QoS-Eigenschaften von MQTT

Fachbereich Informatik
Hochschule Darmstadt

Sandra Schuhmacher & Lisa Stolz

1 Einführung

Das MQTT Protokoll sieht in seiner Spezifikation drei verschiedene **Q**uality **of S**ervice (QoS) Level vor, die je nach Level unterschiedliche Garantien bieten. So können gesendete Nachrichten bestätigt oder die Anzahl der neugesendeten Nachrichten begrenzt werden. Ziel dieses Experiments ist die Untersuchung des Einflusses der verschiedenen QoS-Level auf:

- ✓ Latenzzeiten bei variierender Nutzdatengröße
- ✓ Latenzentwicklung bei limitierter Bandbreite
- ✓ Latenzentwicklung bei Paketverlust
- ✓ Protokolloverhead im Vergleich zu den Nutzdaten

Zur Messung des Einflusses schreiben die Clients jeweils ein Log (siehe Sektion Software). Aus den Zeitstemplen der Logs wird die Roundtriptime (RTT) berechnet. Diese wird unter den verschiedenen Einflussfaktoren beobachtet. Die Bandbreitenlimitierung wird mit dem Linux Traffic Shaper `tc` auf dem Client vorgenommen. Der Paketverlust mit `iptables` ebefalls auf dem Client vorgenommen. Der Protokolloverhead wird mit dem Netzwerkanalysetool `wireshark` beobachtet.

1.1 Versuchsaufbau Hardware

Für den Hardwareaufbau werden folgende Komponenten verwendet:

Broker Zu Beginn des Experiments wird ein *Raspberry Pi 3* als MQTT Broker eingesetzt. Dieses wird im späteren Verlauf des Experiments durch einen Desktop Computer ersetzt aufgrund von technischen Einschränkungen

Client Als Client wird ein *Lenovo X230 Tablet* Computer verwendet

Netzwerk Alle beteiligten Geräte verwenden eine Gigabit LAN Netzwerkkarte und werden über einen *Cisco Linksys E2000* Router via CAT5 Kabelnetzwerk verbunden um in einer isolierten Umgebung messen zu können

Der schematische Hardwareaufbau kann in Abbildung 1 nachvollzogen werden. Die relevanten Leistungsdaten der Hardware sind in Tabelle 1 aufgelistet.

Abbildung 1. Hardware Setup

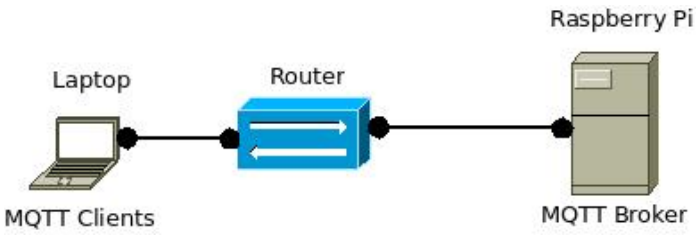


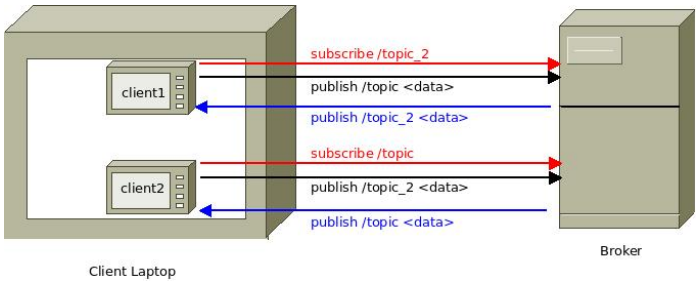
Tabelle 1. Leistungsspezifikation der Hardware

Hardware	CPU Clock Speed	Memory	Festplatte
Raspberry Pi 3	ARM Cortex-A53 @ 1.4 GHz	1GB LPDDR2 (900 MHz)	Transcend 8GB MicroSD Class 6, 6 MB/s
Lenovo X230 Tablet	Intel Core i5-3320M @ 2.6 GHz	4GB DDR3 (1600MHz)	SAMSUNG SSD PM83 128GB, 6 Gb/s
Desktop Computer	AMD FX 4350 @ 4.2 GHz	12GB DDR3 (1333MHz)	WDC WD5000HHTZ-0 500GB, 6 Gb/s, 10000U/min

1.2 Versuchsaufbau Software

Als MQTT Broker wird die freie Implementierung **Eclipse Mosquitto** eingesetzt. Auf Client-Seite wird die Python Library **Eclipse Paho** verwendet um zwei Client Programme zu implementieren.
Den schematischen Ablauf der Roundtrip-Messung ist in Abbildung 2 zu sehen. Client1 führt Publish Anfragen unter einem Topic aus und subscribed auf ein Antwort Topic. Client2 published auf das Antwort Topic und subscribed auf das Topic. Die Publish Aktionen der Clients sind im Bild schwarz dargestellt. Die Subscribe Aktionen sind in Rot dargestellt. Der Broker sendet jedem Subscriber eines Topics einen Publish wenn er neue Nachrichten auf ein Topic erhält - diese sind im Bild blau dargestellt.

Abbildung 2. Software Setup



1.2.1 client1.py

(Quellcode: `https://github.com/yulivee/mqtt-qos-rountrip/blob/master/client1.py`)

client1.py dient zur Messung der **R**ound**t**rip**t**ime (RTT). Das Programm setzt nach Verbindung zum Broker in modifizierbaren zeitlichen Abständen publish Anfragen zum Broker auf einen wählbaren topic-namen ab. Zuvor subscribed es sich auf den topic-namen `<gewähltes topic>_2`. Bei jedem Publish und jedem empfangenen Paket wird ein Logeintrag geschrieben. Das Programm lässt über Kommandozeilenparameter die Modifikation folgender Optionen zu:

- ✓ QoS Level (`--qos_level`)
- ✓ Nutzdaten (`--file`)
- ✓ Länge der Sendeperiode in Minuten (`--time`)
- ✓ Anzahl der versendeten Pakete (`--cycles`), alternative zur Sendeperiode
- ✓ Anzahl Publish Anfragen pro Sekunde (`--pbs`)

1.2.2 client2.py

(Quellcode: `https://github.com/yulivee/mqtt-qos-rountrip/blob/master/client2.py`)

client2.py dient als Relay um die Publish Anfragen von client1 entgegen zu nehmen und unter dem topic-namen <`gewähltes topic`>_2 unverändert zu publishen. Auf diese Weise sendet das Programm im Versuchsaufbau die Daten zu client1 zurück.

1.2.3 IPC

Um die Messungen besser automatisieren zu können betreiben die beiden clients eine primitive Form von Interprozess-Kommunikation. Beide Clients schreiben beim Start jeweils ihre Prozess-ID in eine `.pid` Datei, die der jeweils andere Prozess einliest. Zusätzlich schreibt client1 das topic, das aus den gewählten Versuchsparametern (QoS Level, Sendeperiode oder Anzahl, Pakete pro Sekunde) generiert wird in die Datei `topic.ipc`. Client2 liest dieses Topic ein und hängt ein `_2` an um das Antwort-Topic zu ermitteln. Über die Signalhandler kommunizieren die Clients miteinander. Client1 fordert über `SIGUSR1` Client2 zum einlesen des Topics auf, Client2 signalisiert über das selbe Signal das Client1 mit dem Publish beginnen kann. Auf diese Weise funktioniert die Kommunikation auch wenn einer der beiden Clients beim connect zum Broker länger benötigt. Über das Signal `SIGUSR2` nehmen beide Clients einen Disconnect vom Broker vor. So lassen sich eine Reihe verschiedener Messungen automatisiert vornehmen.

1.2.4 Logging

Um später die RTT berechnen zu können schreiben beide Clients ein Log. Zur einfachen Auswertung der Logs mit der Analyse-Suite `R` werden die Logs direkt im CSV-Format abgelegt und jegliche Statusnachrichten und Fehler auf die Konsole ausgegeben statt geloggt. Code-Listing 1 zeigt die Möglichen Logeinträge und enthaltenen Informationen. Diese enthalten:

1. Zeitstempel
   `Jahr-Monat-Tag_Stunden:Minuten:Sekunden:Millisekunden`
2. Aktion
   `sent,received,fail,discard`
3. Topic Name
   `QoS Level - Nutzdatengröße - Sendedauer - Pakete pro Sekunde`
4. QoS Level
5. Nutzdatengröße
6. Paket ID zur eindeutigen Identifikation

```
#timestamp,action,topic,qos_level,size,paket-id
#example publish
2018-05-23,22:09:04:213,sent,mqtt-roundtrip-qos0-100Byte-1-minutes,0,100,000000
#example receive
2018-05-23,22:09:04:216,received,mqtt-roundtrip-qos0-100Byte-1-minutes_2,0,106,000000
#example for failed publish
2018-05-24,01:48:06:995,fail,mqtt-roundtrip-qos2-10KByte-1-minutes-1pbs,2,10240,000014
2018-05-24,01:48:06:996,discard,mqtt-roundtrip-qos2-10KByte-1-minutes-1pbs,2,10240,000014
```

Listing 1. Client Log Format

Beim Logging zeigt sich eine erste Hürde: Das MQTT von sich aus anonym mit Nachrichten agiert - es gibt im Protokoll keinen eindeutigen Identifier um festzustellen welche Nachricht von welchem Client stammt. Ab QoS-Level 1 gibt es zwar eine Message-ID, diese wird allerdings nur zwischen Client und Broker verwendet um die Bestätigung von Publishes abzuhandeln und anschließend verworfen. Client2 hat beim receive eines Pakets keine Information welches Paket das ist und somit kann mit Protokollmitteln nicht festgestellt werden zu welchem Publish ein Receive nach dem Roundtrip gehört. Um das Problem zu beheben werden die ersten 6 Bit der Nutzdaten als ID verwendet, die client1 pro publish um 1 erhöht. Durch auslesen der ersten 6 Bit Nutzdaten lässt sich so ein Paket eindeutig identifizieren. Der letzte Eintrag paket-id in den Logs entspricht dieser ID. Die geringfügige Verzerrung der Messung um jeweils 6 weitere Bits wird hingenommen.

2 Messung 1: Latenz bei voller Brandbreite

In einem ersten Versuchsaufbau wurde als Broker ein Raspberry Pi verwendet. In späteren Messungen wurde dieser Aufbau abgewandelt und der Pi durch einen Laptop ersetzt, da es aufgrund der geringeren Performance des Pis zu Abbrüchen und verzerrten Messungen kam. Die technischen Hintergründe werden im Kapitel 6 (Diskussion) dargelegt.
Zum Vergleich wurden in den folgenden beiden Tabellen die aggregierten Messungen für beide Versuchsaufbauten dargestellt. Im weiteren Verlauf werden nur noch Ergebnisse mit dem Laptop als Broker präsentiert.

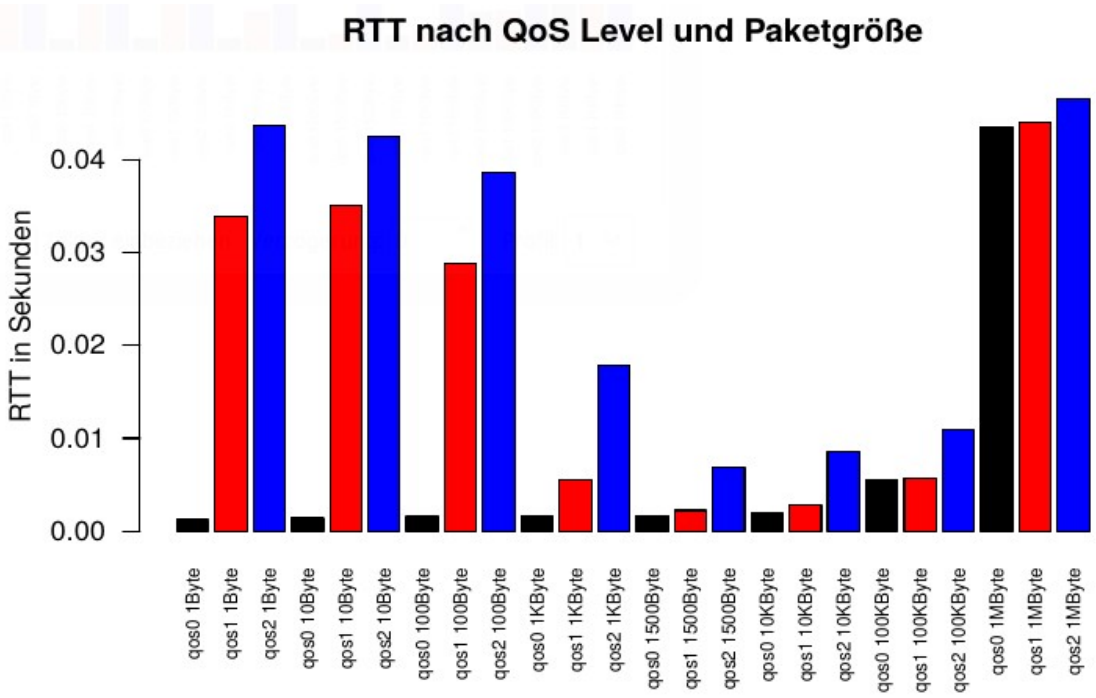
Tabelle 2. Latenzzeit in Abhängigkeit der Paketgröße - Pi

Nutzlast	QoS-0	QoS-1	QoS-2
1 Byte	0.1813469	0.1419741	0.1934519
10 Byte	0.1480741	0.1468307	0.3937651
100 Byte	0.1596423	0.1565406	0.2034270
1 KB	0.2593080	0.1876840	0.5348480
1500 Bytes (MTU Size Limit)	0.1902710	0.1899368	0.9265137
10 KB	0.8326793	0.8707715	4.4394398
100 KB	7.9708026	8.3323956	7.1009707
1 MB	12.8628915	5.3124085	7.2580572

Tabelle 3. Latenzzeit in Abhängigkeit der Paketgröße - Laptop

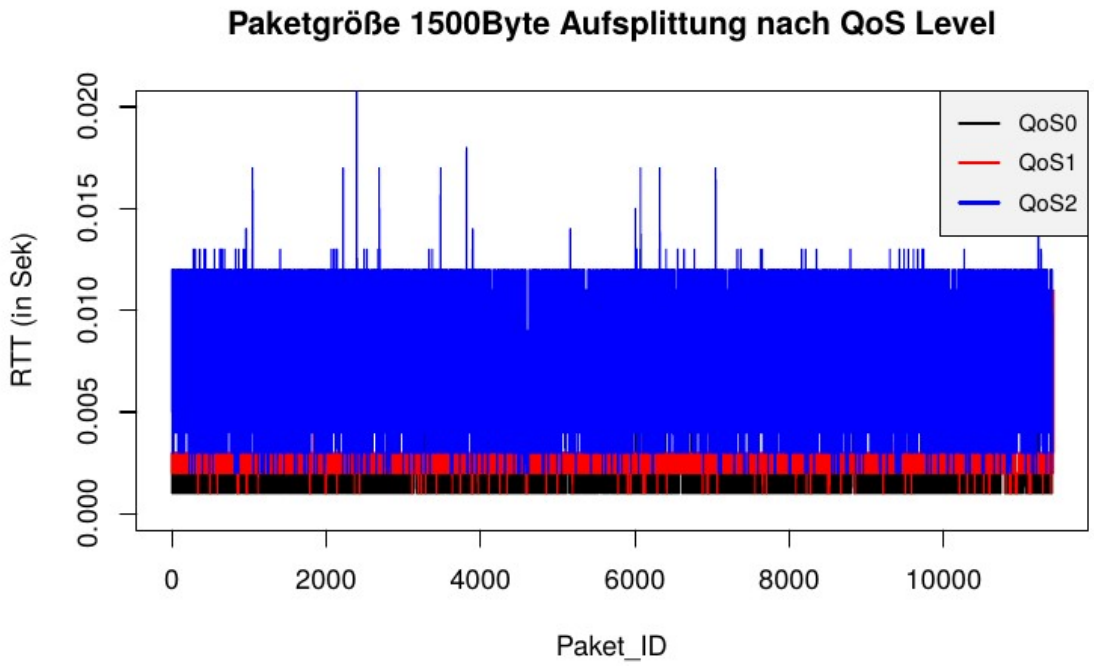
Nutzlast	QoS-0	QoS-1	QoS-2
1 Byte	0.0012375	0.0339261	0.0435640
10 Byte	0.0014071	0.0349903	0.0424829
100 Byte	0.0015745	0.0288117	0.0386520
1 KB	0.0016336	0.0054663	0.0178899
1500 Bytes (MTU Size Limit)	0.0017385	0.0022554	0.0068914
10 KB	0.0020183	0.0027985	0.0085598
100 KB	0.0055814	0.0056585	0.0109123
500 KB	18.7941932	18.9917111	0.0247059
1 MB	0.0434897	0.0440231	0.0465334
10 MB	0.4217931	0.4274530	0.4308966

Die in diesem Kapitel präsentierten Daten wurden auf Basis von 60 generierten Logs aufbereitet. Nach Berechnung der RTT auf Basis der Client1 Logs wurden somit 30 einzelne Datensätze generiert, die anschießend zu einem Gesamtdatensatz mit über 260 Tausend Beobachtungen (versandten Paketen) zusammengefasst wurden.
Durch eine Aggregation auf QoS und Nutzlast Level, konnten die in Tabelle 2 und 3 dargestellten, durchschnittlichen RTT Werte ermittelt werden. Diese Übersicht ist für einen ersten Eindruck sinnvoll und hilft z.B. bei der Ermittlung von Ausreißern wie den sehr hohen RTT Werten bei 500KB Nutzlast. Ohne diese Anomalie können aus der folgenden Grafik erste Erkenntnisse bezüglich des Einflusses der QuOs Level und Nutzlast auf die RTT gewonnen werden. Weitere Abbildungen und Informationen zur Datenaufbereitung sind unter (R Auswertung: `/https://github.com/yulivee/mqtt-qos-rountrip/tree/master/R_Analysis`) zu finden

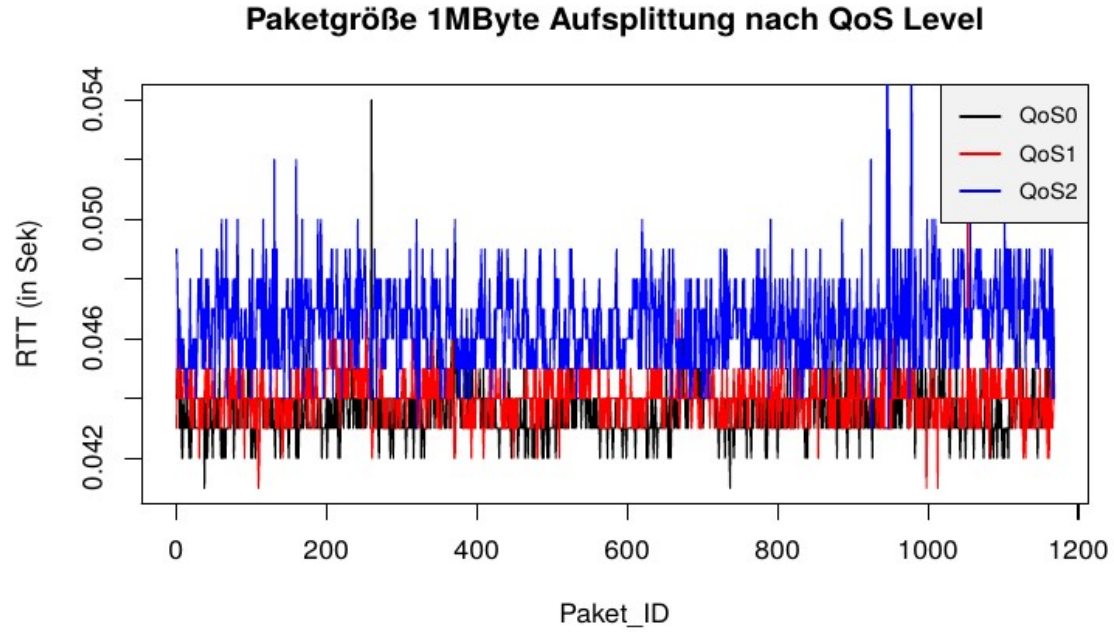


- ✓ Es ist zu beobachten, dass die RTT sich über alle dargestellten Nutzlasten hinweg, im Millisekundenbereich (unter 50ms) bewegt.
- ✓ Die RTT steigt für die selbe Nutzlast mit dem QoS Level.
- ✓ Ab 1KB Nutzlast sinkt die RTT für QoS1 und QoS2.
- ✓ Bis 100Byte ist QoS0 konstant und mit weniger als 6ms sehr schnell.
- ✓ Ab 1MB Nutzlast steigt die RTT und die RTT ist fast identisch über die verschiedenen QoS Level hinweg.

Da für die hier dargestellten Messungen mit voller Bandbreite gesendet wurde, ist eine Darstellung von Graphen auf Detailebene weniger aussagekräftig, als in den folgenden Kapiteln.
Exemplarisch für andere Unterteilungen nach Paketgrößen zeigt die folgende Abbildung zur Paketgröße von 1500Byte, dass die Menge an Beobachtungen lediglich als Balken erscheint. Wie zu erwarten, sind Pakete mit QoS0 Level am schnellsten, gefolgt von QoS2 und anschließend mit größerer Varianz aber dennoch unter 15ms QoS2.



Anders verhält es sich für sehr große Nutzlast, da bei voller Bandbreite im Zeitfenster von einer Minute weniger Pakete transportiert werden und somit in die Messung eingehen. Die folgende Grafik für 1MB Nutzlast stellt nur ein zehntel der Beobachtungen im Vergleich zu der 1500 Byte Messung dar.
Wie zu erwarten sind auch hier die Pakete mit QoS0 am schnellsten gefolgt von QoS1 und QoS2. Insgesamt bewegt sich die RTT im Bereich von 40 bis 50ms, somit sind Pakete mit größerer Nutzlast langsamer.



3 Messung 2: Latenz bei begrenzter Brandbreite

3.1 Einsatz des Traffic Shapers

Für die Messung bei begrenzter Bandbreite wird der Netzwerkdurchsatz künstlich mit dem Traffic-Shaper `tc` limitiert. Es wird über den Zeitraum von einer Minute mit einer konstanten Nutzlastgröße von 10KB Pakete veröffentlicht.

Variiert wird der zeitlichen Abstand der Veröffentlichungen zwischen einem, 10 und 100 Pakete pro Sekunde

Betrachtet wird die Latenzzeit im Vergleich zwischen den QoS-Modi.

Das traffic shaping erfolgte mit dem script `adjust-bandwidth.sh`
(Quellcode: `https://github.com/yulivee/mqtt-qos-rountrip/blob/master/adjust-bandwidth.sh`)

```
#!/bin/bash
SPEED="10kbps"
IFACE="enp0s25"

#show current rules
tc class show dev $IFACE

#clear all tc rules
sudo tc qdisc del dev $IFACE root

#throttle
sudo tc qdisc add dev $IFACE handle 1: root htb default 11
sudo tc class add dev $IFACE parent 1: classid 1:1 htb rate $SPEED
sudo tc class add dev $IFACE parent 1:1 classid 1:11 htb rate $SPEED

#show current rules
tc class show dev $IFACE
```

adjust–bandwidth.sh

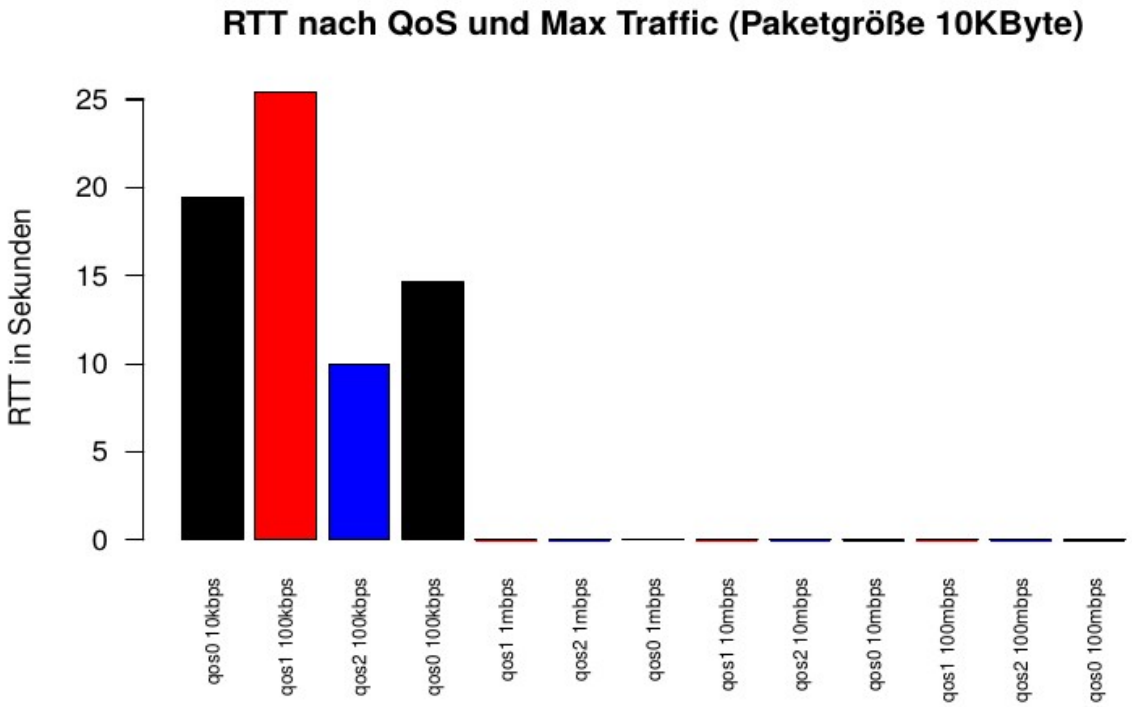
3.2 Auswertung der Messung mit TC

Analog zur Datenaufbereitung in Kapitel 2, wurden die Daten der rund 80 Traffic-Shaper (TC) Logs bearbeitet und aggregiert. Gesendet wurden alle Pakete mit der Nutzlast 10KB und somit ist eine Begrenzung auf 10kbps entsprechend streng und führt zu hohen RTT sowie Übertragungsabbrüchen auf QoS1 und QoS2 Level. Ein ähnliches Bild zeigt sich für eine Begrenzung auf 100kbps.

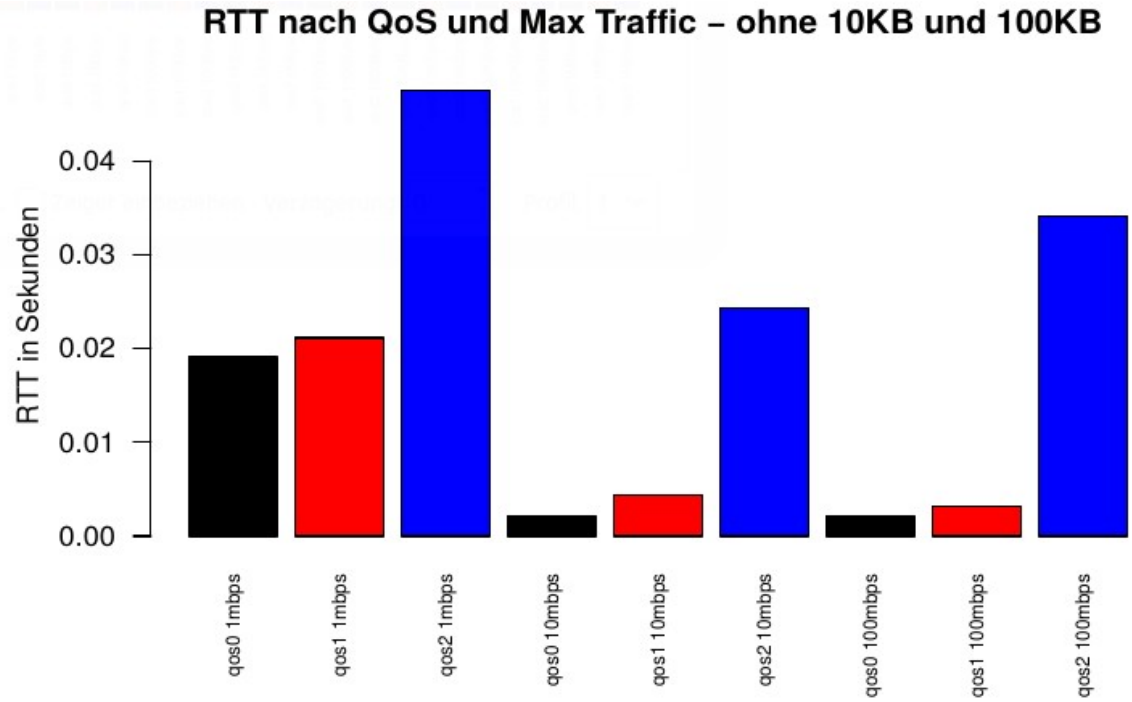
Tabelle 4. Bandbreitenbegrenzung

Pakete/s	QoS-0	QoS-1	QoS-2
10kbps	19.4711302		
100kbps	25.3815006	9.9820152	14.6274025
1mbps	0.0190646	0.0211253	0.0474118
10mbps	0.0021778	0.0043793	0.0242875
100mbps	0.0022214	0.0030982	0.0341508

Die eins zu zu eins Abbildung der Daten aus der Tabelle in Form eines Graphen führt aufgrund der deutlich höheren RTT Zeiten für eine starke Beschränkung zu folgendem Bild:

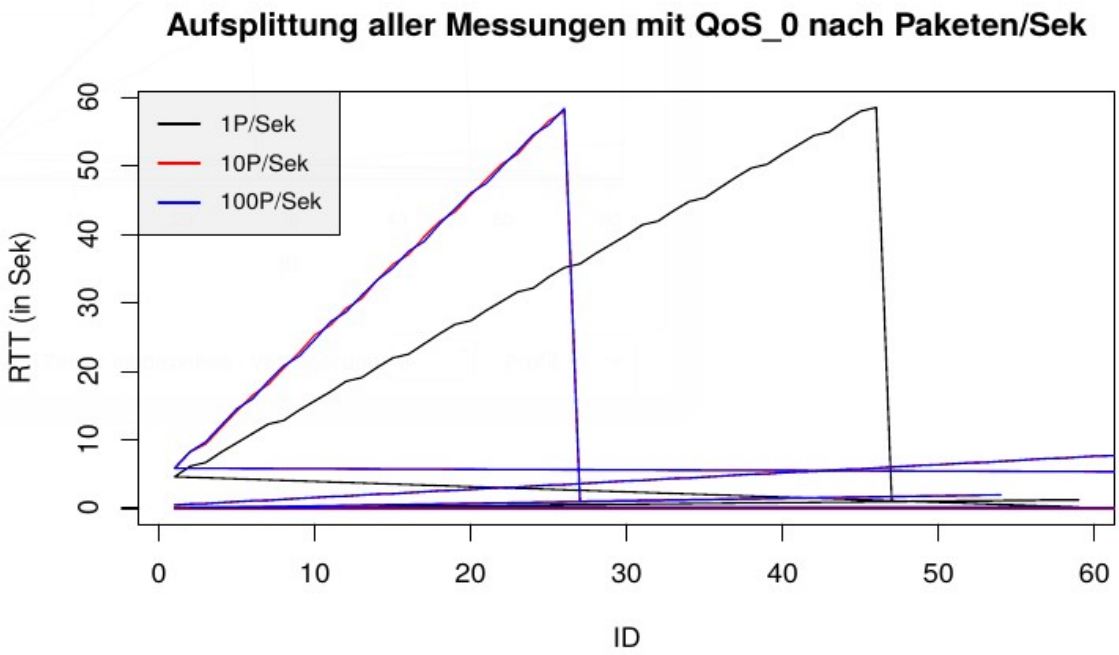


Auffällig ist die relativ niedrige RTT für QoS2 bei 100kbps - im Vergleich zu den anderen QoS Leveln. Eine mögliche Erklärung wird am Ende des Kapitels nach einer eingehenden Betrachtung der Daten präsentiert.
Es ist sinnvoll den selben Graphen im nächsten Schritt ohne die extremen Werte zu betrachten und somit ein Bild der Messungen mit deutlich niedrigeren RTTs im Bereich von weniger als 50ms zu erhalten.
Hier ist ein ähnliches Ergebnis wie bei der Messung mit voller Bandbreite zu beobachten. Mit höherem QoS Level steigt die RTT.

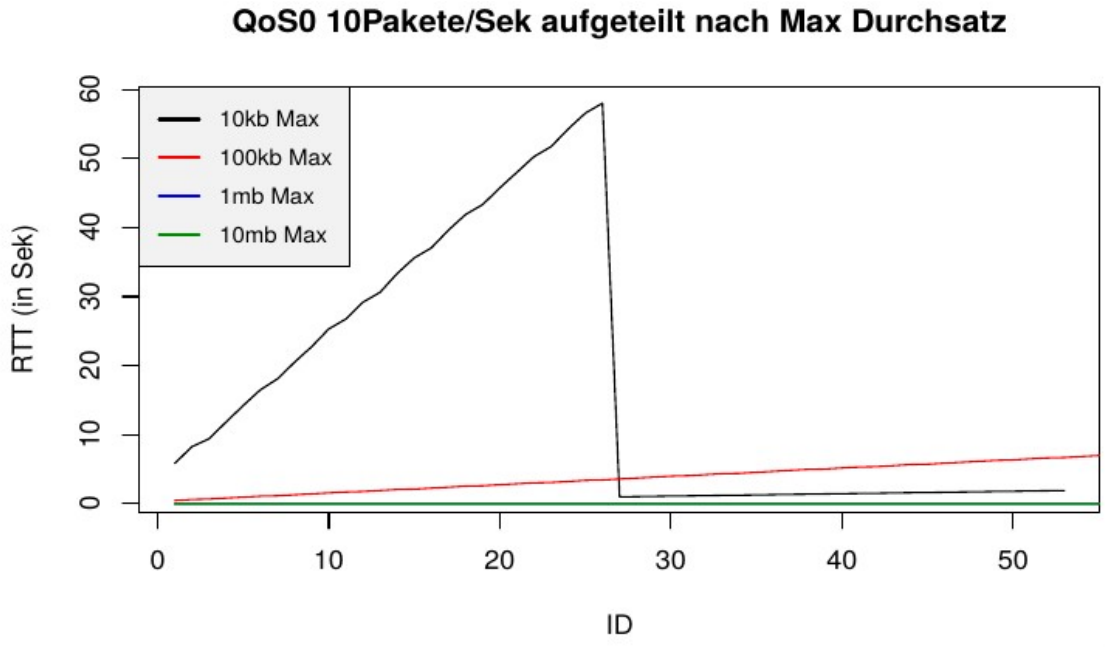


Im folgenden werden die aggregierten Ergebnisse exemplarisch für bestimmte QoS Level, Übertragungsgeschwindigkeiten und Trafficbegrenzungen detailliert betrachtet. Eine breitere Auffächerung für weitere Alternativen und Kom-

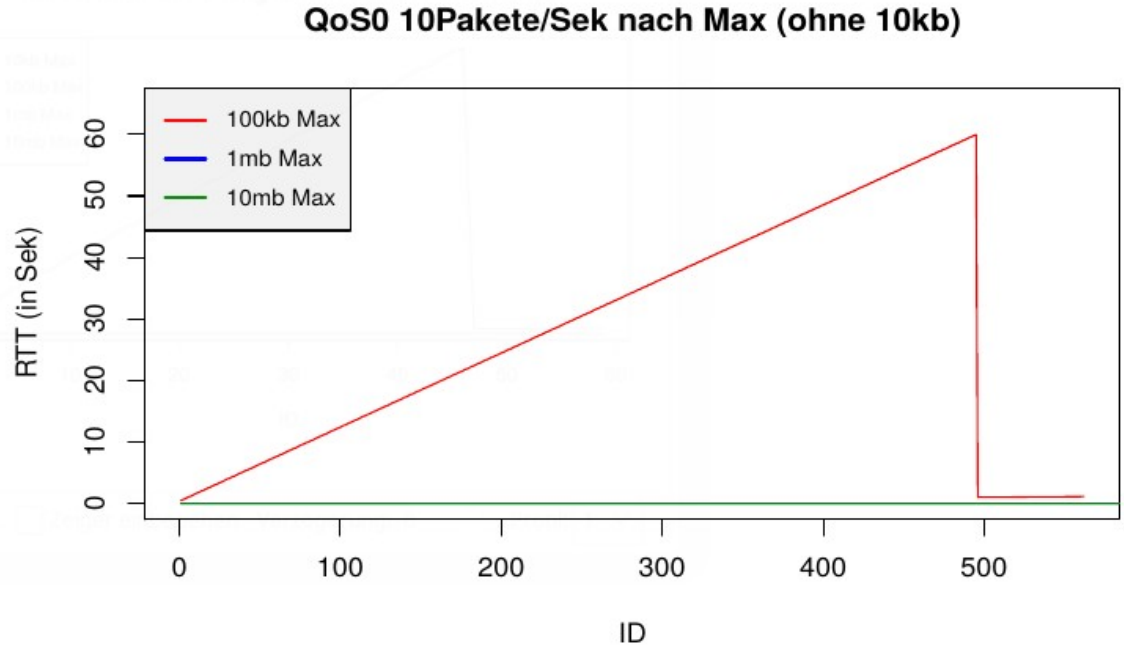
binationen kann unter (Anhang3: `https://github.com/yulivee/mqtt-qos-rountrip/R_Analysis/02_TC/rttGraphenTC.pdf`) eingesehen werden.



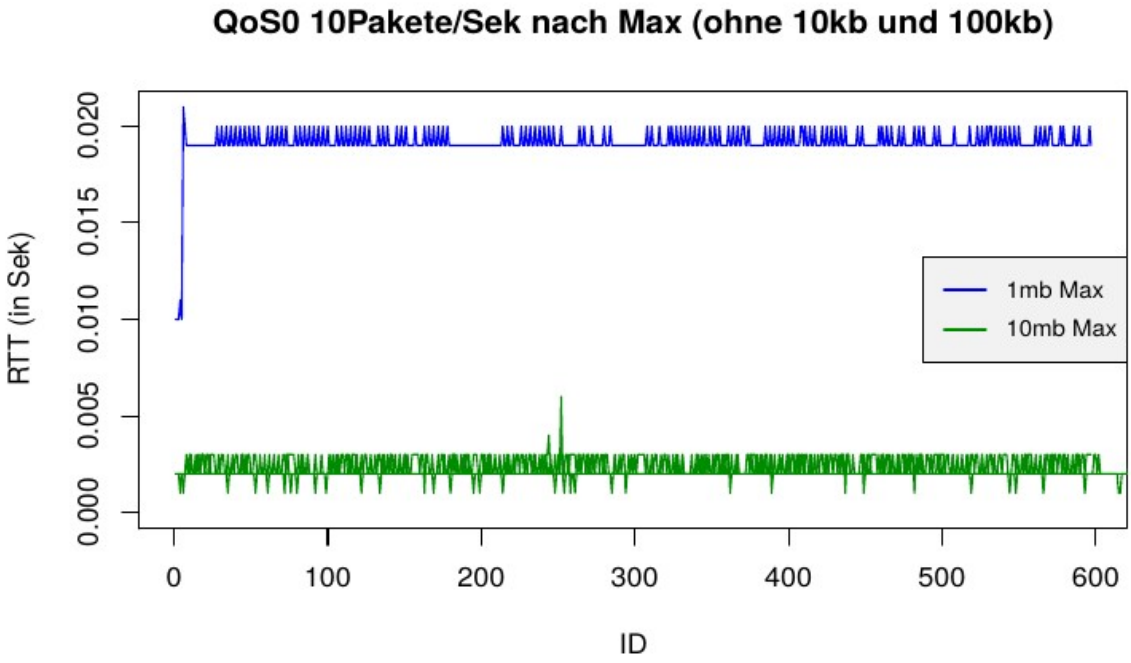
Auf der ersten Ebene der Unterteilung wurden die Messungen nach QoS Leveln untrteilt und in der vorangegangenen Grafik nach Übertragungsgeschwindigkeiten geplottet. Das Bild zeigt die charakteristischen ”Haifischflossen”die häufig in den TC Messungen zu beobachten waren.
Für einen genaueren Einblick ist jedoch auch diese Grafik noch zu verdichtet. Die Linien können nicht einer Variablen zugeordnet werden. Daher wurde in der folgenden Abbildung die Messungen für QoS0 bezüglich der Geschwindigkeit 10 Pakete pro Sekunde weiter unterteilt in die verschiedenen Begrenzungen.



Auch hier ist die Haifischflosse für eine Begrenzung auf 10KB zu erkennen. Bei noch genauerer Betrachtung, wenn die 10KB Beobachtungen außen vor gelassen werden, ist eine weitere Flosse zu beobachten - für eine Begrenzung auf 100kb.

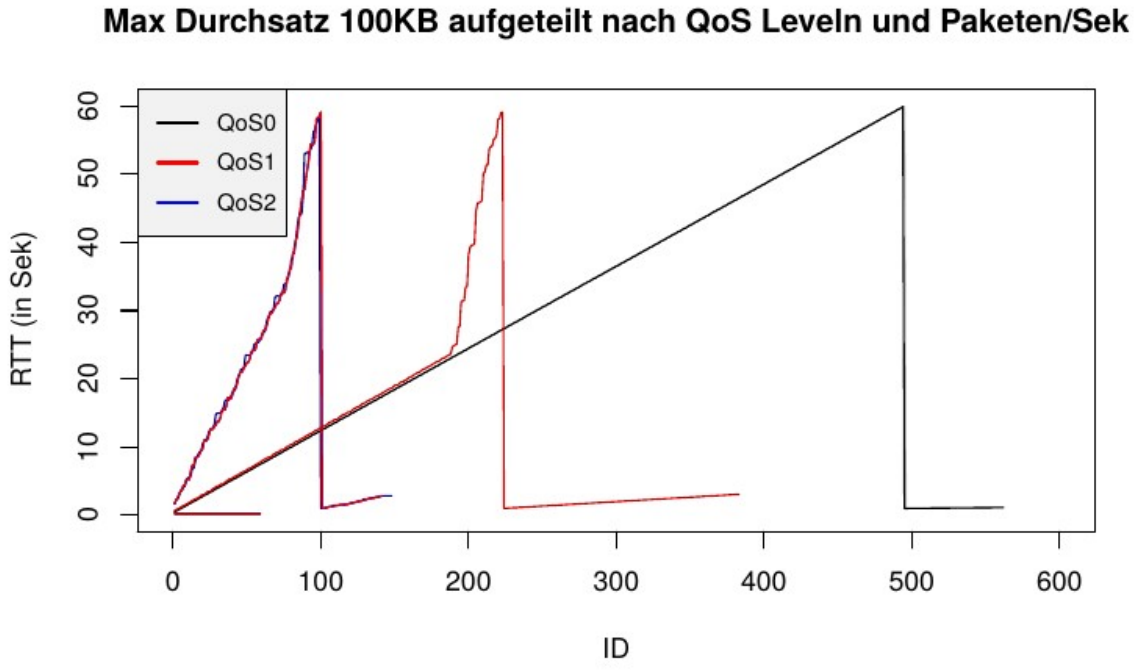


Erst danach - ab 1MB - ist die RTT relativ konstant und bewegt sich um die 19ms (1MB) und 3ms(10MB).



3.3 Erklärungsansatz Haifisch und niedrige RTT für QoS2

Die bis hier dargestellten Beobachtungen zur Haifischflosse und der anfangs beschriebenen geringen RTT für das QoS2 Level bei einer Beschränkung auf 100KB, können möglicherweise wie folgt im Ansatz erklärt werden:
Für starke Begrenzung (100KB) versendet QoS2 nicht mit einer Rate von 10 oder 100 Paketen pro Sekunde. QoS0 und QoS1 hingegen senden mit der höheren Rate und durch die verringerte Bandbreite scheint es zu einer Art “Stau” zu kommen. Die RTT steigt mit jedem neuen Paket und bricht schließlich wieder ein auf wenige Millisekunden. Dieses Verhalten kann in der folgenden Abbildung für alle drei QoS Level beobachtet werden. Die erste Flosse bildet sowohl QoS1 in rot als auch QoS2 (Messung zu 100 Paketen pro Sekunde) in blau ab.



Die geringere RTT für QoS2 lässt sich rechnerisch so erklären, dass für QoS2 bei den Messungen mit höherer Sendungsrate weniger Beobachtungen (versandte Pakete) gespeichert wurden:

- ✓ QoS0: 59(1pbs), 562(10pbs), 562(100pbs)
- ✓ QoS1: 59(1pbs), 141(10pbs), 383(100pbs)
- ✓ QoS2: 59(1pbs), 141(10pbs), 148(100pbs)

Der Grund hierfür ist die nicht befolgte höhere Sendungsrate, was bei einer fixen Messzeit von einer Minute zu weniger versandten Paketen führt.
Somit gehen weniger Messungen mit hohen RTT Wertn in den Durchschnittswert am Anfang des Kapitels ein.

4 Messung 3: Latenz bei Paketverlust

4.1 Simulation von Paketverlusten

Für die Messung unter Simulation von Paketverlust wird `iptables` verwendet. Es wird über den Zeitraum von einer Minute mit einer konstanten Nutzlastgröße von 10KB Pakete veröffentlicht.

Variiert wird der zeitlichen Abstand der Veröffentlichungen zwischen einem, 10 und 100 Pakete pro Sekunde

Betrachtet wird die Latenzzeit im Vergleich zwischen den QoS-Modi.

Die Modifikation des Paketverlust erfolgte mit dem script `adjust-packetloss.sh`
(Quellcode: `https://github.com/yulivee/mqtt-qos-rountrip/blob/master/adjust-packetloss.sh`)

```
#!/bin/bash
LOSS="$1"
IFACE="enp0s25"

echo "setting $LOSS% packetloss on interface on $IFACE"

#show current rules
sudo iptables -L
sudo iptables -F
sudo iptables -A INPUT -m statistic --mode random --probability $LOSS -j DROP
sudo iptables -A OUTPUT -m statistic --mode random --probability $LOSS -j DROP
sudo iptables -L
```

adjust–packetloss.sh

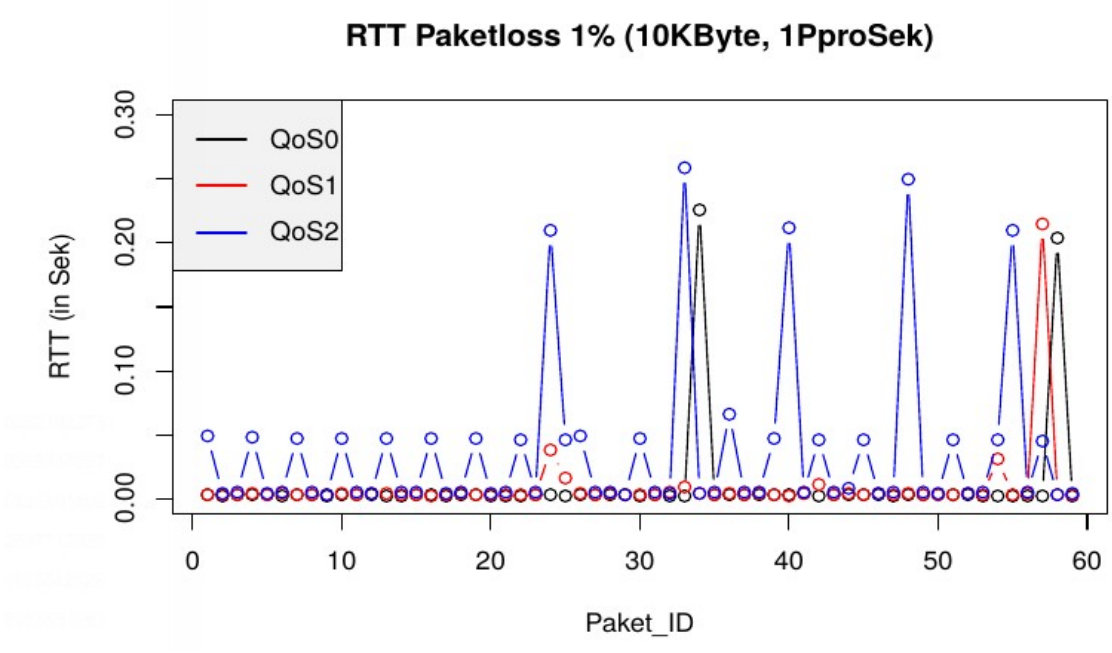
4.2 Auswertung der Messung mit Paketverlusten

Auch im dritten Messaufbau wurde bei der Datenaufbereitung analog des bereits beschriebenen Schemas vorgegangen. Der Gesamtdatensatz wurde aus den rund 60 generierten Logs erstellt und besteht somit aus ca 30 einzelnen Datensets.
In der folgenden Tabelle sind erneut die aggregierten Durchschnittswerte eingetragen.

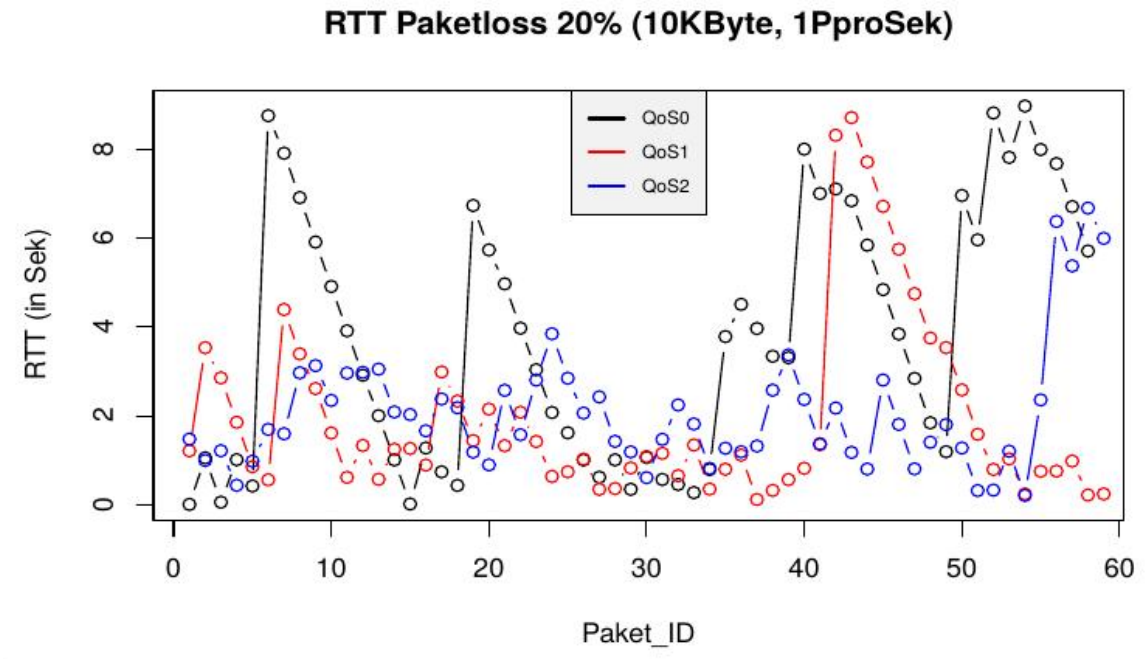
Tabelle 5. Paketverlust

Paketverlust	QoS-0	QoS-1	QoS-2
1%	0.0098475	0.0084068	0.0367627
5%	0.1312034	0.0587966	0.1579322
10%	0.2266102	0.2934068	0.4273559
15%	2.4896271	0.7028474	1.2001356
20%	3.7678276	1.9415424	2.0728983
25%	8.1830323	19.7166666	29.6470833
30%	16.0160500	39.2010000	39.4201305

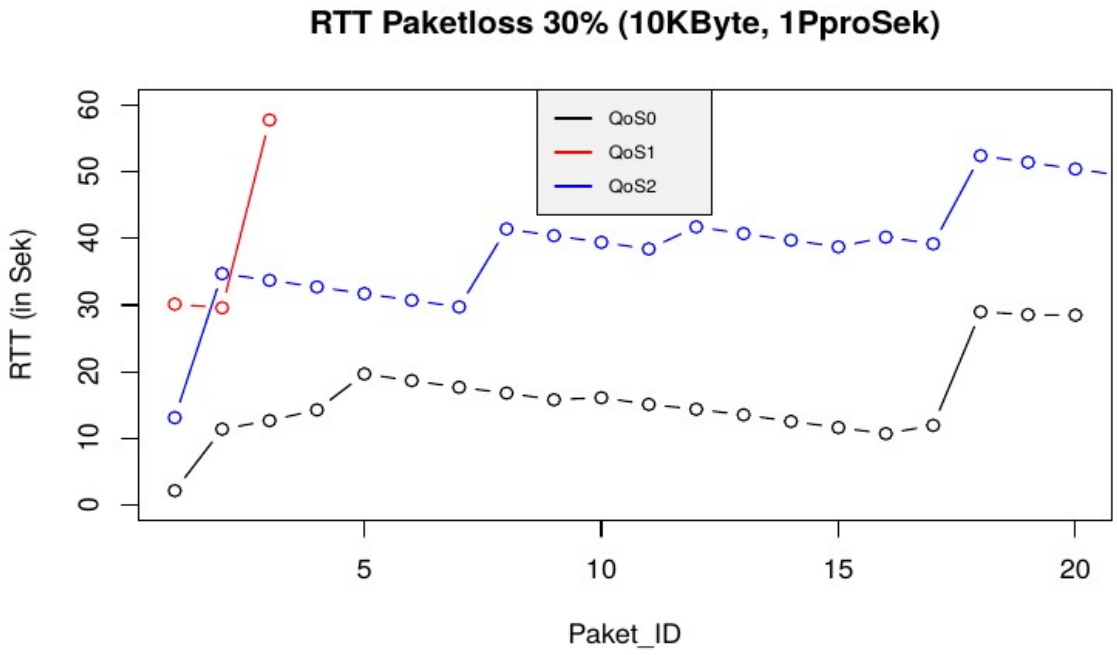
Die folgenden drei Graphen veranschaulichen, wie sich die RTT mit mehr Paketverlusten entwickelt. Für eine einprozentige Verlustrate ist die RTT, abgesehen von wenigen Ausreißern relativ kostant und größtenteils niedriger als 50ms.



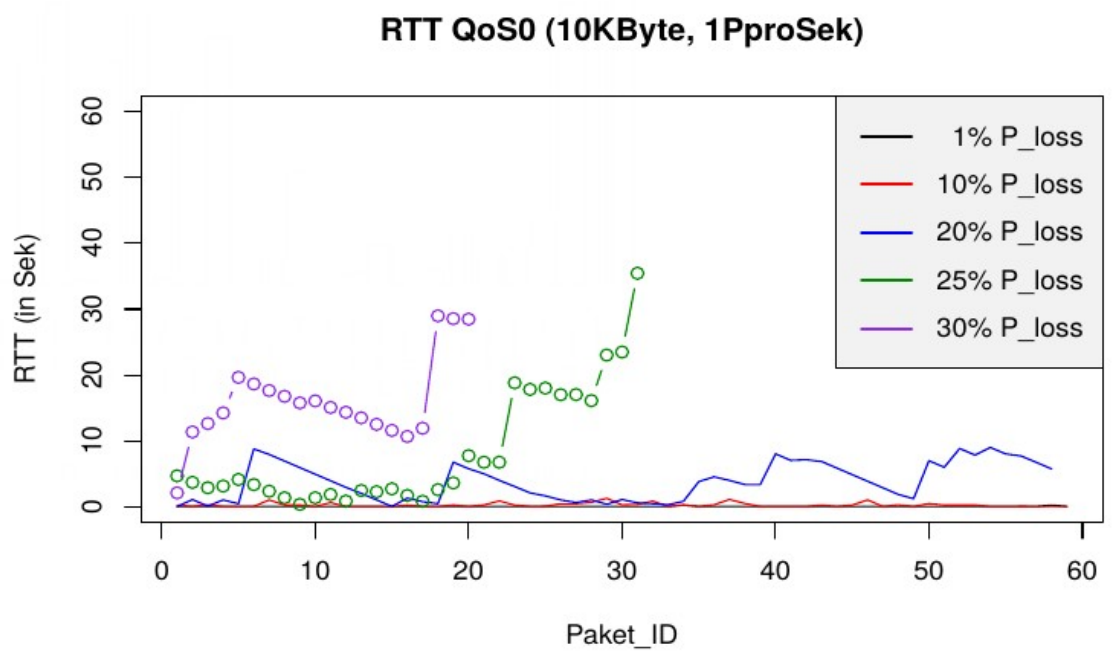
Mit einer zunehmenden Verlustrate steigt die RTT und die Ausschläge nehmen zu, wie der folgende Graph zeigt. Bei 20 Prozent bewegt sich die RTT im Bereich von größer 0 bis 9 Sekunden.
Weitere Abbildungen können unter (Anhang4: `https://github.com/yulivee/mqtt-qos-rountrip/R_Analysis/03_PLoss/rttGraphenPL.pdf`) eingesehen werden.

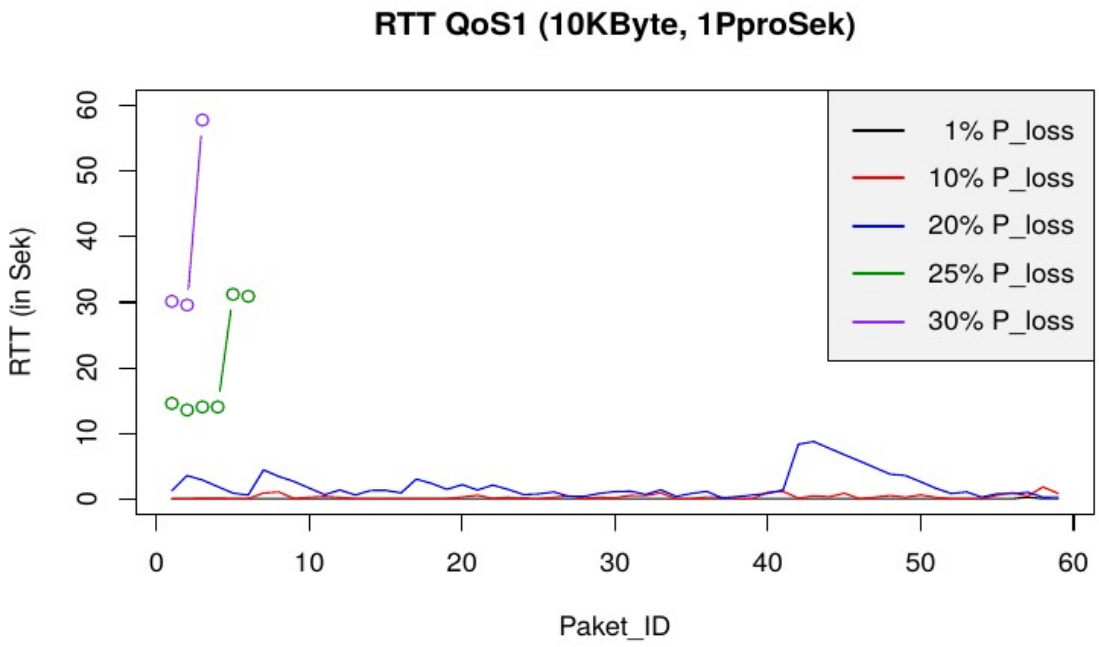


Bei einer Verlustrate von 30 Prozent, kommt es zu Abbrüchen bei Messungen des QoS1. QoS1 bewegt sich im Rahmen bis zu 30 Sekunden und das QoS2 bis zu ca 50 Sekunden.

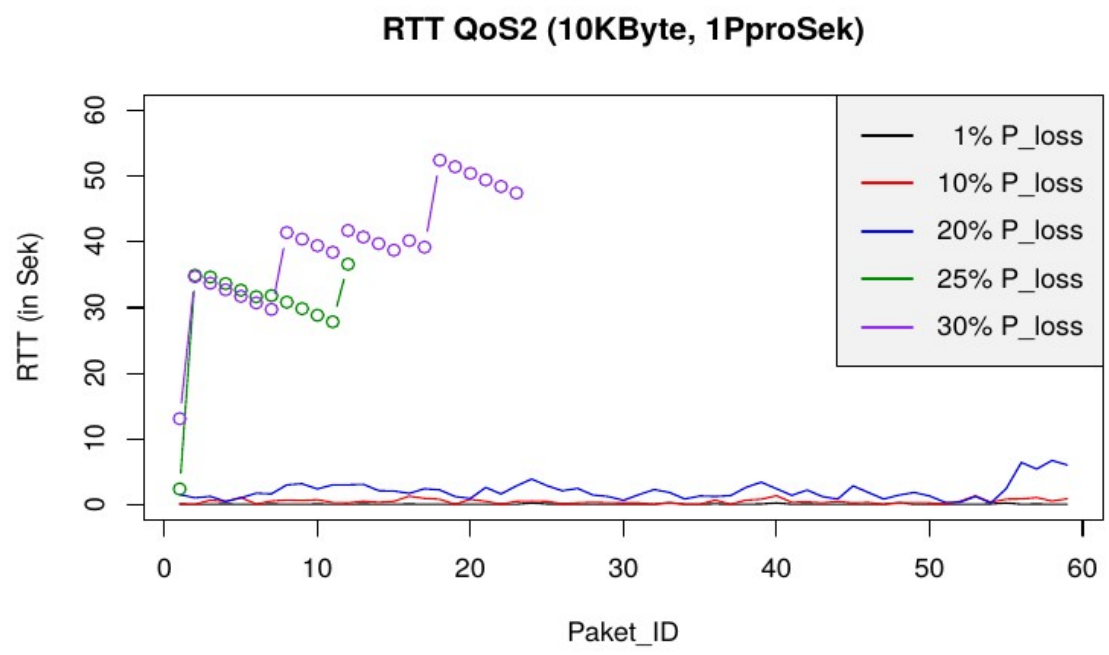


Bis jetzt wurde die RTT für die verschiedenen QoS Level verglichen, wobei immer die Verlustrate im jeweiligen Graphen fixiert war.
Im nächsten Schritt werden die verschiedenen Verlustraten direkt miteinander verglichen - jeweils in einem Graphen für QoS0, QoS1 und Qo2.





Durch die drei Abbildungen wird deutlich, dass ab einer Verlustrate von 20 Prozent die RTT langsam steigt, wobei sie bei 25 Prozent rapide zunimmt und zu Abbrüchen führt. Diese traten besonders früh für das QoS1 auf und für QoS0 und QoS2 nach ca einer halben Minute. Der RTT Anstiegt ist hierbei im Graphen QoS2 (auf über 50 Sekunden) deutlich steiler als für QoS0 (bis zu 40 Sekunden).



5 Messung 4: Protokolloverhead

Zur Ermittlung des MQTT-Protokolloverheads werden 10 Publish-Requests bei einem Paket pro Sekunde an den Broker gesendet und währenddessen der Netzwerkverkehr mit dem dem Programm `wireshark` aufgezeichnet

Variiert wird das QoS Level und der Umfang der Testdaten.

Betrachtet wird der Protokolloverhead im aufgezeichneten Netzwerkverkehr

5.1 Auswertung

Quelldaten: https://github.com/yulivee/mqtt-qos-rountrip/tree/master/logs/overhead
Für die Betrachtung wird nur das MQTT Protokoll betrachtet - TCP Prolog und Epilog und ACK Nachtrichten werden nicht eingerechnet. Der Protokoll Anteil von OSI-Layer 1-3 (Ethernet,IP,TCP) ist zudem aus den Nachrichten herausgerechnet.

5.1.1 Gemeinsamkeiten zwischen allen QoS Modi

Bei Betrachtung des Netzwerkverkehrs für das MQTT Protokoll lässt sich zunächst feststellen das die Nachrichten für den Connect und den Disconnect zum Broker zwischen den QoS-Modi immer die gleichen bleiben.

Tabelle 6. Overhead für MQTT Connect und Disconnect

Nachricht	Overhead	Paket
MQTT Connect	25byte	Connect Command
	4 byte	Connect ACK
MQTT Disconnect	2 byte	Disconnect Req

Es fällt auf, das die verschiedenen QoS Modi zwar Sicherheiten für die Auslieferung von Paketen bieten, aber keine Sicherheiten für die Verbindung zum Broker außer dem initialen Connect Ack. Dieses Problem muss auf Applikationsebene gelöst werden.

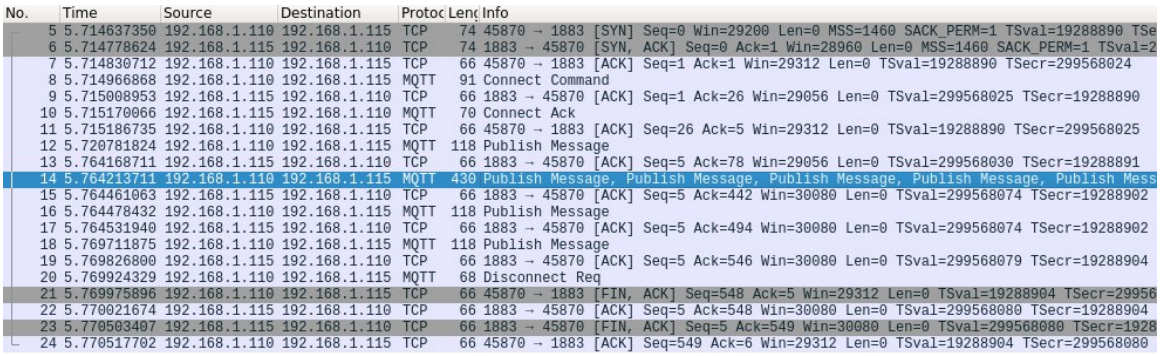
In sämtlichen Messungen hat sich gezeigt, das die Größe der Nutzdaten keinen Einfluss auf den MQTT-Protokolloverhead hat. Solange die Nutzdaten kleiner als die MTU-Size sind, wird die Nutzlast mit im MQTT `Publish Message` Paket verschickt. Somit gibt es nur den Overhead des `Publish Message` Pakets, der geringfügig zwischen den QoS Leveln variiert. Sobald die Nutzlast das MTU Limit überschreitet wird die Nutzlast eine Protokollebene tiefer auf mehrere TCP Frames verteilt und es wird pro TCP Frame ein `TCP ACK` versendet. Dies ist allerdings kein MQTT Overhead und der TCP Overhead ist auch abhänging von der verwendeten Hardware. So zeigt sich das bei großer Nutzlast die ersten TCP Nutzlastfragmente nicht komplett gefüllt sind. Dies kann daran liegen das der client die Nutzlast zunächst von der Festplatte laden muss und TCP nur senden kann was bereits im Speicher geladen ist. Damit variiert die Anzahl der TCP Fragmente in Abhängigkeit von der Lesegeschwindigkeit des Speichermediums.

5.1.2 QoS 0 Overhead

Auf QoS Level 0 werden keine Statusnachrichten verschickt. Für jedes gesendete Publish Paket gibt es einen TCP-ACK. Um Netzwerkoverhead auf den unteren Layern zu sparen werden Pakete mit kleinen Nutzdaten und kurzem zeitlichem Abstand gebundelt - das bedeutet das mehrere MQTT Nachrichten Körper in einem TCP Frame gemeinsam verschickt werden. Dies lässt sich in der blau markierten Zeil in Abbildung 3 beobachten. Bei sehr kleiner Nutzlast wird teilweise auch ein `Publish Message` Paket mit in ein `Disconnect Req` Paket gebundlet.
Der Protokolloverhead für die `Publish-Message` Nachricht setzt sich auf QoS Level 0 wie folgt zusammen:

- ✓ 1 byte Header Flags
- ✓ 2 byte Message Length

Abbildung 3. Trace des QoS 0 Overhead



No.	Time	Source	Destination	Protoc	Lenc	Info
5	5.714637350	192.168.1.110	192.168.1.115	TCP	74	45870 → 1883 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19288890 TSe
6	5.714778624	192.168.1.115	192.168.1.110	TCP	74	1883 → 45870 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=2
7	5.714830712	192.168.1.110	192.168.1.115	TCP	66	45870 → 1883 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=19288890 TSecr=299568024
8	5.714966868	192.168.1.110	192.168.1.115	MQTT	91	Connect Command
9	5.715008953	192.168.1.115	192.168.1.110	TCP	66	1883 → 45870 [ACK] Seq=1 Ack=26 Win=29056 Len=0 TSval=299568025 TSecr=19288890
10	5.715170066	192.168.1.115	192.168.1.110	MQTT	70	Connect Ack
11	5.715186735	192.168.1.110	192.168.1.115	TCP	66	45870 → 1883 [ACK] Seq=26 Ack=5 Win=29312 Len=0 TSval=19288890 TSecr=299568025
12	5.720781824	192.168.1.110	192.168.1.115	MQTT	118	Publish Message
13	5.764168711	192.168.1.115	192.168.1.110	TCP	66	1883 → 45870 [ACK] Seq=5 Ack=78 Win=29056 Len=0 TSval=299568030 TSecr=19288891
14	5.764213711	192.168.1.110	192.168.1.115	MQTT	430	Publish Message, Publish Message, Publish Message, Publish Message, Publish Mess
15	5.764461063	192.168.1.115	192.168.1.110	TCP	66	1883 → 45870 [ACK] Seq=5 Ack=442 Win=30080 Len=0 TSval=299568074 TSecr=19288902
16	5.764478432	192.168.1.110	192.168.1.115	MQTT	118	Publish Message
17	5.764531940	192.168.1.115	192.168.1.110	TCP	66	1883 → 45870 [ACK] Seq=5 Ack=494 Win=30080 Len=0 TSval=299568074 TSecr=19288902
18	5.769711875	192.168.1.110	192.168.1.115	MQTT	118	Publish Message
19	5.769826800	192.168.1.115	192.168.1.110	TCP	66	1883 → 45870 [ACK] Seq=5 Ack=546 Win=30080 Len=0 TSval=299568079 TSecr=19288904
20	5.769924329	192.168.1.110	192.168.1.115	MQTT	68	Disconnect Req
21	5.769975896	192.168.1.110	192.168.1.115	TCP	66	45870 → 1883 [FIN, ACK] Seq=548 Ack=5 Win=29312 Len=0 TSval=19288904 TSecr=29956
22	5.770021674	192.168.1.115	192.168.1.110	TCP	66	1883 → 45870 [ACK] Seq=5 Ack=548 Win=30080 Len=0 TSval=299568080 TSecr=19288904
23	5.770503407	192.168.1.115	192.168.1.110	TCP	66	1883 → 45870 [FIN, ACK] Seq=5 Ack=549 Win=30080 Len=0 TSval=299568080 TSecr=1928
24	5.770517702	192.168.1.110	192.168.1.115	TCP	66	45870 → 1883 [ACK] Seq=549 Ack=6 Win=29312 Len=0 TSval=19288904 TSecr=299568080

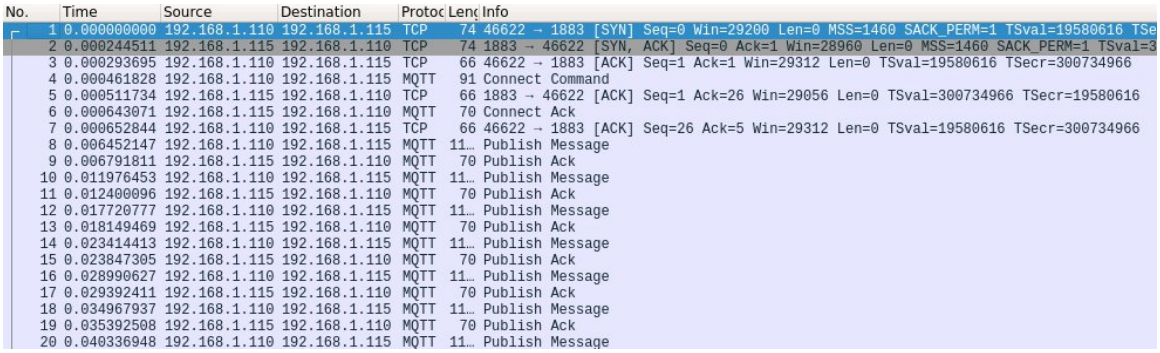
- ✓ 42 byte Topic Length (variiert mit dem gewählten Topic, bis zu 64KB)
- ✓ x byte Nutzlast (variiert mit der Nutzlast, bis zu 256MB)

Im Versuchsaufbau beträgt der Protokolloverhead damit 45 byte pro Publish.

5.1.3 QoS 1 Overhead

Auf QoS Level 1 gibt es pro versendeter `Publish-Message` Nachricht eine `Publish Ack` Nachricht. Diese ersetzt die `TCP ACK` Nachricht von QoS 0 und ist somit keine zusätzliche Nachricht, sondern sie erweitert den `TCP ACK` um 4 byte MQTT Overhead. QoS 1 bundelt auch keine Nachrichten, somit entsteht mehr Overhead da mehr Einzelpakete versendet werden. Dies macht sich allerdings bei Annäherung der Nutzlast an die MTU-Size im Vergleich zu QoS 1 nichtmehr bemerkbar, da QoS 0 dann durch die Größe der Nutzlast kein Bundeling mehr vornimmt.

Abbildung 4. Trace des QoS 1 Overhead



No.	Time	Source	Destination	Protoc	Lenc	Info
1	0.000000000	192.168.1.110	192.168.1.115	TCP	74	46622 → 1883 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19580616 TSe
2	0.000244511	192.168.1.115	192.168.1.110	TCP	74	1883 → 46622 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=3
3	0.000293695	192.168.1.110	192.168.1.115	TCP	66	46622 → 1883 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=19580616 TSecr=300734966
4	0.000461828	192.168.1.110	192.168.1.115	MQTT	91	Connect Command
5	0.000511734	192.168.1.115	192.168.1.110	TCP	66	1883 → 46622 [ACK] Seq=1 Ack=26 Win=29056 Len=0 TSval=300734966 TSecr=19580616
6	0.000643071	192.168.1.115	192.168.1.110	MQTT	70	Connect Ack
7	0.000652844	192.168.1.110	192.168.1.115	TCP	66	46622 → 1883 [ACK] Seq=26 Ack=5 Win=29312 Len=0 TSval=19580616 TSecr=300734966
8	0.006452147	192.168.1.110	192.168.1.115	MQTT	11…	Publish Message
9	0.006791811	192.168.1.115	192.168.1.110	MQTT	70	Publish Ack
10	0.011976453	192.168.1.110	192.168.1.115	MQTT	11…	Publish Message
11	0.012400096	192.168.1.115	192.168.1.110	MQTT	70	Publish Ack
12	0.017720777	192.168.1.110	192.168.1.115	MQTT	11…	Publish Message
13	0.018149469	192.168.1.115	192.168.1.110	MQTT	70	Publish Ack
14	0.023414413	192.168.1.110	192.168.1.115	MQTT	11…	Publish Message
15	0.023847305	192.168.1.115	192.168.1.110	MQTT	70	Publish Ack
16	0.028990627	192.168.1.110	192.168.1.115	MQTT	11…	Publish Message
17	0.029392411	192.168.1.115	192.168.1.110	MQTT	70	Publish Ack
18	0.034967937	192.168.1.110	192.168.1.115	MQTT	11…	Publish Message
19	0.035392508	192.168.1.115	192.168.1.110	MQTT	70	Publish Ack
20	0.040336948	192.168.1.110	192.168.1.115	MQTT	11…	Publish Message

Der Protokolloverhead für die `Publish-Message` Nachricht setzt sich auf QoS Level 1 wie folgt zusammen:

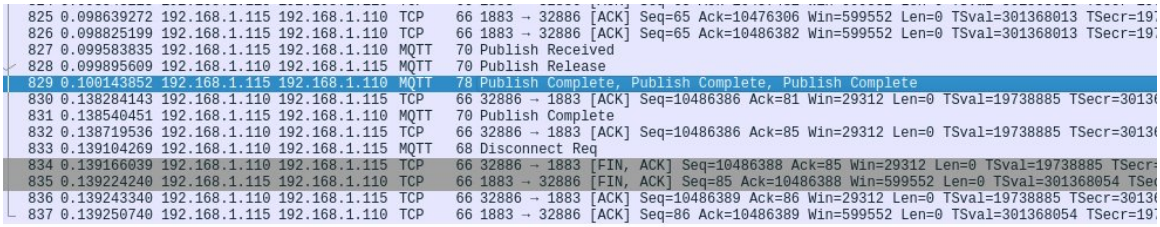
- ✓ 1 byte Header Flags
- ✓ 2 byte Message Length
- ✓ 10 byte Message ID
- ✓ 42 byte Topic Length (variiert mit dem gewählten Topic, bis zu 64KB)
- ✓ x byte Nutzlast (variiert mit der Nutzlast, bis zu 256MB)

Auf QoS Level 1 kommt das Nachrichten Feld `Message ID` neu hinzu, das verwendet wird um eine Nachricht zwischen Broker und Client eindeutig zu identifizieren.
Im Versuchsaufbau beträgt der Protokolloverhead damit 55 byte für die `Publish-Message` Nachricht plus 4 byte für die `Publish Ack` Nachricht. Der Gesamt Overhead beträgt damit 59 byte.

5.1.4 QoS 2 Overhead

Auf QoS Level 2 gibt es pro versendeter `Publish Message` Nachricht eine `Publish Received` Nachricht. `Publish Message` Nachrichten werden niemals gebundelt. Die `Publish Received` Nachricht ersetzt genauso wie die `Publish Ack` Nachricht von QoS 1 den `TCP ACK` und generiert 4 byte Overhead. Diese Nachricht wird nicht gebundelt. Zusätzlich gibt es noch (weiterhin pro versendeter `Publish-Message` Nachricht) eine `Publish Release` und eine `Publish Complete` Nachricht mit jeweils 4 byte Overhead. Diese beiden Nachrichtentypen werden bei großen Nutzdaten (im Versuchsaufbau ab 1 MB) ebenfalls gebundelt und können an andere Publishes oder den Disconnect angehängt werden. Dadurch wird low-level Netzwerkoverhead gespart und der Unterschied zu QoS 1 weniger signifikant. Dieser Effekt kann in Abbildung 5 in der blau markierten Zeile beobachtet werden. Ebenfalls zeigt sich das die einzige Nachricht die unmittelbar auf die `Publish-Message` Nachricht folgt `Publish Release` ist - die anderen beiden Nachrichten können mit zeitlichem Versatz eintreffen. So entsteht keine Wartezeit trotz des erhöhten Nachrichtenaufkommens.

Abbildung 5. Trace des QoS 2 Overhead



Der Protokolloverhead setzt sich bei QoS 2 genauso zusammen wie für QoS 1.Im Versuchsaufbau beträgt der Protokolloverhead damit 55 byte für die `Publish-Message` Nachricht plus 4 byte für `Publish Received` plus 4 byte für `Publish Release` und 4 byte für `Publish Complete`. Der Gesamt Overhead pro `Publish-Message` beträgt damit 67 byte.

6 Diskussion und Fazit

Im Zuge der Messungen hat sich gezeigt dass das verwendete Raspberry Pi 3 an seine Grenzen kommt. Beim Versand sehr vieler Publish Nachrichten in kurzer Zeit stürzt der Mosquitto Broker ab. Ist die Persistierung von Nachrichten zusätzlich aktiviert (was in der mitgelieferten Standardkonfiguration der Fall ist) beschleunigt sich dieser Prozess zusätzlich. Eine mögliche Erklärung ist, das mosquitto den verfügbaren RAM ausschöpft, wenn er sein Log nicht schnell genug wegschreiben kann weil das Speichermedium zu langsam ist. Aus diesem Grund sind sämtliche Messungen mit dem leistungsstärkeren Desktop Computer wiederholt worden. Dieses Verhalten zeigt allerdings auch, das die Wahl der richtigen Hardware beim Design eines MQTT Systems eine wichtige Rolle spielt - bei hohem Nachrichtenaufkommen muss Wert auf genügend RAM gelegt werden und schnelle Speichermedien. Sogar auf dem Desktop Computer ist es möglich den mosquitto Broker durch zu viele Nachrichten in sehr kurzer Zeit zum Absturz zu bringen, wenn auch erst nach deutlich längerer Sendezeit. Daher sind die Messungen auf maximal 100 Pakete/s begrenzt. Für zukünftige Versuche ist die Auswahl eines anderen Brokers eine Option und die Prüfung der maximalen Nachrichtenaufkommen.

Die Python Paho Library die zur Implementierung der beiden Client-Programme verwendet wird, zeigt Schwierigkeiten bei den Paketverlust Messungen. Das größte Problem beim Paketverlust ist nicht wie erwartet der Verlust einzelner MQTT Nachrichten, sondern der Verlust der Connection Pakete beim Versuch eines Verbindungsaufbaus. Die Client library kann hierbei beim Aufruf der Funktion `client.connect(<broker-ip>` intern endlos stecken bleiben oder mit der Fehlermeldung `Interrupted System Call` abbrechen. Vor allem der erste Fehler ist durch Error-Handling im Code schwer abzufangen. Dieses Problem lässt sich durch sorgfältige Fehlerprüfung und -abhandlung in den Griff bekommen aber da der Zeitraum der Messung begrenzt ist, wird dies nicht mehr vorgenommen. Die Wahl einer anderen Library und/oder einer anderen Programmiersprache zur implementierung kann auch Abhilfe schaffen.

Abgesehen von diesen technischen Schwierigkeiten zeigt sich, das MQTT ein resourcensparendes Protokoll ist. Durch das Message-Bundling (siehe Sektion Protokolloverhead) wird die Low-Level Overhead gering gehalten und zusätzlich ergeben sich ein paar interessante Anwendungen: So könnte man unter Verwendung

von QoS 0 und dem absetzen mehrere Publish-Nachrichten in kleiner zeitlicher Abfolge in einem Home Automation Szenario zeitgleich mehrere Lampen einschalten, weil die Publish-Nachrichten im selben TCP-Frame ausgeliefert werden. Insgesamt ist der beste Anwendungsfall für QoS 0 Nachrichten kleine Paketgrößen (z.B. 128 oder 256 byte), da diese sich gut in einem TCP Frame bundeln lassen und so das Protokoll beschleuningen. Ist es allerdings wichtig das kein Paket verloren geht oder die Leitung instabil, muss mindestens QoS Level 1 gewählt werden, da sonst das MQTT Protokoll den Verlust der Nachricht nicht bemerkt. Erhält der Client kein Publish Ack vom Broker sendet er das Paket erneut. Dies kann zu Duplikaten führen - wenn die Anwendung welche die Nachrichten empfängt damit nicht umgehen kann, ist QoS Level 2 zu wählen.

Wie schon in der Sektion Logging ausgeführt, ist MQTT ein anonymes Protokoll in Bezug auf die Clients untereinander. Ein sendender Client kann nicht davon ausgehen das außer dem Broker irgend ein anderer Client auf seine Nachrichten hört. Das Protokoll bringt auch keine Boardmittel zur Nachrichtenidentifikation mit. Somit können lauschende Clients weder feststellen von welchen Client eine Nachricht gekommen ist, noch bietet das Protokoll irgendwelche Garantien über die Reihenfolge eingehender Nachrichten. Die einzige Möglichkeit eine Nachrichtenreihenfolge zu garantieren ist, die Anzahl der Nachrichten die unterwegs sein können bevor neue Nachrichten versendet werden können auf 1 zu reduzieren. Dies hat allerdings Konsequenzen für die Geschwindigkeit. Wie drastisch sich diese Begrenzung auswirkt ist in weiteren Versuchen zu ermitteln.

Verliert ein Client die Verbindung zum Broker, gehen die Nachrichten verloren die bis zum Reconnect versendet werden. Das liegt daran, das in der Standard Einstellung Nachrichten vom Broker nicht aufbewahrt werden wenn Clients disconnecten. Daher gibt es bei einigen Messungen zu Paketverlust oder Bandbreitenbegrenzung irgendwann einen Abbruch an Messdaten - der Client wurde zwischendurch disconnected und anschließend gab es nichts mehr zu empfangen. MQTT bietet hierfür die Option der *Message Retention* und *persisten Sessions* - in diesem Fall werden Nachrichten aufbewahrt und wieder verschickt wenn die persistenten Clients wieder connecten. Den Einfluss dieser Option auf die RTT ist ebenfalls eine Option für weitere Versuche.