discoveryAddress=example.cluster.internal
resolutionTimeout=30
```

Abbildung 3: Dies ist eine HiveMQ DNS Cluster-Discovery Konfigurationsdatei, die alle 30 Sekunden die Domain *example.cluster.internal* auflöst.

5.2 Lastverteilung HiveMQ Cluster

In Kapitel 4.5 wurde erläutert, dass in einem HiveMQ Cluster eine ungleiche Lastverteilung durch die verschiedenen Verhaltensweisen der Clients auftreten kann. In dem Fall, dass ein Node eine höhere Auslastung als ein anderer Node aufweist, soll die Verteilung neuer Clients an die entstandene Situation angepasst werden. Es müssen mehr Clients mit Nodes, die eine geringere Auslastung vorweisen, verbunden werden. Durch den weighted round-robin load balancing Algorithmus wird die Verteilung neuer Clients angepasst. Individuelle Gewichtungen pro Node bestimmen die Wahrscheinlichkeit, mit der ein neuer Client mit einem Node verbunden wird.

Für die Bestimmung der Gewichtungen des weighted round-robin load balancing Algorithmus müssen geeignete Kriterien gefunden werden, die die Auslastungen der einzelnen Nodes widerspiegeln. Mögliche Indikatoren bieten Systemressourcen wie CPU, Arbeitsspeicher, Festplattenspeicher und Netzwerkbandbreite. Im Rahmen der vorliegenden Arbeit wird untersucht, wie sich ein weighted round-robin load balancing Algorithmus, basierend der CPU Auslastungen der einzelnen Nodes, auf den Betrieb des Clusters im Vergleich zu einem round-robin und least-connection load balancing Algorithmus auswirkt.

5.2.1 Testszenarien

Um den round-robin, least-connection und weighted round-robin load balancing Algorithmus in Bezug auf die gleichmäßige Verteilung der Last in einem HiveMQ Cluster zu vergleichen, werden Testszenarien entworfen, die eine ungleiche Lastverteilung in einem HiveMQ Cluster erzeugen.

- **Szenario 1:** In *Bare-Metal* Deployments sind die Server-Kapazitäten an die verfügbare Hardware geknüpft. Es besteht die Möglichkeit, dass ein Cluster aus unterschiedlich dimensionierten Nodes besteht. In Szenario eins besteht das HiveMQ Cluster aus zwei Nodes mit acht CPU Kernen und acht Gigabyte (GB) Random-Access-Memory (RAM) sowie einem Node mit vier CPU Kernen und acht GB RAM. Im Verlauf von zehn Minuten werden sich 5.000 Clients mit ähnlichem Verhalten mit dem Cluster verbinden und Nachrichten veröffentlichen sowie Topics abonnieren.
- **Szenario 2:** Eine dynamische Topologieänderung des HiveMQ Cluster kann durch mehrere Aktionen ausgelöst werden. Dazu zählen Rolling-Upgrades oder Skalierungen des Clusters aufgrund einer Überdimensionierung oder aufgebrauchter Ressourcen. In diesem Szenario wird ein zwei Node Cluster mit jeweils acht CPU Kernen und acht GB RAM pro Node mit einem neuen Node mit selben Dimensionen zur Laufzeit erweitert. Vor der Erweiterung sind 3.000 leichtgewichtige Clients mit dem

Cluster verbunden. Nach der Erweiterung des Clusters verbinden sich zusätzlich 600 leistungsstarke Clients.

- **Szenario 3:** Die Dimensionen des Clusters sind dieselben wie in Szenario zwei. In diesem Szenario verbinden sich die leistungsstarken Clients vor der Erweiterung und die leichtgewichtigen Clients nach der Erweiterung des Clusters.

Szenario zwei wird in folgende Phasen untergliedert:

- **Phase 1:** Formen des Clusters aus zwei Nodes (Node eins und zwei) mit jeweils acht CPU Kernen und acht GB RAM.
- **Phase 2:** Verbinden von 3.000 Subscribern, die zufällig 10 von 1.000 Topics abonnieren und jedes Topic mindestens einmal abonniert wird.
- **Phase 3:** Verbinden von 2.000 Publishern. Jeder Publisher veröffentlicht zufällig alle 500 - 2.000 Millisekunden eine Nachricht mit der Payload `Hello World`, einem zufälligen QoS Level auf ein zufälliges Topic.
- **Phase 4:** Ein dritter Node (Node drei) mit acht CPU Kernen und acht GB RAM tritt dem HiveMQ Cluster bei.
- **Phase 5:** Es verbinden sich 300 Publisher. Jeder Publisher veröffentlicht Nachrichten wie in Phase drei beschrieben, jedoch zufällig alle 15 bis 25 Millisekunden.
- **Phase 6:** Es verbinden sich 300 Publisher. Jeder Publisher veröffentlicht Nachrichten wie in Phase vier beschrieben.

Szenario drei hat dieselben Phasen wie Szenario zwei in unterschiedlicher Reihenfolge. Phase drei wird mit Phase fünf und sechs getauscht.

Alle Szenarien werden mit einem round-robin und least-connection load balancing Algorithmus ausgeführt, um die Last der einzelnen Nodes zu untersuchen. Zur Veranschaulichung der Testszenarien werden die CPU Auslastungen und deren Standardabweichung mit Grafana visualisiert. Eine blau gestrichelte, senkrechte Linie kennzeichnet das Beenden aller Phasen des jeweiligen Testszenarios. Die Metriken werden alle zwei Sekunden mit Prometheus erfasst.

Die Standardabweichung der CPU Auslastungen dient als Indikator für die gleichmäßige Verteilung der Last im Cluster. Je kleiner der Wert der Standardabweichung ist, desto gleichmäßiger ist die Last verteilt. Die Standardabweichung ist die Quadratwurzel der Varianz und somit ein Maß für die Streubreite von Werten um deren Mittelwert in der gleichen Maßeinheit.[39, S. 72]. Die Standardabweichung wird mit der Prometheus Funktion `stddev` berechnet.

Szenario 1 Abbildung 5 und 4 zeigen den zeitlichen Verlauf von Testszenario eins für einen round-robin und least-connection Load-Balancer. Beide verhalten sich in diesem Szenario ähnlich. Alle Clients werden gleichmäßig auf die Nodes verteilt. Da die Clients ein ähnliches Verhalten aufweisen, entsteht auf jedem Node eine gleiche Belastung. Node drei hat nur die Hälfte der CPU Kerne zur Verfügung wie Node eins und zwei und ist daher nicht in der Lage, die Last so schnell abzuarbeiten wie Node eins oder zwei. Somit entsteht ein ungleiches Lastverhalten im Cluster.

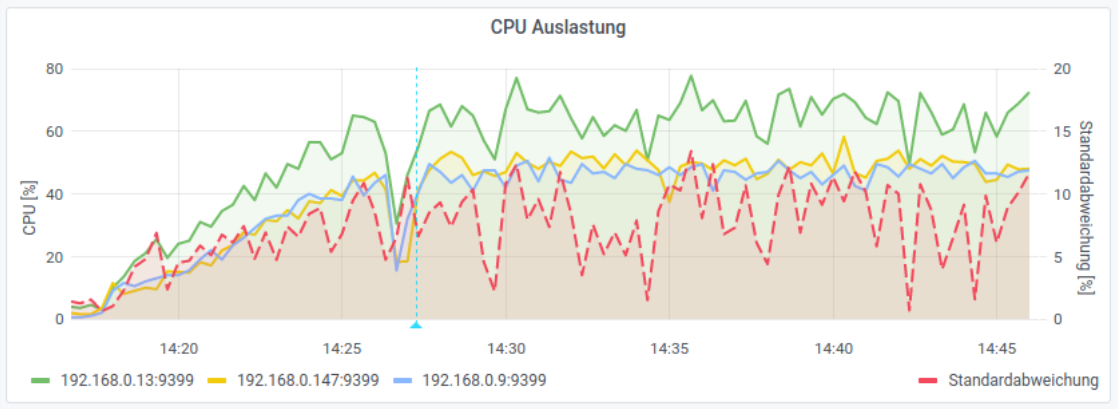

Abbildung 4: Testszenario 1 - least-connection Load-Balancer

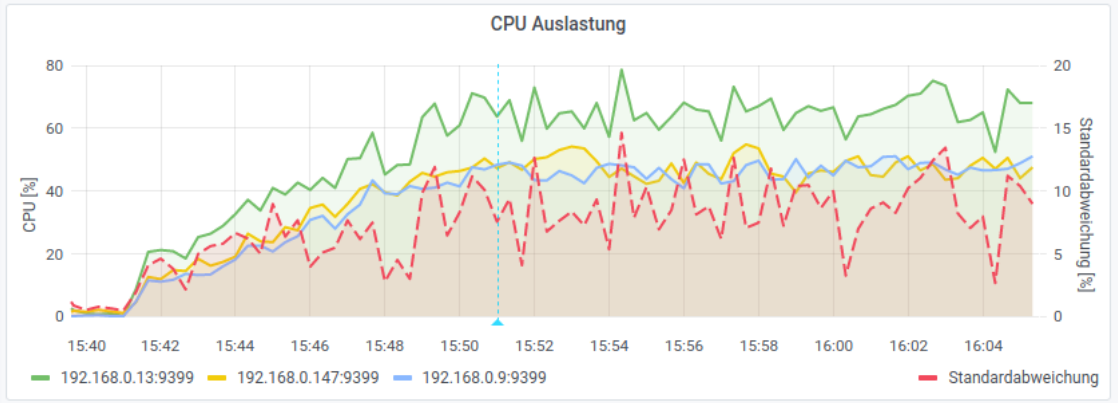

Abbildung 5: Testszenario 1 - round-robin Load-Balancer

Szenario 2 - Least-Connection Abbildung 6 zeigt den zeitlichen Verlauf von Testszenario zwei bei einem least-connection LB. Nach Abschluss von Phase sechs hat dieses Szenario eine Standardabweichung von zehn Prozent. Die Last auf dem Cluster ist ungleich verteilt, da Node drei doppelt so hoch ausgelastet ist wie Node eins und Node zwei. Dieses Ungleichgewicht wird durch Phase sechs verstärkt. Auf Node eins und Node zwei sind jeweils 3.500 Clients verbunden auf Node drei hingegen nur 300 Clients. Der least-connection Load-Balancer verteilt nun alle Clients aus Phase sechs auch auf Node drei. Trotz der noch höheren Auslastung auf Node drei würden die nächsten 2.900 Clients ebenfalls mit Node drei verbunden werden.

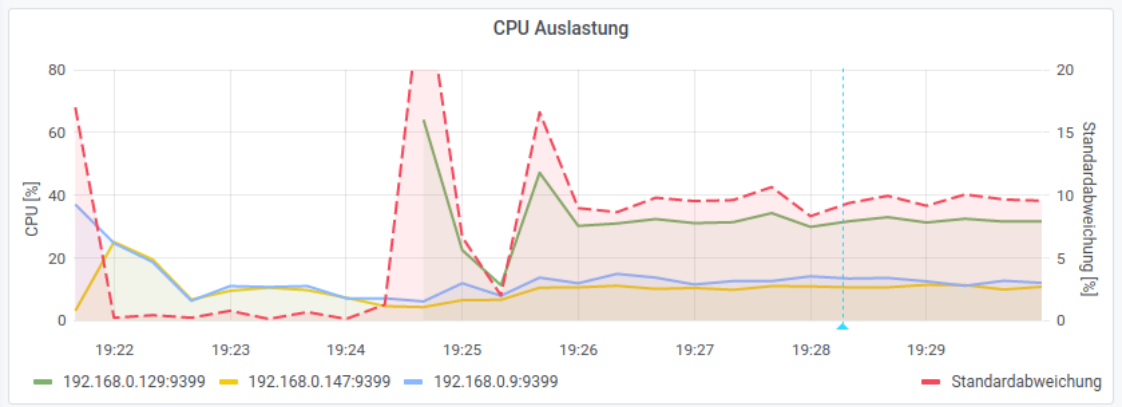

Abbildung 6: Testszenario 2 - least-connection Load-Balancer

Szenario 2 - Round-Robin Abbildung 7 zeigt den zeitlichen Verlauf von Testszenario zwei bei einem round-robin LB. Nach Abschluss von Phase sechs hat dieses Szenario eine Standardabweichung von drei Prozent. Nach Phase fünf und sechs ist die Last auf dem Cluster im wesentlichen gleichmäßig verteilt. Node eins und zwei sind um die Anzahl der Clients aus Phase zwei und drei stärker belastet. Diese sind jedoch leichtgewichtige Clients und erzeugen demnach nicht viel Last auf den Nodes. Der round-robin Algorithmus verteilt alle Clients gleichmäßig auf dem Cluster. Daher werden nach dem Beitreten von Node drei in Phase vier nicht wie bei dem least-connection Algorithmus alle neuen Clients auf den neuen Node verteilt.

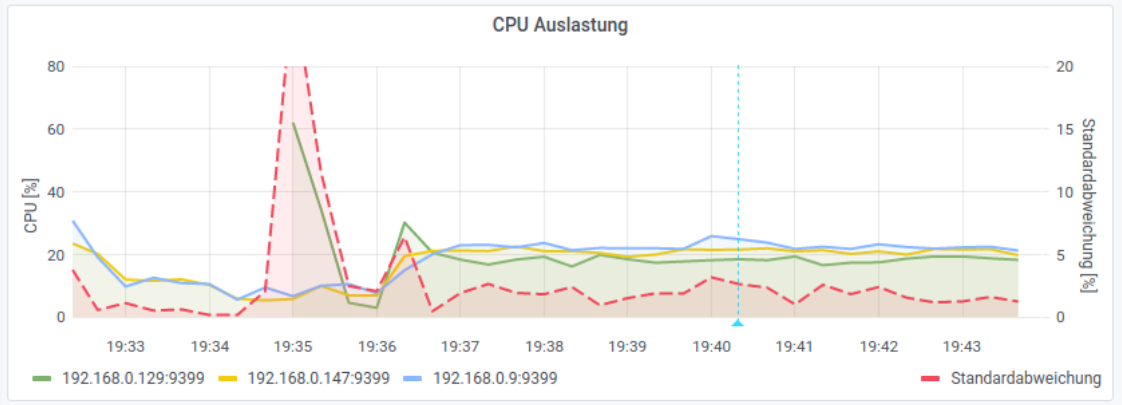

Abbildung 7: Testszenario 2 - round-robin Load-Balancer

Szenario 3 - Least-Connection Abbildung 8 zeigt den zeitlichen Verlauf von Testszenario drei bei einem least-connection LB. Nach Abschluss von Phase sechs hat dieses Szenario einen Lastindikator von sechs. Node eins und zwei sind mehr belastet als Node drei, da alle Clients aus Phase drei und vier mit diesen beiden Nodes verbunden sind. Alle Clients aus Phase sechs sind mit Node drei verbunden, jedoch verursachen diese nicht so viel Arbeitslast wie die Clients aus Phase drei und vier.

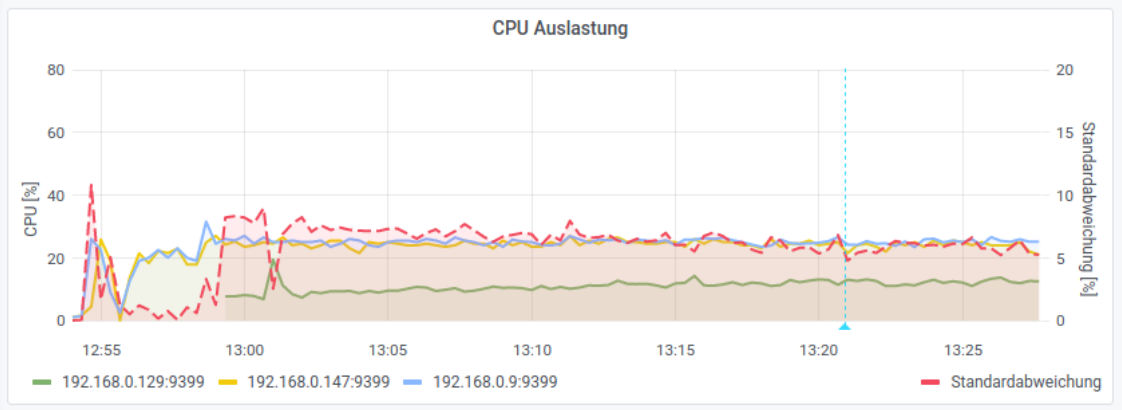

Abbildung 8: Testszenario 3 - least-connection Load-Balancer

Szenario 3 - Round-Robin Abbildung 9 zeigt den zeitlichen Verlauf von Testszenario drei bei einem round-robin LB. Nach Abschluss von Phase sechs hat dieses Szenario eine Standardabweichung von acht Prozent. Nach Phase sechs sind Node eins und zwei jeweils dreifach so sehr ausgelastet wie Node drei. Die Last auf dem Cluster ist somit ungleich verteilt. Die Clients aus Phase drei und vier werden nur auf Node eins und zwei verteilt, da der dritte Node noch nicht dem Cluster beigetreten ist. Die leichtgewichtigen Clients aus Phase sechs werden durch den round-robin Algorithmus auf alle Nodes verteilt, obwohl Node drei gar nicht ausgelastet ist. Im Idealfall würde der LB alle 3.000 Clients aus Phase sechs auf Node drei verteilen.

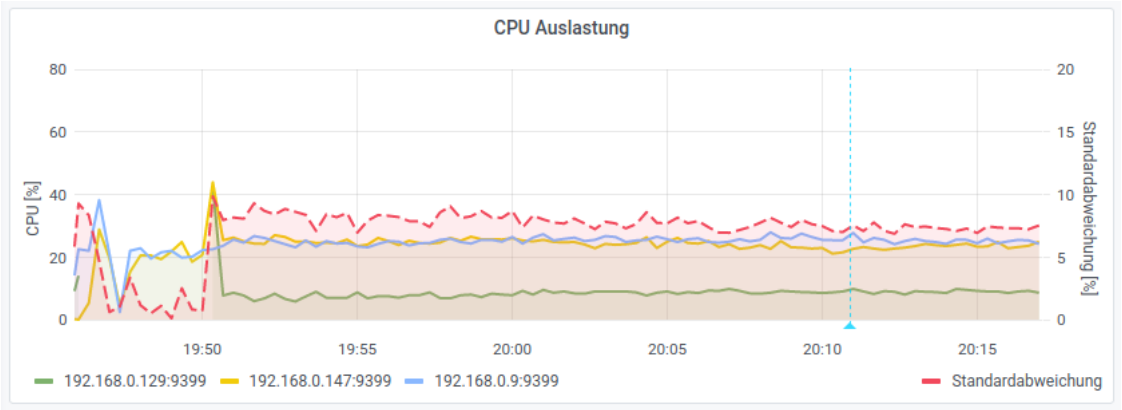

Abbildung 9: Testszenario 3 - round-robin Load-Balancer

5.2.2 Weighted Round-Robin

Kapitel 5.2.1 zeigt, dass ein least-connection Load-Balancer in Szenario drei und ein round-robin LB in Szenario zwei die Last gleichmäßig verteilen kann. Bei den anderen Szenarien befindet sich das Cluster anschließend in einem ungleichen Lastverhältnis. Da MQTT auf langlebigen TCP-Verbindungen basiert, wird sich dieses Lastverhältnis nicht verändern, bis die Clients ihre Verbindung neu aufbauen oder ihr Verhalten verändern. Der LB muss dynamisch auf ein bestimmtes Client-Verhalten reagieren und Clients basierend auf der Arbeitslast der einzelnen Nodes verteilen.

Bei dem weighted round-robin load balancing Algorithmus wird jedem Node eines Clusters eine Gewichtung zugeordnet. Die Gewichtung des Nodes bestimmt die Wahrscheinlichkeit, mit der ein neuer Client mit einem Node verbunden wird. Um mehr Clients mit Nodes von geringerer Auslastung zu verbinden, muss die derzeitige Auslastung direkten Einfluss auf die Gewichtung des Nodes haben. Diese Mechanik wird bereits bei anderen Protokollen wie HTTP eingesetzt. Eine load balancing Entscheidung bei MQTT ist jedoch relevanter als bei Protokollen wie HTTP, weil die Entscheidung für die Dauer der Verbindung nicht mehr verändert werden kann. Da MQTT von langlebigen TCP-Verbindungen profitiert, kann der LB die Last eines bestimmten Clients nach dem Verbindungsaufbau nicht mehr steuern.

Envoy hat mehrere load balancing Algorithmen, unter anderem weighted round-robin, implementiert. Jedem Node eines Clusters wird eine Gewichtung in Form eines Integer-Wertes zugeordnet. Je größer der Wert im Verhältnis zu allen anderen Nodes des Clusters ist, desto höher ist die Wahrscheinlichkeit, dass ein neuer Client mit dem Node verbunden wird. Der Wert der Gewichtung muss größer als eins und die Werte aller Nodes addiert dürfen nicht größer als 4294967295 sein. Um den Prozentsatz zu berechnen, wie viele Clients mit einem individuellen Node verbunden werden, teilt Envoy die Gewichtung des jeweiligen Nodes durch die Summe der Gewichtungen aller Nodes und multipliziert das Ergebnis mit 100. [40]

Tabelle 5 zeigt drei Nodes mit ihren Gewichtungen und den dazu errechnetem Prozentsatz aller Clients, die mit diesem Node verbunden werden.

Node	**Gewichtung**	**Prozent Traffic**
Node 1	10	40%
Node 2	10	40%
Node 3	5	20%

Tabelle 5: Die Tabelle zeigt drei Nodes eines Clusters und ihren Gewichtungen bei einem weighted round-robin load balancing Algorithmus des Envoys. Die Spalte *Prozent Traffic* gibt an, mit welcher Wahrscheinlichkeit ein neuer Client mit diesem Node verbunden wird.

Der Quellcodeauszug 10 zeigt eine statische Envoy Konfiguration, um ein Cluster aus drei Nodes mit den Gewichtungen aus Tabelle 5 zu formen.

```yaml
clusters:
- name: hivemq
  connect_timeout: 0.25s
  type: STATIC
  dns_lookup_family: v4_only
  lb_policy: round_robin
  load_assignment:
    cluster_name: hivemq
    endpoints:
    - lb_endpoints:
      - load_balancing_weight: 10
        endpoint:
          address:
            socket_address:
              address: 192.168.0.2
              port_value: 1883
      - load_balancing_weight: 10
        endpoint:
          address:
            socket_address:
              address: 192.168.0.3
              port_value: 1883
      - load_balancing_weight: 5
        endpoint:
          address:
            socket_address:
              address: 192.168.0.4
              port_value: 1883
```

Abbildung 10: Es wird eine statische Envoy Cluster Konfiguration gezeigt, bei der neue Clientverbindungen basierend dem weighted round-robin load balancing Algorithmus verteilt werden.

Die Problematik bei der Konfiguration aus Quellcodeauszug 10, ist die statische Angabe der Nodes mit ihren Gewichtungen. Eine dynamische Cluster-Discovery wie in Kapitel 5.1 beschrieben, ist nicht möglich. Neben der statischen Konfigurationsdatei bietet Envoy zwei Mechanismen für eine dynamische Konfiguration an.

- **Dynamisch via Dateisystem:** Envoy liest Konfigurationsdateien, die das xDS Protokoll implementiert haben, vom Dateisystem ein. Wenn sich die Dateien in dem Dateisystem ändern, aktualisiert Envoy automatisch seine Konfiguration. [41]
- **Dynamisch via Control-Plane:** Envoy bezieht seine Konfiguration dynamisch von einer Control-Plane. Eine Control-Plane ist ein Application-Programming-Interface (API) Server, der Konfigurationen an Envoy Instanzen schickt. Die Schnittstelle, um Konfigurationen zwischen Envoy und Control-Plane auszutauschen, ist in der *Data-Plane API* (siehe [42]) dokumentiert. [43]

Die dynamische Konfiguration via Dateisystem ist geeignet für einzelne Envoy Instanzen, da keine individuelle Control-Plane programmiert werden muss. Möchte man jedoch mehrere Envoy Instanzen mit derselben Konfiguration betreiben, muss bei einer Aktualisierung der Konfiguration die neue Datei auf die Dateisysteme aller Envoys verteilt werden. Für den Anwendungsfall mehrerer Envoy Instanzen ist das dynamische Verteilen mit einer Control-Plane besser geeignet. Jeder Envoy wird mit einer statischen Datei konfiguriert, in der die Adresse der Control-Plane eingetragen ist. Sobald eine Envoy Instanz startet, registriert sich diese an der Control-Plane und erhält seine Konfiguration. Das Verteilen von Konfigurationsaktualisierungen an die Envoy Instanzen wird nun von der Control-Plane übernommen.

Quellcodeauszug 11 zeigt eine statische Konfigurationsdatei zur Registrierung bei einer Control-Plane, die unter `example.controlplane.internal:18000` erreichbar ist. Die Konfigurationsdatei ist in drei Teile gegliedert:

- `node:` Identifikation der Envoy Instanz bei der Control-Plane. Eine Control-Plane kann verschiedene Konfigurationen verwalten. Durch `node.id` kann die Envoy Instanz einer Konfiguration zugeordnet werden.
- `dynamic_resources:` Gibt die Quelle der dynamischen Provisionierung an. Im Quellcodeauszug 11 wird eine gRPC API Namens `xds_cluster` verwendet.
- `static_resources:` Es wird die Quelle für eine dynamische Provisionierung definiert. Im Quellcodeauszug 11 wird eine Quelle namens `xds_cluster` definiert. Diese Quelle wird in `dynamic_resources` referenziert.

Eine Control-Plane kann in jeder Programmiersprache entwickelt werden und muss lediglich die Spezifikationen der Data-Plane API [42] berücksichtigen. Um den Einstieg für Entwickler:innen zu erleichtern, stellt Envoy Bibliotheken für Java und Golang bereit, die eine Abstraktion der Data-Plane API bieten.

Um eine horizontale Skalierung der Envoy Instanzen zu ermöglichen, wird im Rahmen der Thesis eine dynamische Konfiguration via Control-Plane in Golang entwickelt.

```yaml
node:
  cluster: hivemq-controller
  id: hivemq

dynamic_resources:
  ads_config:
    api_type: GRPC
    transport_api_version: V3
    grpc_services:
    - envoy_grpc:
        cluster_name: xds_cluster
  cds_config:
    resource_api_version: V3
    ads: {}
  lds_config:
    resource_api_version: V3
    ads: {}

static_resources:
  clusters:
  - connect_timeout: 1s
    type: STRICT_DNS
    typed_extension_protocol_options:
      envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
        "@type": >-
          type.googleapis.com
          /envoy.extensions.upstreams.http.v3.HttpProtocolOptions
        explicit_http_config:
          http2_protocol_options: {}
    name: xds_cluster
    load_assignment:
      cluster_name: xds_cluster
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: example.controlplane.internal
                port_value: 18000
```

Abbildung 11: Der Quellcodeauszug zeigt eine statische Envoy Konfiguration, um eine gRPC-Verbindung mit einer Control-Plane unter der Adresse example.controlplane.internal:18000 aufzubauen.

5.2.3 Control-Plane

Eine Envoy Control-Plane wird als eigenständiger Webservice betrieben und muss von allen Envoy Instanzen, die ihre Konfiguration von der Control-Plane erhalten sollen, erreichbar sein.

Abbildung 12 zeigt den Fluss des Traffics bei einem Envoy als Load-Balancer, konfiguriert von einer Control-Plane, und einem HiveMQ Cluster. Die Envoy Instanz erhält ihre Listener- und Clusterkonfiguration von der Control-Plane. MQTT Clients verbinden sich mit dem LB und werden an einen entsprechenden Node des HiveMQ Clusters weitergeleitet.

Envoy stellt eine beispielhafte Control-Plane [44] bereit, die als Starthilfe für eine anwendungsspezifische Control-Plane dient. Für den Anwendungsfall MQTT soll die Control-Plane eine Konfiguration ähnlich dem Quellcodeauszug 10 erzeugen. Dabei werden die Nodes und deren Gewichtung von der Control-Plane bestimmt.

Die in Kapitel 5.1 beschriebene HiveMQ Cluster-Discovery schließt den Einsatz des weighted round-robin Algorithmus in Envoy aus. Bei der Strict DNS Cluster-Discovery Methode ermitteln die Envoy Instanzen die Nodes eines Clusters und propagieren diese nicht an die Control-Plane zurück. Somit kann die Control-Plane den einzelnen Nodes keine spezifischen Gewichtungen vergeben. Anstatt den Envoy Instanzen einen Domain-Namen mitzuteilen und diesen auflösen zu lassen, muss der DNS Cluster-Discovery Mechanismus in der Control-Plane implementiert werden. Den Envoy Instanzen werden somit explizit alle HiveMQ Node IP-Adressen mit Gewichtung und Port mitgeteilt.

5.2.4 DNS Cluster-Discovery

Die Implementierung der HiveMQ Cluster-Discovery in der Control-Plane muss sich konzeptuell der DNS Cluster-Discovery von der HiveMQ Erweiterung anlehnen. Wie in Kapitel 5.1 beschrieben würde sonst die Topologie des HiveMQ Clusters im Envoy von der tatsächlichen Topologie divergieren.

Die Control-Plane muss asynchron und periodisch einen gegebenen Domain-Namen auflösen und alle hinterlegten IP-Adressen als Nodes eines HiveMQ Clusters eintragen. Das Intervall der periodischen Auflösung sollte sehr gering gehalten sein, um auf Topologieänderungen schnell reagieren zu können. Cloudumgebungen wie Kubernetes haben in der Regel eigene DNS Services installiert, die eine hohe Abfragerate verarbeiten können.

Quellcodeauszug 13 zeigt die Auflösung aller IP-Adressen eines Domain-Namens in Golang mit der Standard `net` Bibliothek. Durch den Start der Funktion `refreshDNS` als Go-Routine, wird in der `main` Funktion ein Warten auf die Terminierung der `refreshDNS` Funktion vermieden. Die `refreshDNS` Funktion erneuert die IP-Adressen periodisch in gegebenem Intervall.

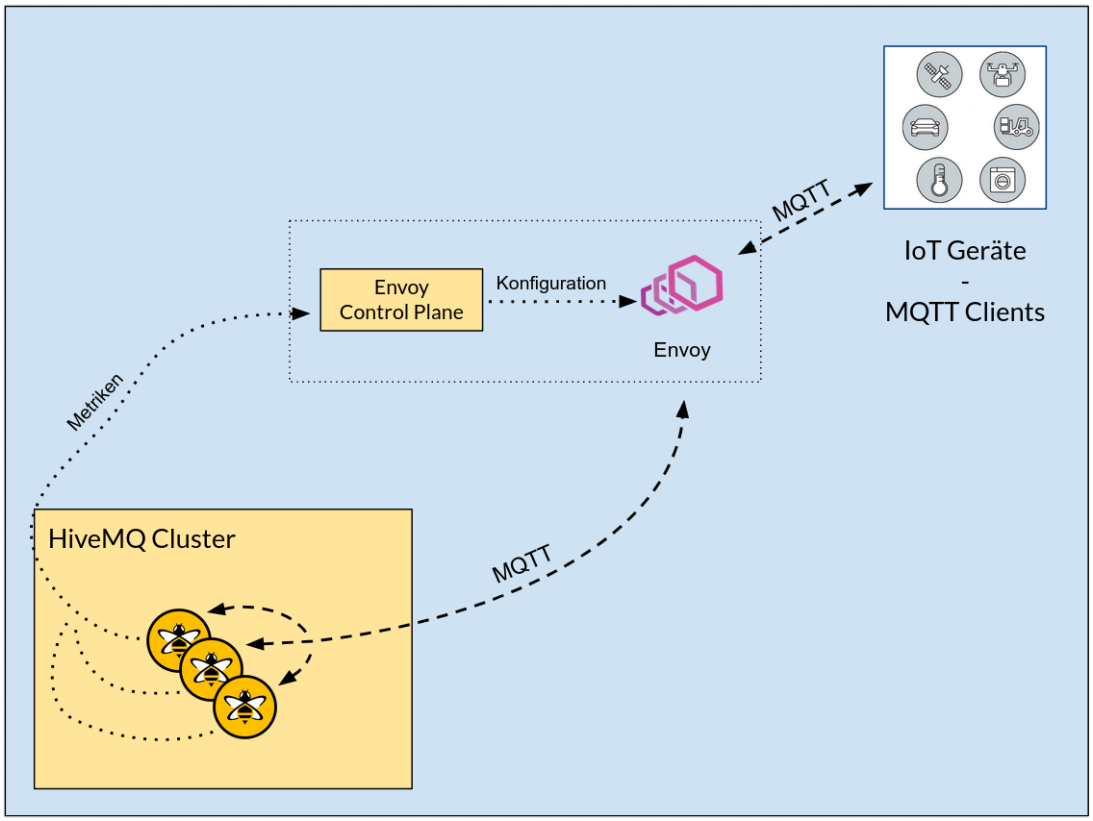

Abbildung 12: Die Abbildung zeigt die Architektur eines Envoy Load-Balancers konfiguriert von einer Control-Plane mit einem HiveMQ Cluster und MQTT Clients. Verbindungspfeile zwischen den einzelnen Komponenten sind Netzwerkverbindungen.

5.2.5 Weighted CPU Round-Robin

Die Testszenarien aus Kapitel 5.2.1 zeigen, dass es bei MQTT viele verschiedene Client-Verhaltensmuster gibt. Zudem kann ein Client sein Verhalten zur Laufzeit verändern. Folglich kann der Load-Balancer diesen Client nicht mehr mit einem anderen Node verbinden. Eine TCP-Verbindung bei MQTT korreliert nicht mit der verursachten Arbeitslast auf dem Broker. In Szenario zwei haben 300 Publisher doppelt so viel Last erzeugt wie 3.000 Subscriber und 1.000 Publisher zusammen. Nur anhand der Anzahl aktiver Verbindungen pro Node kann im Fall MQTT keine optimale load balancing Entscheidung getroffen werden.

Über eine Control-Plane kann die Gewichtung einzelner Nodes eines Clusters im Envoy dynamisch angepasst werden. Die Control-Plane muss periodisch und asynchron die aktuelle CPU Auslastung eines jeden Nodes abfragen und darauf basierend die Nodes gewichten. HiveMQ stellt die aktuelle CPU Auslastung über die Prometheus Metriken zur Verfügung, die einen Wertebereich zwischen 0 und 100 hat. Dieser Wert ist über alle CPU Kerne gemittelt. Quellcodeauszug 14 zeigt die Abfrage der Prometheus Metriken eines HiveMQ Brokers gefiltert nach `cpu_total_total`. Der Datentyp dieser Metrik ist ein *Gauge*, der einen einzelnen numerischen Wert darstellt.[45] Der Wert beträgt in diesem Beispiel 5.0 und bedeutet, dass die aktuelle CPU Auslastung über alle CPU Kerne gemittelt fünf Prozent beträgt.

```go
package main

import (
  "net"
  "fmt"
  "time"
)

var INTERVAL = 5 * time.Second

func refreshDNS(domain string) {
  for {
    ips, _ := net.LookupIP(domain)

    for _, ip := range ips {
      fmt.Printf("IP: %s\n", ip.String())
    }

    time.Sleep(INTERVAL)
  }
}

func main() {
  go refreshDNS("example.cluster.internal")

  // ...
}

// Output:
// IP: 192.168.0.2
// IP: 192.168.0.3
// IP: 192.168.0.4
```

Abbildung 13: Der Quellcodeauszug zeigt die Auflösung aller IP Adressen für einen gegebenen Domain-Namen in Golang.

```
curl -s http://localhost:9399/metrics | grep cpu_total_total

# HELP com_hivemq_system_os_global_cpu_total_total Generated from \
# Dropwizard metric import \
# (metric=com.hivemq.system.os.global.cpu.total.total, type=e0.h)
# TYPE com_hivemq_system_os_global_cpu_total_total gauge
com_hivemq_system_os_global_cpu_total_total 5.0
```

Abbildung 14: Zeile 1 zeigt einen *curl* Aufruf in einem Terminal, der Prometheus Metriken von einem lokalen HiveMQ Broker abfragt. Nachfolgend wird die Ausgabe mit einem *grep* Befehl nach *cpu_total_total* gefiltert. Zeile 3 bis 7 zeigen die Ausgabe im Terminal.

Für die programmatische Auswertung von Prometheus Metriken werden Bibliotheken für verschiedene Programmiersprachen bereitgestellt, so auch für Golang. Das `expfmt` Paket dieser Bibliothek ist für das Parsen der Metriken zuständig [46]. Quellcodeauszug 15 zeigt ein Golang Programm, das periodisch und asynchron die Prometheus Metrik `com_hivemq_system_os_global_cpu_total_total` von einem HiveMQ Broker unter der Adresse `http://localhost:9399/metrics` abruft. Der Wert der Metrik kann alle Werte zwischen 0 (keine Auslastung) und 100 (volle Auslastung) annehmen. Er wird in der Variable `value` als Gleitkommazahl gespeichert.

Die CPU Auslastung eines Nodes kann durch diverse Faktoren zeitweise schwanken. Mögliche Ursachen sind zum Beispiel der Java-Garbage-Collector, eine kurzfristige Traffic Spitze oder ein Client-Takeover. Diese kurzfristigen Unterschiede der Auslastung dürfen nicht direkt zu einer Veränderung der Gewichtung des Nodes führen. Um den direkten Einfluss solcher Spitzen zu vermeiden, werden Werte über einen bestimmten Zeitraum gesammelt und gemittelt. Der Mittelwert ist die durchschnittliche Arbeitslast eines Nodes in Prozent für eine bestimmte Dauer.

Anschließend wird die durchschnittliche Arbeitslast in eine Gewichtung für Envoy konvertiert. Je größer der Wert der Gewichtung im Vergleich zu den Gewichtungen der anderen Nodes ist, desto mehr Clients werden mit diesem Node verbunden (siehe Kapitel 5.2.2). Demnach ist es nicht möglich, die durchschnittliche Arbeitslast direkt als Gewichtung zu benutzen. Es würde ein gegenteiliger Effekt entstehen.

Da Nodes mit größeren Gewichtungen mehr Clients erhalten, wird die freie CPU Auslastung als Gewichtung der Nodes gewählt. Die freie Auslastung ergibt sich aus der Differenz von maximaler Auslastung (100) und durchschnittlicher Auslastung. Wenn man die freie Auslastung als Gewichtung der Nodes wählt, verteilen sich neue Clients bei einem Cluster aus Tabelle 6 wie folgt auf die Nodes:

- **Node 1:**

$$13/(13 + 70 + 95 + 78) \cdot 100 \approx 5,1\% \tag{1}$$

- **Node 2:**

$$70/(13 + 70 + 95 + 78) \cdot 100 \approx 27,3\% \tag{2}$$

- **Node 3:**

$$95/(13 + 70 + 95 + 78) \cdot 100 \approx 37,1\% \tag{3}$$

- **Node 4:**

$$78/(13 + 70 + 95 + 78) \cdot 100 \approx 30,5\% \tag{4}$$

```go
package main

import (
        "github.com/prometheus/common/expfmt"
        "net/http"
        "time"
        "fmt"
)

var INTERVAL = 5 * time.Second

func fetchMetrics(url string) {
        for {
                resp, _ := http.Get(url)
                defer resp.Body.Close()

                var parser expfmt.TextParser
                mf, _ := parser.TextToMetricFamilies(resp.Body)

                metric := "com_hivemq_system_os_global_cpu_total_total"
                value := mf[metric].GetMetric()[0].Gauge.GetValue()

                fmt.Printf("CPU load: %.1f\n", value)

                time.Sleep(INTERVAL)
        }
}

func main() {
        go fetchMetrics("http://localhost:9399/metrics")

        // ...
}

// Output:
// CPU load: 5.0
// CPU load: 10.3
// CPU load: 7.5
```

Abbildung 15: Der Quellcodeauszug zeigt wie man in Golang Prometheus Metriken mit *expfmt* und *net/http* abfragen und parsen kann.

	Ø CPU Auslastung	**Ø freie CPU Kapazitäten**
Node 1	87 %	13 %
Node 2	30 %	70 %
Node 3	5 %	95 %
Node 4	22 %	78 %

Tabelle 6: CPU Auslastung und Kapazitäten von Nodes eines Clusters über einen beliebigen Zeitraum

Nodes mit einer höheren Auslastung, zum Beispiel Node 1, erhalten nun weniger neue Clients als Nodes mit einer geringeren Auslastung wie Node 3.

Um einen weighted CPU round-robin Load-Balancer zu validieren, wurden ebenfalls die Testszenarien aus Kapitel 5.2.1 durchgeführt. Tabelle 7 zeigt die Standardabweichung der einzelnen Testszenarien verglichen mit einem round-robin und least-connection load balancing Algorithmus. Nachdem sich alle MQTT Clients aus den Testszenarien mit dem Cluster verbunden haben, wurden die Werte der Standardabweichung gemessen. Abbildungen 16, 17 und 18 zeigen die zeitlichen Verläufe des weighted CPU Load-Balancers bei den Testszenarien eins, zwei und drei.

Insgesamt erzielte der weighted round-robin load balancing Algorithmus die besten Ergebnisse. Dieser weist eine leichte Optimierung gegenüber dem round-robin Algorithmus auf. Der least-connection Algorithmus hat die größte Standardabweichung der CPU Auslastungen und bestätigt, dass MQTT Verbindungen unterschiedliche Lastprofile aufweisen. Die unterschiedlichen Lastverteilungen zwischen den load balancing Algorithmen kommen durch die unterschiedlichen Testszenarien zustande. Diese wurden basierend auf den Schwachstellen von round-robin und least-connection in Bezug auf MQTT ausgelegt, um zu untersuchen, ob ein weighted round-robin load balancing Algorithmus basierend auf der CPU Auslastung generell eine bessere Lastverteilung bietet.

	round-robin	least-connection	weighted CPU
Testszenario 1	9	8	8
Testszenario 2	3	10	4
Testszenario 3	8	6	6
Gesamt	20	24	18

Tabelle 7: Die Tabelle zeigt die Standardabweichungen der Testszenarien aus Kapitel 5.2.1 und des weighted round-robin load balancing Algorithmus.

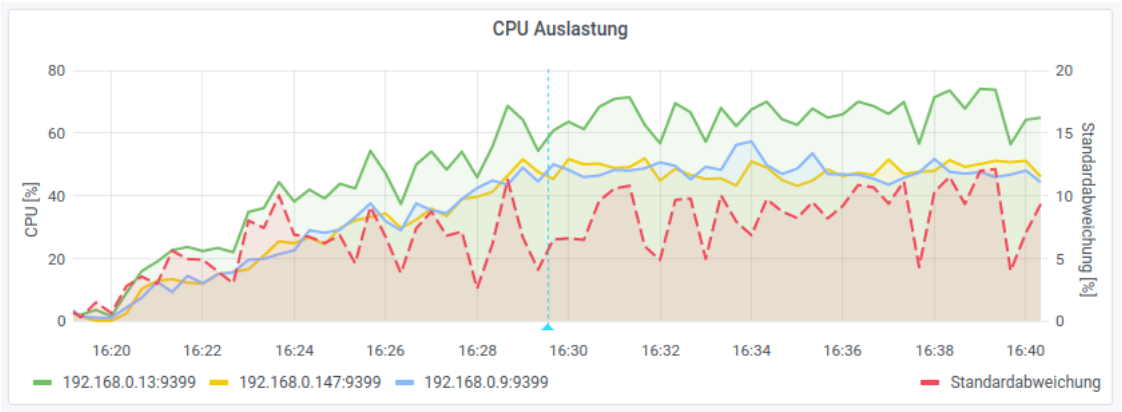

Abbildung 16: Testszenario 1 - weighted CPU round-robin Load-Balancer

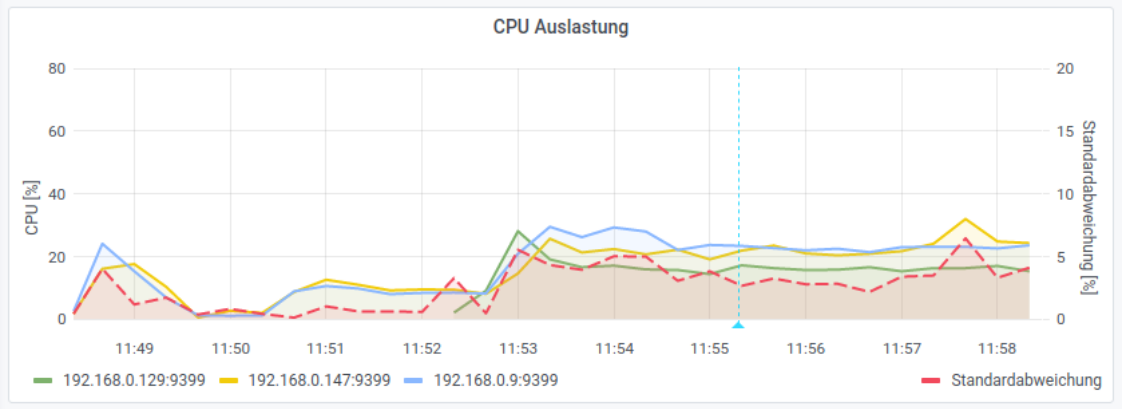

Abbildung 17: Testszenario 2 - weighted CPU round-robin Load-Balancer

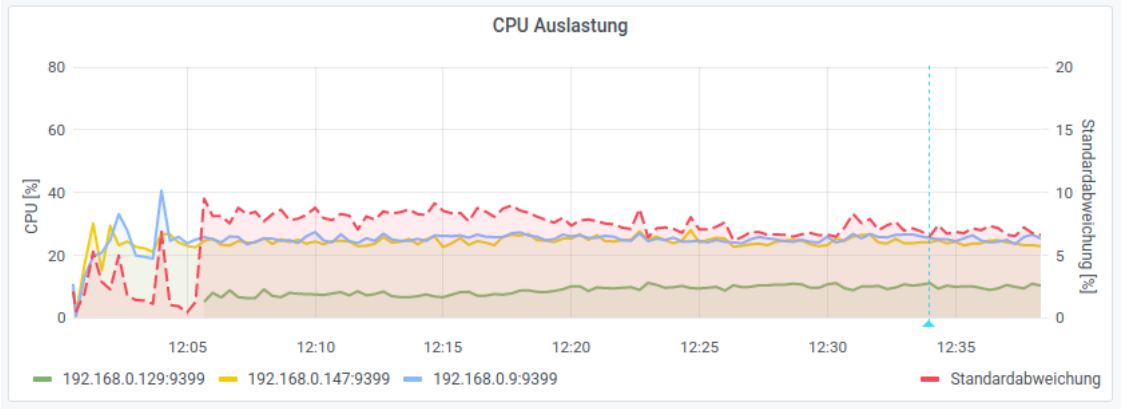

Abbildung 18: Testszenario 3 - weighted CPU round-robin Load-Balancer

5.2.6 Überlastschutz

Wie in Kapitel 5.2.5 beschrieben, kann das Lastverhalten von MQTT Clients während der Laufzeit stark variieren. Da ein Client zur Laufzeit nicht mit einem anderen Node verbunden werden kann, führt es bei dem Verändern des Lastverhaltens von Clients zu einer Überlastsituation eines Nodes. Ein HiveMQ Broker hat mehrere Möglichkeiten, sich vor einer solchen Überlastsituation zu schützen (siehe Kapitel 3.2.1.1). Im ersten Schritt wendet der Broker TCP *Back pressure* auf individuelle Clientverbindungen an, um den Paketfluss zu verzögern. Führt diese Mechanik nicht zur gewünschten Kontrolle der Lastspitze, wird die Verbindung zu allen Clients unterbrochen, die ihre Credits aufgebraucht haben. Dadurch werden gezielt Clients vom Broker getrennt, die zu viel Arbeitslast für den Broker verursachen. Durch die Trennung dieser Clients gibt es keine Serviceeinschränkungen für alle anderen Clients, die mit dem Broker verbunden sind. Sonst käme es zu Paketverzögerungen oder Paketverlusten auf einem Broker.

Wenn sich ein Node eines Clusters in einer Überlastsituation befindet, ist es wichtig, dass der Load-Balancer keine neuen Clients mehr mit diesem Node verbindet. Über die Prometheus Schnittstelle der HiveMQ Nodes werden diverse Metriken über den Zustand eines Nodes abgefragt. Eine Liste aller Prometheus Metriken findet sich in der HiveMQ Dokumentation [47]. Tabelle 8 gibt eine Übersicht der Metriken, die Indikatoren für eine mögliche Überlastsituation sind. Metriken, die sich auf Clients beziehen, enthalten nur Daten von Clients, die mit dem Node verbunden sind und bei dem die Metriken abgefragt wurden.

Metrik	**Beschreibung**
`com.hivemq.supervision.overload.protection.level`	Aktueller *Overload-Protection Level* zwischen 1 und 10.
`com.hivemq.overload-protection.credits.per-tick`	Anzahl der Credits, die ein Client alle 200 Millisekunden regeneriert.
`com.hivemq.overload-protection.clients.average-credits`	Durchschnittliche Anzahl der Credits aller Clients.
`com.hivemq.overload-protection.clients.backpressure-active`	Anzahl der Clients, die durch eine Overload-Protection TCP Back pressure haben.

Tabelle 8: Die Tabelle zeigt HiveMQ Metriken, die in einer Beziehung zu dem Overload-Protection Level stehen.

Die Metrik `com.hivemq.supervision.overload.protection.level` aggregiert intern mehrere Metriken zusammen und bestimmt einen allgemeinen Überlastlevel des Nodes. Der Wert des Levels liegt zwischen null und maximal zehn. Um ein maximales Überlastlevel zu vermeiden, kann der Load-Balancer präventiv keine neuen Clients mehr mit einem

Node verbinden, der einen bestimmten Schwellwert als Level überschritten hat.

Ein hohes Überlastlevel kann kurzfristig durch Events wie das Beitreten eines neuen Nodes in ein Cluster auftreten. Dabei werden einige Client-Queues auf den neuen Node repliziert, was zeitweise die Nodes, auf denen die Queues liegen, stark beansprucht. Es ist wichtig, dass der LB auf solche Events schnell reagiert und keine neuen Clients mehr mit den betroffenen Nodes verbindet. Um dies zu gewährleisten, wird eine geringe Abtastfrequenz der Prometheus Metriken gewählt.

Eine in Abbildung 19 gezeigte Überlastsituation verursacht einen maximalen Überlastlevel von zehn. Dieser wird zu Beginn durch das Beitreten eines dritten Nodes und einem Verbinden von 800 Clients ausgelöst. Um ein HiveMQ Cluster in dieser Situation zu unterstützen, kann ein Schwellenwert für den Überlastschutz auf vier gesetzt werden, damit sich keine neuen Clients mehr mit einem in Überlastsituation befindenden Node verbinden. In Kombination mit der Möglichkeit, TCP Back pressure bei Clients anzuwenden, kann HiveMQ eine solche Überlastsituation kontrollieren, ohne die Verbindungen der Clients zu unterbrechen. Diese Methode wird auch als *load-shedding* bezeichnet. Dabei reduziert sich der angebotene Service für einige Clients, um das Gesamtsystem aufrechtzuerhalten.

Um Envoy zu instruieren, keine neuen Clients mit einem sich in einer Überlastsituation befindenden HiveMQ Node zu verbinden, wird der *Health-Status* des Nodes entsprechend gesetzt. Der Wert ist entweder `HEALTHY` oder `UNHEALTHY`. Dieser Zustand kann über die Control-Plane für jeden Node individuell gesetzt werden. [48] Die Funktionalität des Health-Status eines Nodes wird in Kapitel 5.2.7 detailliert betrachtet.

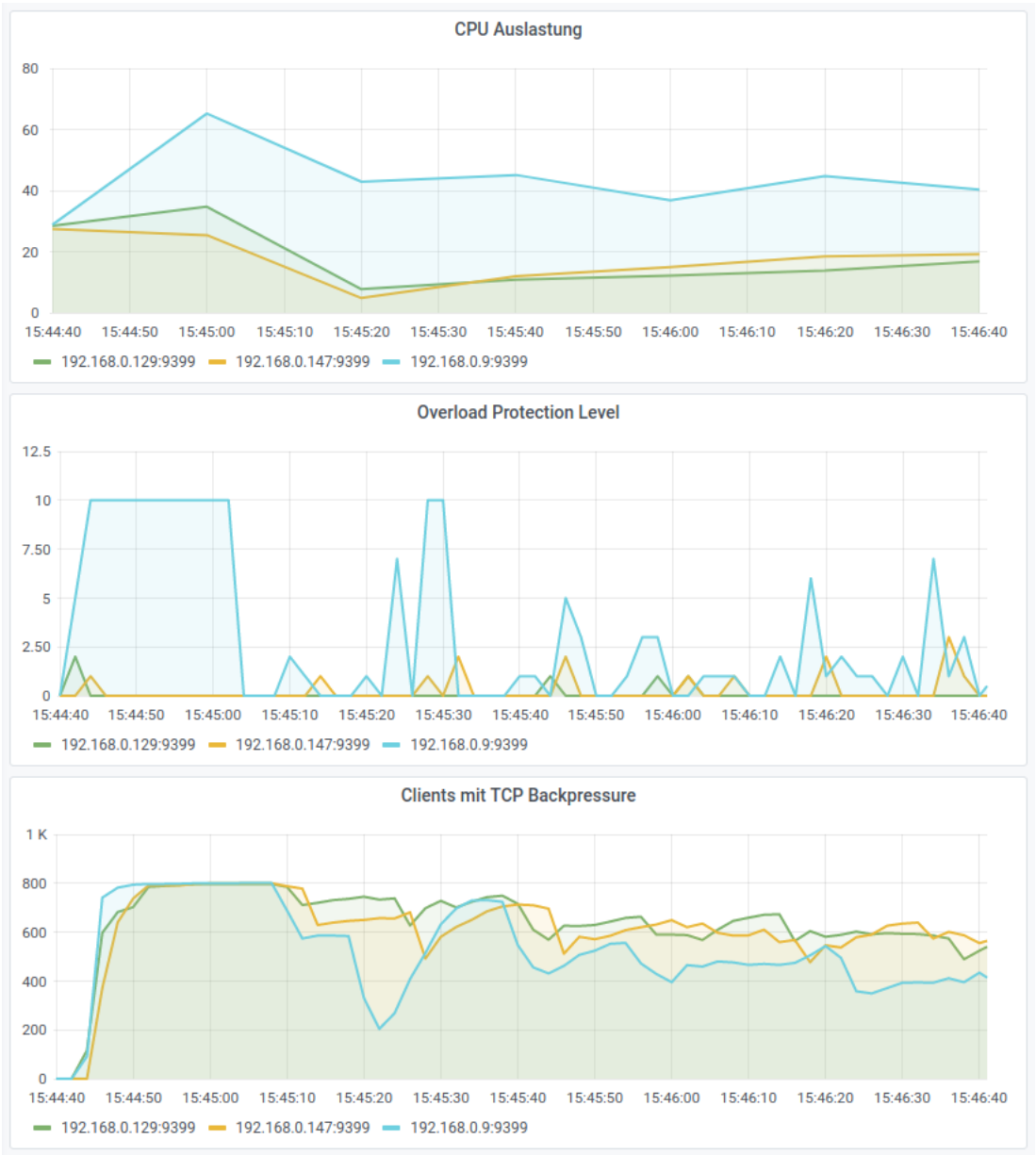

Abbildung 19: Die Abbildung zeigt Prometheus Metriken, die mit Grafana dargestellt werden, von einem HiveMQ Cluster bestehend aus drei Nodes. Bei den MQTT Clients, die mit dem Cluster verbunden sind, wird von dem Broker TCP backpressure angewendet, da diese zu viele Nachrichten veröffentlichen.

5.2.7 Health-Check

Nodes eines Clusters in Envoy haben einen Gesundheitszustand (engl. *Health-Status*) der angibt, ob ein Node neue Clientverbindungen erhalten kann. Der Health-Status eines Nodes ist anwendungsspezifisch und kann unter anderem durch *health-checks* bestimmt werden. HTTP-Services implementieren für diesen Zweck beispielsweise eine `/health` Route, die der Load-Balancer periodisch abrufen kann. Wenn der HTTP-Status-Code der Antwort 200 beträgt, ist der Service *healthy* und bereit für neue Verbindungen.

Um zu überprüfen, ob ein HiveMQ Broker bereit ist, neue Clientverbindungen MQTT konform anzunehmen, muss sich die Control-Plane als MQTT Client bei dem Broker anmelden. Bei einer erfolgreichen Verbindung wird die MQTT Verbindung MQTT konform beendet und der Node als `HEALTHY` markiert. Um einen ausführlicheren Health-Check zu implementieren, können noch weitere MQTT Funktionen wie Publish und Subscribe getestet werden. Dazu abonniert die Control-Plane ein Topic, das nicht von anderen Clients verwendet wird. Nach dem Abonnieren veröffentlicht die Control-Plane eine beliebige Nachricht auf dem zuvor abonnierten Topic und überprüft, ob diese Nachricht empfangen wird. Abbildung 20 zeigt die erforderlichen MQTT Pakete und den Ablauf des beschriebenen Health-Checks in einem UML Ablaufdiagramm. Wenn keine Fehler während des Health-Checks auftreten, wird der Node als `HEALTHY` markiert.

Um Clients möglichst immer mit einem funktionierenden HiveMQ Node eines Clusters zu verbinden, muss der Health-Check periodisch von der Control-Plane ausgeführt werden. Zudem kann ein Zeitfenster definiert werden, in dem ein Health-Check erfolgreich durchgelaufen sein muss. Falls der Health-Check länger als das Zeitfenster benötigt, gilt der Node als `UNHEALTHY`. Dadurch wird neben der Funktionalität des Brokers auch eine gegebene Antwortzeit gewährleistet.

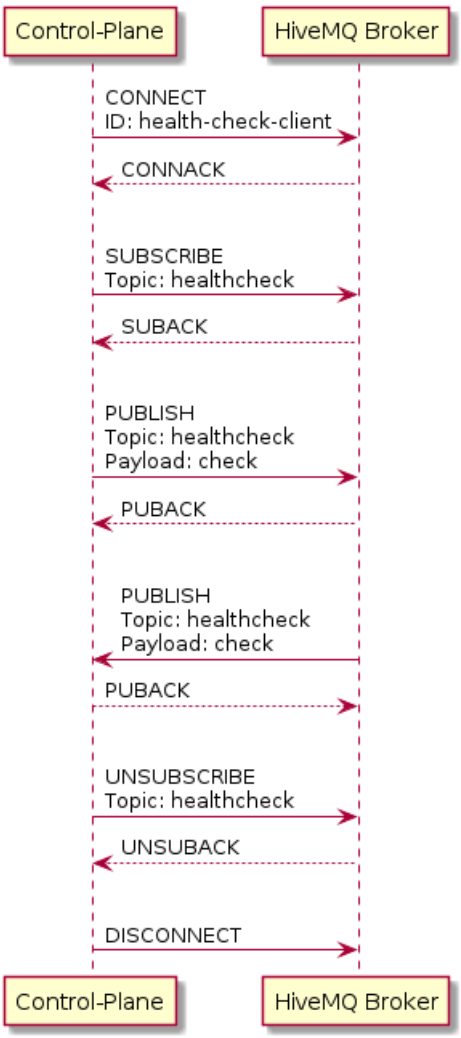

Abbildung 20: Die Grafik zeigt ein UML Ablaufdiagramm des in Kapitel 5.2.7 erläuterten MQTT Health-Checks.

5.2.8 Aktualisierung der Konfiguration

Um Aktualisierungen wie Health-Status, Gewichtung oder Anzahl der Nodes an eine Envoy Instanz zu schicken, werden versionierte Momentaufnahmen (engl. *Snapshots*) von einem Zustand der Konfiguration erstellt. Wenn die Version des Snapshots auf dem Envoy eine andere Version ist als die Version des aktuellen Snapshots auf der Control-Plane, liefert die Control-Plane den aktuellen Snapshot an die Envoy Instanz aus.

Um Konfigurationen optimal in einem Cache-Speicher abzulegen, hat eine einzigartige Konfiguration immer dieselbe Version. Dazu kann die Clusterkonfiguration in eine Hash-Funktion gegeben werden, um als Resultat einen String mit fixer Länge als Rückgabewert zu erhalten. Die Hash-Funktion garantiert einen immer gleichen Rückgabewert bei gleichem Input. Dementsprechend wird bei gleichbleibender Konfiguration immer die gleiche Version erzeugt. Als Eingabe der Hash-Funktion dienen alle Informationen, bei deren Änderung eine neue Version erstellt werden soll. Dies ist eine Liste aller HiveMQ Nodes mit folgenden Werten:

- IP-Adresse
- Port
- Gewichtung
- Health-Status

Die Reihenfolge, in der die Nodes in der Liste für die Hash-Funktion vorkommen, ist von hoher Relevanz. Hätten alle Nodes dieselben Werte, sich aber die Reihenfolge der Liste ändert, würde die Hash-Funktion einen neuen Wert zurückgeben. Daher muss sichergestellt sein, dass die Nodes immer in der gleichen Reihenfolge vorkommen. Um die Liste der Nodes zu sortieren, werden einzigartige Eigenschaften der Nodes verglichen. Es können niemals zwei HiveMQ Nodes dieselbe IP-Adresse und denselben Port haben. Beide Werte als String vereint können verglichen werden, um bei gleichbleibenden Nodes dieselbe Reihenfolge in einer Liste einzuhalten.

Die Funktion `generateVersion` aus Quellcodeauszug 21 generiert die Version für einen Snapshot. Es wird eine HiveMQ Node Liste übergeben und ein String zurückgegeben, der als Version des Snapshots dient. Nach der Sortierung der HiveMQ Node Liste nach Identifier, der aus Hostnamen und Port besteht, werden die Werte Identifier, Gewichtung und Health-Status in eine Hash-Funktion gegeben. Das Ergebnis der Hash-Funktion ist der Rückgabewert der Funktion `generateVersion`.

```go
type HiveMQ struct {
        ID          string
        HostName    string
        Port        uint32
        Weight      uint32
        Healthy     bool
}

func NewHiveMQ(host string, port uint32) *HiveMQ {
        o := new(HiveMQ)

        o.HostName = host
        o.Port = port
        o.MetricsPort = metricsPort

        // default values
        o.Weight = 1
        o.Healthy = false

        o.ID = fmt.Sprintf("%v:%v", host, port)

        return o
}

func (h *HiveMQ) GetHashString() string {
        return fmt.Sprintf("%v-%v-%v", h.ID, h.Weight, h.Healthy)
}

func generateVersion(endpoints []*HiveMQ) string {
        // sort array so nodes are always in the same order
        sort.Slice(endpoints, func(i, j int) bool {
                return endpoints[i].ID < endpoints[j].ID
        })

        s := sha256.New()
        for _, endpoint := range endpoints {
                s.Write([]byte(
                        fmt.Sprintf("%v", endpoint.GetHashString())
                ))
        }

        return fmt.Sprintf("%x", s.Sum(nil))
}
```

Abbildung 21: Der Quellcodeauszug zeigt die Generierung einer Envoy Snapshot Version basierend der HiveMQ Node Eigenschaften Hostname, Port, Gewichtung und Health-Status in Golang.

5.3 Sticky Session

Ein HiveMQ Broker speichert temporär Daten von MQTT Clients ab. Darunter sind Daten wie zum Beispiel Topic Aliasse oder Nachrichten, die aufgrund eines Verbindungsabbruchs nicht zugestellt werden konnten. Sobald sich ein nicht mehr verbundener Client wieder mit dem Broker verbindet, kann der Broker die vorbehaltenen Nachrichten zustellen. Dies verringert den Datenverlust in instabilen Netzwerken.

Wie in Kapitel 4.3 erläutert, werden clientspezifische Daten nur auf bestimmten HiveMQ Nodes eines Clusters gespeichert. Wenn die Verbindung eines Clients unterbrochen wird und beim Verbindungsaufbau mit einem anderen Node als zuvor verbunden wird, müssen die Daten des Clients im Cluster synchronisiert werden. Im Optimalfall verbindet sich der Client mit demselben Node wie zuvor, wodurch keine Synchronisation notwendig ist. Ein solches Verfahren nennt sich *Sticky-Session* oder *Session-Affinity*. Dabei werden einzigartige Identifier eines Clients im Load-Balancer extrahiert, und beispielsweise in einer HashMap mit dem Zielnode gespeichert. Sobald der Client eine erneute Verbindung aufbaut, kann der Load-Balancer in der HashMap den alten Node nachschlagen und den Client mit diesem Node verbinden. [49]

5.3.1 Clientkennung

Jeder MQTT Client identifiziert sich durch eine einzigartige Clientkennung. Dieser wird, wie in Kapitel 3.1.5 beschrieben, in dem MQTT `CONNECT` Paket beim Verbindungsaufbau vom Client an den Broker geschickt. Eine Clientkennung darf niemals doppelt vergeben werden. Sobald sich ein Client mit dem Broker verbindet, dessen Kennung bereits von einem anderen Client genutzt wird, unterbricht der Broker die Verbindung mit dem bereits verbundenen Client und führt einen Client-Takeover durch. Dieser Vorgang gewährleistet, dass jede Clientkennung nur einmal vergeben ist.

Die Clientkennung eignet sich durch ihre Einzigartigkeit als Identifier für eine Sticky-Session.

5.3.2 MQTT CONNECT

In Kapitel 5.3.1 wurde die Benutzung der MQTT Clientkennung als Merkmal für eine Sticky Session beschrieben. Envoy ist im Kern ein OSI-Layer vier Proxy und hat auf Informationen, wie eine Layer sieben MQTT Clientkennung, keinen Zugriff. Wie in Kapitel 3.3.2 erläutert, kann Envoy zum Beispiel für HTTP als Layer sieben LB betrieben werden, da das Dekodieren von HTTP-Paketen bereits in Envoy implementiert wurde. Das MQTT Protokoll ist nicht in Envoy integriert. Aus diesem Grund gibt es keine Möglichkeit in der Envoy Konfiguration eine load balancing Entscheidung anhand der Clientkennung zu treffen, da Envoy die einzelnen MQTT Pakete nicht dekodiert.

Envoy ermöglicht das Einbinden von WebAssembly-Modulen als Netzwerkfilter. Ein Netzwerkfilter könnte MQTT Pakete dekodieren, um diese den nachfolgenden Filtern in der Pipeline zur Verfügung zu stellen. Bislang gibt es WASM Software Development Kits (SDKs) für Rust, C++, AssemblyScript und Golang. [50] Beispiele für eine Implementa-

tion eines Golang Netzwerk Filters finden sich auf GitHub [51].

Das Dekodieren der MQTT Pakete in Golang wurde bereits von der Eclipse MQTT Bibliothek [52] implementiert. Der Quellcodeauszug 23 zeigt einen Auszug aus der Datei `packets/connect.go` des Repositories und dekodiert ein MQTT `CONNECT` Paket. Die Fehlerabhandlung wurde für eine bessere Lesbarkeit entfernt. Quellcodeauszug 22 zeigt die Struktur eines `CONNECT` Paketes basierend auf den in Kapiel 3.1.5 beschriebenen Feldern.

```
type FixedHeader struct {
        MessageType     byte
        Dup             bool
        Qos             byte
        Retain          bool
        RemainingLength int
}

type ConnectPacket struct {
        FixedHeader
        ProtocolName    string
        ProtocolVersion byte
        CleanSession    bool
        WillFlag        bool
        WillQos         byte
        WillRetain      bool
        UsernameFlag    bool
        PasswordFlag    bool
        ReservedBit     byte
        Keepalive       uint16

        ClientIdentifier string
        WillTopic        string
        WillMessage      []byte
        Username         string
        Password         []byte
}
```

Abbildung 22: Es wird eine Golang Struktur gezeigt, die alle Daten eines MQTT CONNECT Paketes beinhalten kann. Quelle: [52]

Bei jeder neuen Clientverbindung dekodiert der Load-Balancer das erste Paket des Clients mit der Funktion `Unpack` aus Quellcodeauszug 23, um an den Wert von `ConnectPacket.ClientIdentifier` zu gelangen. Die Clientkennung wird beispielsweise in einer HashMap gespeichert, um bei neuen Clientverbindungen zu prüfen, ob der Client bereits mit einem Node verbunden wurde.

```
func (c *ConnectPacket) Unpack(b io.Reader) {
        c.ProtocolName, _ = decodeString(b)
        c.ProtocolVersion, _ = decodeByte(b)
        options, _ := decodeByte(b)
        c.ReservedBit = 1 & options
        c.CleanSession = 1 & (options >> 1) > 0
        c.WillFlag = 1 & (options >> 2) > 0
        c.WillQos = 3 & (options >> 3)
        c.WillRetain = 1 & (options >> 5) > 0
        c.PasswordFlag = 1 & (options >> 6) > 0
        c.UsernameFlag = 1 & (options >> 7) > 0
        c.Keepalive, _ = decodeUint16(b)
        c.ClientIdentifier, _ = decodeString(b)
        if c.WillFlag {
                c.WillTopic, _ = decodeString(b)
                c.WillMessage, _ = decodeBytes(b)
        }
        if c.UsernameFlag {
                c.Username, _ = decodeString(b)
        }
        if c.PasswordFlag {
                c.Password, _ = decodeBytes(b)
        }
}
```

Abbildung 23: Der Quellcodeauszug zeigt eine Golang Funktion, die ein MQTT CONNECT Paket dekodiert. Quelle: [52]

5.3.3 Cluster Health

Bei dem in Kapitel 5.2.5 vorgestellten weighted round-robin load balancing Algorithmus ist es nicht möglich, einen individuellen Client mit einem bestimmten Node zu verbinden. Der Load-Balancer entscheidet, ob ein Client basierend auf dem weighted round-robin oder dem Sticky-Session Algorithmus aus Kapitel 5.3.1 mit einem Zielnode verbunden wird.

Eine Möglichkeit, beide Algorithmen in einem Load-Balancer zu verwenden, ist bei einem neuen Client zuerst in der HashMap nachzuschlagen, ob dieser Client bereits mit einem Node verbunden wurde. Falls dieser Node noch vorhanden und `HEALTHY` ist, wird der Client erneut mit dem Node verbunden. Andernfalls wird ein neuer Node basierend dem weighted round-robin Algorithmus bestimmt. Diese Methode harmoniert mit dem beschriebenen Überlastschutz aus Kapitel 5.2.6. Für den Fall, dass ein HiveMQ Node die Verbindung eines Clients aufgrund einer Überlastung unterbricht, markiert der Überlastschutz diesen Node als `UNHEALTHY` und der Load-Balancer wird den Client, der sich wieder mit dem Cluster verbindet, mit einem anderen Node verbinden.

Durch diese Implementation der Sticky Session werden Clients, die ihre Verbindung verlieren, immer mit demselben Node verbunden und Clients, deren Verbindung absichtlich beendet wurde, werden einem neuen Node zugewiesen. Clients, die sich zum ersten Mal mit dem Cluster verbinden, werden basierend auf der freien Arbeitslast verteilt.

6 Architektur und Implementierung

In Kapitel 5 wurden Konzepte für einen *klugen* MQTT Load-Balancer beschrieben. In diesem Kapitel wird gezeigt, wie man einige dieser Konzepte in einen Envoy Load-Balancer integriert. Im Rahmen der Thesis wurde die DNS Cluster-Discovery (5.2.4), das weighted CPU round-robin (5.2.5), der Überlastschutz (5.2.6) und der MQTT Health-Check (5.2.7) aus Kapitel 5 in einer Envoy Control-Plane implementiert.

6.1 Envoy Control-Plane

Die Envoy Control-Plane ist die *kluge* Komponente des Systems aus Kapitel 5. Obwohl die einzelnen Envoy Instanzen die load balancing Entscheidung treffen, liefert die Control-Plane alle Informationen, auf dessen Basis diese Entscheidung getroffen wird. Als Quellcodebasis für die Control-Plane wurde ein minimales Beispiel aus dem offiziellen Github Repository [53] verwendet. Dieses startet einen gRPC-Server, der eine Envoy Konfiguration erzeugt, die wiederum einen HTTP-Endpunkt definiert. Da die offizielle Referenzimplementierung einer Control-Plane in Golang programmiert ist, wurde die Control-Plane des Smart Load-Balancers, nachfolgend Control-Plane genannt, ebenfalls in Golang implementiert.

Die parallele Ausführung von mehreren Go-Routinen ist ein wichtiger Bestandteil der Control-Plane. Diese muss asynchron und nicht blockend Aufgaben abarbeiten, wie zum Beispiel das Überprüfen eines Health-Status der HiveMQ Nodes, und parallel den neusten Snapshot an alle Envoys ausliefern. Um bei einem asynchronen Programm *Data-Races* zu vermeiden, muss darauf geachtet werden, dass immer nur eine Go-Routine Schreibzugriff auf eine Variable erhält und lesende Go-Routinen sich zu Beginn eine Kopie der Variable erstellen, damit diese nicht während der Bearbeitung von der schreibenden Go-Routine verändert wird.

Die Control-Plane besteht aus den in Abbildung 24 dargestellten Komponenten. Komponenten des Paketes `github.com/breuerfelix/mqtt-control-plane` wurden im Rahmen der Thesis programmiert.

- **Main:** Die Main-Komponente ist der Einstiegspunkt des Programms und initialisiert die Metrics-, Resources- und Cluster-Komponente. Nach der Initialisierung werden alle nötigen Go-Routinen aufgerufen und anschließend ein gRPC-Server gestartet, der eingehende Envoy Verbindungen akzeptiert, um die aktuelle Konfiguration eines Cache-Speichers auszuliefern.
- **Resources:** Generiert Envoy Ressourcen basierend auf einer Cluster-Struktur. Es wird ein TCP-Listener und ein Upstream Envoy Cluster mit allen Endpunkten und Gewichtungen erzeugt. Die Envoy Ressourcen werden als Snapshot zusammengefasst und können in einem Cache-Speicher hinterlegt werden.

- **Metrics:** Wertet periodisch alle Metriken der HiveMQ Nodes einer Cluster-Struktur aus. Dabei werden Health-Status und Gewichtung entsprechend angepasst.
- **Cluster:** Ein Cluster repräsentiert ein HiveMQ Cluster und dient als Container für eine Liste von allen HiveMQ Node-Strukturen. Ein Cluster enthält einen Domainnamen, unter dem die DNS Discovery die einzelnen Nodes identifizieren kann.
- **Node:** Node ist eine Struktur, die alle Daten eines HiveMQ Nodes enthält. Darunter befinden sich IP-Adresse, Port, Health-Status, Weight und nach Zeitstempel sortierte Metriken für einen bestimmten Zeitraum.

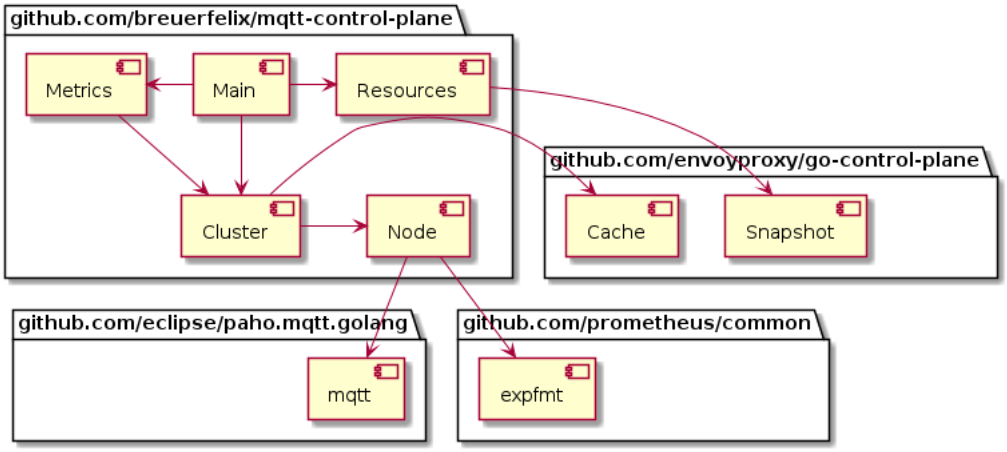

Abbildung 24: Die Abbildung zeigt das UML-Komponentendiagramm der entwickelten Control-Plane. Die Komponenten des Paketes *github.com/breuerfelix/mqtt-control-plane* wurden im Rahmen der vorliegenden Thesis programmiert.

Abbildung 25 zeigt die Cluster- und Node-Strukturen in einem UML-Klassendiagramm.

- **Cluster:** Die Cluster-Struktur enthält alle erforderlichen Daten, um die Envoy Konfiguration für ein HiveMQ Cluster zu generieren. Die Funktion `CheckNodes` löst die IP-Adressen der Domain des Feldes `Domain` auf. Anhand der erhaltenen IP-Adressen wird die Liste aller Node-Strukturen des Clusters erstellt. Die Funktion `UpdateSnapshot` benutzt die Resources-Komponente, um einen Snapshot des aktuellen Clusters zu bilden und in den Cache-Speicher des `Cache` Feldes zu speichern.
- **Node:** Eine Node-Struktur repräsentiert einen HiveMQ Node. Die Funktion `FetchMetrics` ruft die aktuellen Prometheus Metriken des Nodes ab und speichert die geparsten Daten in dem `Metrics` Feld der Struktur ab. Die Funktion `filterOldData` löscht alte gespeicherte Metriken, die nicht mehr verarbeitet werden. `CheckHealth` benutzt die MQTT Komponente der Eclipse Bibliothek, um einen Health-Check des Nodes durchzuführen (siehe Kapitel 6.2.3).

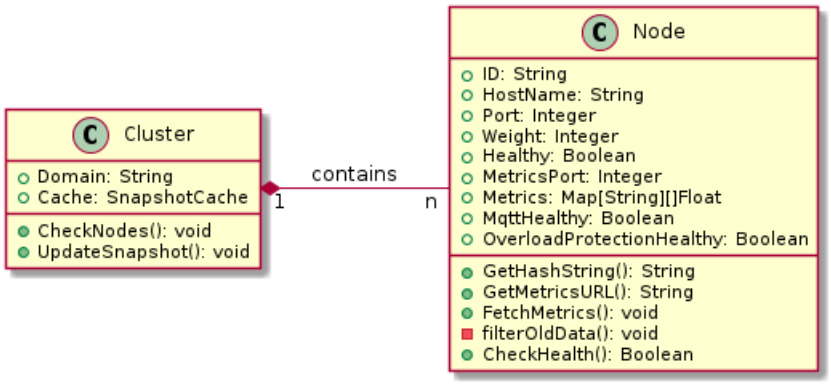

Abbildung 25: Die Abbildung zeigt das UML-Klassendiagramm der entwickelten Control-Plane.

6.2 Funktionalität

6.2.1 DNS Cluster-Discovery

Die DNS Cluster-Discovery aus Kapitel 5.2.4 ist in der Cluster-Komponente implementiert. Beim Start der Control-Plane wird von der Main-Komponente eine Go-Routine gestartet, die periodisch und asynchron in einem gegebenen Intervall die Funktion `CheckNodes` der Cluster-Struktur aufruft. Diese verwendet die `net` Standard Bibliothek von Golang, um den Domain-Namen des Clusters aufzulösen und für alle IP-Adressen eine Node-Struktur zu erstellen. Wenn bereits eine Node-Struktur für ein IP-Adressen und Port Paar vorhanden ist, wird die Referenz der Struktur in die neue Liste übernommen.

6.2.2 Weighted Round-Robin

Der weighted round-robin Algorithmus basierend auf der CPU Auslastung der einzelnen HiveMQ Nodes aus Kapitel 5.2.5 ist in der Metrics-Komponente implementiert.

Quellcodeauszug 26 zeigt den Algorithmus für die Bestimmung der Gewichtung der HiveMQ Nodes anhand der CPU Auslastung. Zeile 8 bis 17 berechnen die durchschnittliche CPU Auslastung aller gespeicherten Metriken individuell für jeden Node. Abgefragte Metriken werden nur für einen bestimmten Zeitraum gespeichert (siehe Kapitel 6.1). Durch die Bildung des Durchschnittswertes über einen bestimmten Zeitraum, wird die Regelung der Gewichtung träge. Somit wird die Auswirkung von kurzfristigen Lastspitzen auf die Gewichtung reduziert. Zeile 19 bis 34 bestimmen die Gewichtung der Nodes basierend der zuvor berechneten Durchschnittsauslastung. Die Metrik der CPU Auslastung kann einen Maximalwert von 100 erreichen. Da die freie CPU Auslastung für die Gewichtung verwendet wird, wird die Durchschnittsauslastung von der maximalen Auslastung abgezogen, um die freie Auslastung zu erhalten. Bevor das Ergebnis als Gewichtung gesetzt wird, muss überprüft werden, ob dieses `NaN` oder kleiner als eins ist, da dies die kleinste erlaubte Gewichtung in Envoy ist.

```
// init data container
processed := make(map[string]map[string]float64)
for _, broker := range endpoints {
  processed[broker.ID] = make(map[string]float64)
}

// process average values over all stored metrics
for _, broker := range endpoints {
  for metric, data := range broker.Metrics {
    mean := 0.0
    for _, d := range data {
      mean += d.Value
    }

    processed[broker.ID][metric] = mean / float64(len(data))
  }
}

for _, broker := range endpoints {
  // calculate weight based on free cpu capacity
  val := 100.0 - processed[broker.ID][Metrics.CpuLoad]

  // check for NaN
  if val != val || val < 1.0 {
    val = 1.0
  }

  // set the actual weight to the node
  broker.Weight = uint32(math.Round(val))
  log.Printf(
    "Broker: %v / free CPU: %v / Weight: %v / Healthy: %v",
    broker.ID, val, broker.Weight, broker.Healthy,
  )
}

// update snapshot
```

Abbildung 26: Der Quellcodeauszug zeigt einen Algorithmus in Golang um die Gewichtungen von HiveMQ Nodes basierend ihrer CPU Auslastungen zu bestimmen.

6.2.3 Health-Check

Der MQTT Health-Check aus Kapitel 5.2.7 ist in der Node-Komponente implementiert. Die Funktion `CheckHealth` wird periodisch für jeden HiveMQ Node in einer Go-Routine der Main-Komponente aufgerufen. Wenn die `CheckHealth` Methode `false` zurückgibt, ist der HiveMQ Node nicht in der Lage neue Clientverbindungen zu erhalten und sein Health-Status wird auf `UNHEALTHY` gesetzt. Es wird außerdem sofort ein neuer Snapshot in den Cache geschrieben, damit so wenig Clients wie möglich mit einem `UNHEALTHY` HiveMQ Node verbunden werden. Der MQTT Health-Status wird außerdem in die Variable `MqttHealthy` der Node-Struktur gespeichert, damit diese von anderen Komponenten verarbeitet werden kann. Für den Fall, dass mehrere Faktoren den Health-Status eines Nodes bestimmen, müssen alle Faktoren den Status `HEALTHY` aufweisen, damit ein Node derart gekennzeichnet wird.

Eclipse stellt eine Open Source MQTT Bibliothek für Golang bereit. [52] Die Bibliothek bietet eine Abstraktion für das MQTT Protokoll und übernimmt das De- und Enkodieren von MQTT Paketen. Die Funktion `CheckHealth` wurde mit der Bibliothek umgesetzt und verhält sich wie das in Abbildung 20 dargestellte Ablaufdiagramm.

6.2.4 Überlastschutz

Der Überlastschutz aus Kapitel 5.2.6 ist in der Metrics-Komponente implementiert. Quellcodeauszug 27 zeigt den Algorithmus, um den Health-Status aller Nodes basierend der Metrik `OverloadProtection` zu bestimmen. In Zeile 10 wird auf den zuletzt bekannten Wert der Metrik zugegriffen und in Zeile 12 wird die Varible `OverloadProtectionHealthy` der Node-Struktur auf `true` gesetzt, falls der Wert der Metrik kleiner oder gleich fünf ist. Sobald der Overload-Protection Wert überschritten wurde, oder der MQTT Health-Check aus Kapitel 6.2.3 fehlgeschlagen ist, darf der HiveMQ Node nicht mehr als `HEALTHY` gekennzeichnet werden. Zeile 18 bis 22 setzen den Health-Status erst auf `HEALTHY` sobald beide Bedingungen erfüllt sind.

```
// calculations based on metrics
for _, broker := range endpoints {
  // calculate health status based on overload protection
  metrics := broker.Metrics[Metrics.OverloadProtection]
  if len(metrics) < 1 {
    continue
  }

  // get the latest value
  overload := metrics[len(metrics)-1]
  // if value is <= 5, set to healthy
  broker.OverloadProtectionHealthy = overload.Value <= 5.0
}

// determine health status of nodes
for _, broker := range endpoints {
  // only set to healthy if all healh conditions are met
  if broker.MqttHealthy &&
    broker.OverloadProtectionHealthy {
    broker.Healthy = true
    continue
  }

  broker.Healthy = false
}

// update snapshot
```

Abbildung 27: Der Quellcodeauszug zeigt einen Algorithmus in Golang, der den Health-Status eines HiveMQ Nodes basierend der Overload-Protection Prometheus Metrik bestimmt.

6.3 Einsatz

In Kapitel 5.2.3 wurde der Einsatz des Envoy Load-Balancer in einem MQTT Anwendungsfeld bereits eingeordnet. Abbildung 28 zeigt den Aufbau einer HiveMQ Cluster Installation mit einem Envoy als Load-Balancer und einer Control-Plane.

MQTT Clients bauen ihre Verbindung mit der Envoy Instanz auf, die das HiveMQ Cluster transparent zum Client abstrahiert. Die Envoy Instanz erhält seine Konfiguration von der Control-Plane und entscheidet auf dessen Grundlagen, mit welchem HiveMQ Node ein neuer Client verbunden wird. Die Control-Plane ruft periodisch die aktuellen Metriken aller HiveMQ Nodes des Clusters ab. Anhand der Metriken werden Health-Status und Gewichtung der einzelnen Nodes bestimmt.

Bei der in Abbildung 28 gezeigten Architektur müssen weder HiveMQ Nodes, noch Control-Plane, für die MQTT Clients erreichbar sein. Control-Plane und Envoy müssen Netzwerkverbindungen mit allen Nodes des HiveMQ Clusters aufbauen können.

Es können sich mehrere Envoy Instanzen bei einer Control-Plane registrieren. Um die einzelnen Envoy Instanzen für die MQTT Clients zu abstrahieren, wird ihnen ein Cloud Load-Balancer vorgeschaltet. Da jeder Envoy seine Konfiguration von der Control-Plane erhält, wird in jeder Envoy Instanz dieselbe load balancing Entscheidung getroffen. Eine solche Installation ist, wie ein HiveMQ Cluster, horizontal skalierbar.

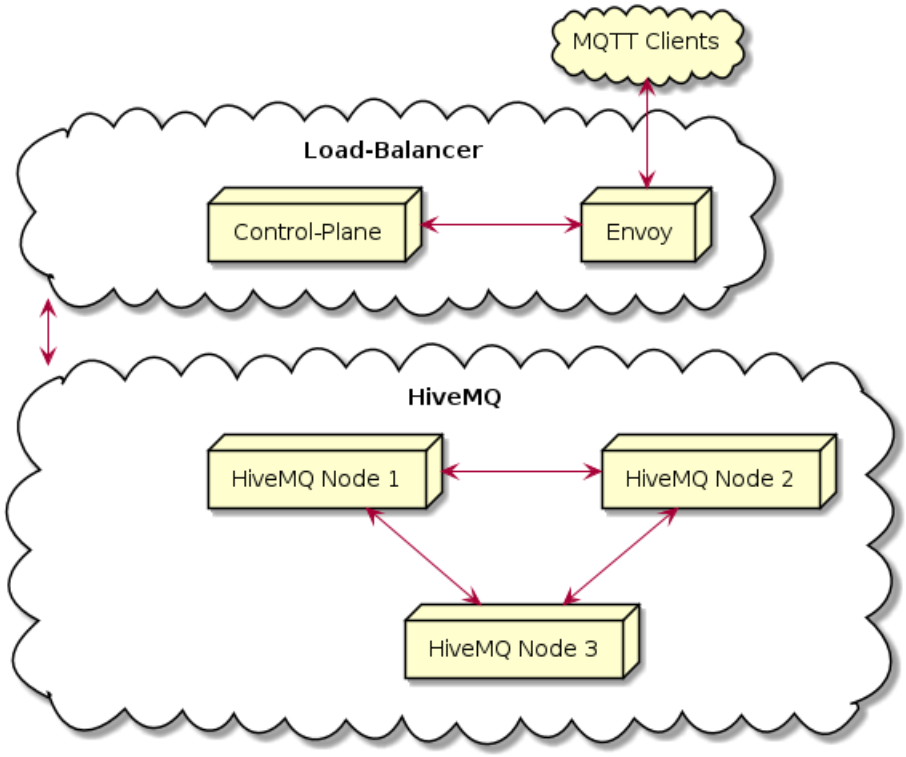

Abbildung 28: Die Abbildung zeigt das UML-Einsatzdiagramm der in der vorliegenden Arbeit entwickelten Control-Plane.

7 Fazit

Das Ziel der vorliegenden Arbeit war die Analyse des MQTT Protokolls mit Hinblick auf das *kluge* Verteilen der Clients an HiveMQ Nodes und die Programmierung einer Envoy Control-Plane, die einige der analysierten Features implementiert.

Die Vorbereitung auf diese Analyse bedurfte einer umfangreichen Einarbeitung in das MQTT Protokoll, um die Eigenheiten und verschiedenen Verhaltensmuster der Clients zu untersuchen. Trotz der in den letzten Jahren stetig steigenden Verwendung des Protokolls, unter anderem bei großen Cloud Anbietern wie Amazon Web Services und Microsoft Azure, gibt es nur wenig veröffentlichte Informationen zu dem Betrieb von MQTT Load-Balancern. Auch die Unterstützung des MQTT Protokolls in Proxies wie Nginx, HAProxy oder Envoy steht noch nicht zur Verfügung. Bei einem Envoy Proxy ist es bereits möglich, Protokoll *sniffing* für Redis, MongoDB oder Postgres zu betreiben, um eine ausführliche Analyse des Traffics durchzuführen. Der Mangel an Ressourcen des Ökosystems *load balancing MQTT* könnte durch die Kommerzialisierung im Bereich Industrial Internet of Things (IIoT) zu begründen sein.

Um IoT-Plattformen zu optimieren, müssen typischerweise entkoppelte Load-Balancer ein integraler Bestandteil des IoT-Ökosystems werden. Sie müssen das Protokoll verstehen, Informationen der Plattform aggregieren und reaktive load balancing Entscheidungen treffen. Nur so kann der Traffic von Millionen IoT-Geräten optimal verteilt werden. Der Load-Balancer muss zu einem *Smart-Device* im *Internet of Things* werden.

8 Ausblick

Die entwickelte Control-Plane macht den Load-Balancer nicht nur konfigurierbar, sondern auch programmierbar. Dies ermöglicht eine optimale Integration des Load-Balancers durch progressive Weiterentwicklung. Während der Bearbeitung der vorliegenden Arbeit wurde das Testwerkzeug HiveMQ Swarm [54] von HiveMQ veröffentlicht. Mit diesem Tool können Client-Verhaltensmuster definiert werden, um eine HiveMQ Cluster Installation zu testen. Mit HiveMQ Swarm kann eine Control-Plane iterativ erweitert und für den eigenen Use-Case optimiert sowie getestet werden.

Mit dem Erweitern von Envoy durch WebAssembly-Module kann zukünftig das gesamte MQTT Protokoll geparst werden. Dies eröffnet neben einer Realisierung der beschriebenen *Session-Affinity* die Möglichkeit, noch weitere protokollspezifische load balancing Entscheidungen zu treffen. Mit der fünften Version des MQTT Protokolls wurden *user properties* eingeführt, die es Benutzer:innen erlauben, beliebige Daten als *key-value* Paare an MQTT Pakete anzuhängen [55]. Ein protokollbewusster Load-Balancer kann mit dem Protokoll *sniffing* zu einem detaillierten Auswerten des MQTT Traffics beitragen und über *user properties* ein- sowie ausgehende Nachrichten mit nützlichen Informationen anreichern.

Denkbar sind Erweiterungen, wie ein neues Protokoll für die Kommunikation zwischen Load-Balancer und HiveMQ Cluster. Der Load-Balancer bietet eine MQTT konforme Schnittstelle für die Clients und leitet die einzelnen Pakete optimiert an die Nodes des HiveMQ Clusters weiter. Dabei können beispielsweise `PUBLISH` Pakete, unter Berücksichtigung der Topic Aliasse der Clients, verbindungsunabhängig an die Nodes verteilt werden. Die Konzepte der vorliegenden Arbeit bieten einen Einstieg, um den Nutzen derartig fundamentaler Erweiterungen für das load balancing von MQTT Traffic zu erforschen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Brief History of the Internet. Internet Society. [Online]. Available: https://www.internetsociety.org/internet/history-internet/brief-history-internet/

[2] R. van Kranenburg, *The Internet of Things: A Critique of Ambient Technology and the All-Seeing Network of RFID*. Institute of Network Cultures.

[3] D. Uckelmann, M. Harrison, and F. Michahelles, Eds., *Architecting the Internet of Things*. Springer.

[4] M. Hung, "Gartner Insights on How to Lead in a Connected World."

[5] Global IoT and non-IoT connections 2010-2025. Statista. [Online]. Available: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/

[6] S. Ranger. What is the IoT? Everything you need to know about the Internet of Things right now. ZDNet. [Online]. Available: https://www.zdnet.com/article/what-is-the-internet-of-things-everything-you-need-to-know-about-the-iot-right-now/

[7] What is MQTT? Definition and Details. [Online]. Available: https://www.paessler.com/it-explained/mqtt

[8] T. H. Team. Getting Started with MQTT. [Online]. Available: https://www.hivemq.com/blog/how-to-get-started-with-mqtt/

[9] S. Tarkoma, *Publish/Subscribe Systems: Design and Principles*. Wiley.

[10] T. H. Team. Publish & Subscribe - MQTT Essentials: Part 2. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe

[11] ——. MQTT Topics & Best Practices - MQTT Essentials: Part 5. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices

[12] Mqtt v5.0 specification. OASIS. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[13] T. H. Team. Retained Messages - MQTT Essentials: Part 8. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-8-retained-messages

[14] D. Soni and A. Makwana, "A SURVEY ON MQTT: A PROTOCOL OF INTERNET OF THINGS(IOT)."

[15] F. Raschbichler. MQTT 5: How the new Shared Subscriptions feature works and why we love it. [Online]. Available: https://www.hivemq.com//blog/mqtt5-essentials-part7-shared-subscriptions/

[16] About the HiveMQ Company. [Online]. Available: https://www.hivemq.com/company/

[17] D. Obermaier. MQTT Sparkplug Essentials Part 1 - Introduction. [Online]. Available: https://www.hivemq.com/blog/mqtt-sparkplug-essentials-part-1-introduction

[18] HiveMQ Cluster :: HiveMQ Documentation. [Online]. Available: https://www.hivemq.com/docs/hivemq/4.5/user-guide/cluster.html

[19] How to Scale IT Infrastructure Definition. SDxCentral. [Online]. Available: https://www.sdxcentral.com/cloud/definitions/what-is-it-infrastructure-scaling/

[20] Cluster Overload Protection :: HiveMQ Documentation. [Online]. Available: https://www.hivemq.com/docs/hivemq/4.5/user-guide/overload-protection.html

[21] D. Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architecture (Internetworking with TCP/IP)*.

[22] T. H. Team. Persistent Session and Queuing Messages - MQTT Essentials: Part 7. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages

[23] Why is the keep-alive needed? [Online]. Available: https://groups.google.com/g/mqtt/c/zRqd8JbY4oM/m/XrMwlQ5TU0EJ

[24] T. H. Team. Keep Alive and Client Take-Over - MQTT Essentials Part 10. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-10-alive-client-take-over

[25] T. Bourke, *Server Load Balancing*, 1st ed. O'Reilly.

[26] Software Load Balancing. VMware. [Online]. Available: https://www.vmware.com/topics/glossary/content/software-load-balancing

[27] What is a Load Balancer? - Load Balancing Definition - Citrix. Citrix.com. [Online]. Available: https://www.citrix.com/solutions/app-delivery-and-security/load-balancing/what-is-load-balancing.html

[28] "Envoyproxy/envoy," Envoy Proxy - CNCF. [Online]. Available: https://github.com/envoyproxy/envoy

[29] Migrating Millions of Concurrent Websockets to Envoy. Slack Engineering. [Online]. Available: https://slack.engineering/migrating-millions-of-concurrent-websockets-to-envoy/

[30] Istio. Istio. [Online]. Available: https://istio.io/

[31] Traffic Director | Google Cloud. [Online]. Available: https://cloud.google.com/traffic-director

[32] What is Envoy envoy 1.18.0-dev-e4f839 documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy

[33] What is a Reverse Proxy vs. Load Balancer? NGINX. [Online]. Available: https://www.nginx.com/resources/glossary/reverse-proxy-vs-load-balancer/

[34] Service discovery envoy 1.18.0-dev-23a43b documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/service_discovery

[35] Allow configuration of discovery interval ů Issue #4 ů hivemq/hivemq-dns-cluster-discovery-extension. GitHub. [Online]. Available: https://github.com/hivemq/hivemq-dns-cluster-discovery-extension/issues/4

[36] Exponential back-off, general discovery interval configuration by sbaier1 ů Pull Request #5 ů hivemq/hivemq-dns-cluster-discovery-extension. GitHub. [Online]. Available: https://github.com/hivemq/hivemq-dns-cluster-discovery-extension/pull/5

[37] DNS for Services and Pods. Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

[38] HiveMQ Extension - DNS Discovery. [Online]. Available: https://www.hivemq.com/extension/dns-discovery-extension/

[39] A. Büchter and H.-W. Henn, *Elementare Stochastik: eine Einführung in die Mathematik der Daten und des Zufalls*, ser. Mathematik für das Lehramt. Springer.

[40] Supported load balancers envoy 1.18.0-dev-50e812 documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers

[41] Configuration: Dynamic from filesystem envoy 1.18.0-dev-2a701b documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/start/quick-start/configuration-dynamic-filesystem

[42] “Envoyproxy/data-plane-api,” Envoy Proxy - CNCF. [Online]. Available: https://github.com/envoyproxy/data-plane-api

[43] Configuration: Dynamic from control plane envoy 1.18.0-dev-2a701b documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/start/quick-start/configuration-dynamic-control-plane

[44] Dynamic configuration (control plane) envoy 1.18.0-dev-2a701b documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/start/sandboxes/dynamic-configuration-control-plane

[45] Prometheus. Metric types | Prometheus. [Online]. Available: https://prometheus.io/docs/concepts/metric_types/#gauge

[46] Expfmt ů pkg.go.dev. [Online]. Available: https://pkg.go.dev/github.com/prometheus/common/expfmt

[47] Monitoring :: HiveMQ Documentation. [Online]. Available: https://www.hivemq.com/docs/hivemq/4.5/user-guide/monitoring.html

[48] Health check envoy 1.18.0-dev-25d506 documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/core/v3/health_check.proto#envoy-v3-api-enum-config-core-v3-healthstatus%C2%AC

[49] What does the term sticky session mean and how is it achieved? Red Hat Customer Portal. [Online]. Available: https://access.redhat.com/solutions/900933

[50] T. Sebastian. How to write WASM filters for Envoy and deploy it with Istio. [Online]. Available: https://banzaicloud.com/blog/envoy-wasm-filter/

[51] "Tetratelabs/proxy-wasm-go-sdk," Tetrate Labs. [Online]. Available: https://github.com/tetratelabs/proxy-wasm-go-sdk

[52] "Eclipse/paho.mqtt.golang," Eclipse Foundation. [Online]. Available: https://github.com/eclipse/paho.mqtt.golang

[53] Envoyproxy/go-control-plane. GitHub. [Online]. Available: https://github.com/envoyproxy/go-control-plane

[54] T. H. Team. HiveMQ Swarm - Find problems before production deployment. [Online]. Available: https://www.hivemq.com/hivemq-swarm

[55] F. Raschbichler. MQTT 5: How the new User Properties feature works and why we love it. [Online]. Available: https://www.hivemq.com//blog/mqtt5-essentials-part6-user-properties/

Abkürzungsverzeichnis

MQTT	Message Queuing Telemetry Transport
OASIS	Organization for the Advancement of Structured Information Standards
TCP	Transmission-Control-Protocol
IP	Internet-Protocol
IoT	Internet of Things
IIoT	Industrial Internet of Things
HTTP	Hypertext-Transfer-Protocol
QoS	Quality-of-Service
MSB	Most-Significant-Byte
LSB	Least-Significant-Byte
LB	Load-Balancer
CPU	Central-Processing-Unit
RAM	Random-Access-Memory
DNS	Domain-Name-System
API	Application-Programming-Interface
GB	Gigabyte
WASM	WebAssembly
SDK	Software Development Kit
GmbH	Gesellschaft mit beschränkter Haftung
HLB	Hardware-Load-Balancer
OSI	Open-System-Interconnection
UDP	User-Datagram-Protocol
TLS	Transport-Layer-Security
ITO	IT-Engineering & Operations
RPC	Remote-Procedure-Call

Eidesstattliche Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, den 07.05.2021

Unterschrift