Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Evaluierung des Stromverbrauchs eines WLAN-basierten Systems im Internet der Dinge

von

Hendrik Sören Erik Linka
(9019554)

1. Prüfer:	Prof. Dr. Karl Jonas
2. Prüfer:	Prof. Dr. Stefan Böhmer
Studiengang:	Master Informatik
Eingereicht am:	23. Januar 2019

Persönliche Erklärung

Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Sankt Augustin, (23. Januar 2019) _____

(Hendrik Sören Erik Linka)

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

ACK Acknowledgment

ADC Analog-in-Digital-Wandler

AES Advanced Encryption Standard

API Application Programming Interface

AP Access Point

ASN.1 Abstract Syntax Notation One

CA Central Authority

CON Confirmable Message

CPU Central Processing Unit

CSV Comma-separated Values

CoAP Constrained Application Protocol

DER Distinguished Encoding Rules

DES Data Encryption Standard

DHCP Dynamic Host Configuration Protocol

DTLS Datagram Transport Layer Security

GPIO General Purpose Input Output

GPS Global Positioning System

HTTP Hypertext Transfer Protocol

I2C Inter-Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

ISO International Organization for Standardization

IdD Internet der Dinge

LDO Low-Dropout Regulator

LPWAN Low Power Wide Area Network

LSB Least Significant Bit

LoRaWAN Long Range Wide Area Network

LwIP Lightweight IP

M2M Machine-To-Machine

MQTT-SN MQTT For Sensor Networks

MQTT Message Queuing Telemetry Transport

MSB Most Significant Bit

MTU Maximum Transmission Unit

NON Non-confirmable Message

NVS Non-Volatile Memory

OTA Over-The-Air

PGP Pretty Good Privacy

PSK Pre-Shared Key

QFN Quad Flat No Leads Package

QoS Quality Of Service

RAM Random-Access Memory

RFC Request For Comments

ROM Read-only Memory

RST CoAP-Reset-Nachricht

RTC Real-Time Clock

SCTP Stream Control Transmission Protocol

SMP Symmetric Multiprocessing

SMS Short Message Service

SOT Small-Outline Transistor

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

SSID Service Set Identifier

TCP Transmission Control Protocol

TKL Token Length

TLS Transport Layer Security

UART Universal Asynchronous Receiver Transmitter

UDP User Datagram Protocol

URI Uniform Resource Identifier

USB Universal Serial Bus

VPN Virtual Private Network

WLAN Wireless Local Area Network

1 Einführung

1.1 Motivation

Im Internet der Dinge (IdD) werden batteriebetriebene Mikrocontroller, die unabhängig von einer ständigen Stromversorgung agieren können, immer wichtiger. Die Batterielaufzeit ist ein entscheidender Faktor für den Nutzen und Einsatz eines batteriebetriebenen Netzwerks aus Mikrocontrollern. Der Sinn eines solchen Netzwerks ist es, dem Nutzer oder dem System einen Mehrwert zu liefern. Dieser Mehrwert ist immer dem Aufwand gegenübergestellt, der von diesem Netzwerk erzeugt wird. Eine lange Betriebszeit ist wünschenswert. Der frühe Austausch von mobilen Stromversorgungen wie Batterien und Akkus ist hingegen ein zentraler Punkt, der gegen den Einsatz eines solchen Systems spricht.

Der Energieverbrauch eines Mikrocontrollers wird durch die verschiedenen Phasen beeinflusst, die der Mikrocontroller durchläuft. Die energiesparendste dieser Phasen nennt sich Tiefschlaf. Sie wird angewendet, wenn der Mikrocontroller sehr wenig Strom verbrauchen soll. Dabei wird er zeit- oder ereignisbasiert gesteuert. Tritt z. B. eine Abweichung bei einem Messwert auf, soll der Mikrocontroller erst beim Überschreiten des vorher definierten Schwellenwertes den Tiefschlaf verlassen und eine Nachricht senden. Dann tritt die zweite Phase ein. In dieser Phase wacht er auf und initialisiert alle notwendigen Systeme, die für eine Kommunikation benötigt werden. Anschließend baut er in der dritten Phase eine aktive Verbindung mit einem Server auf. Erst dann kann er in der vierten Phase eine Nachricht versenden.

Im besten Fall werden die Phasen zwei bis vier so kurz wie nötig ausgeführt. Vor allem das Senden und Empfangen von Nachrichten verbraucht viel Strom. Daher ist die Optimierung dieser Phasen von großer Bedeutung. Eine wichtige Rolle spielen dabei Protokolle. Sie definieren den Ablauf der Kommunikation und ermöglichen es dem Mikrocontroller erst, eine sichere oder zuverlässige Verbindung aufzubauen. Diese Kommunikation hat aber auch ihren Preis. Es müssen deutlich mehr Nachrichten ausgetauscht werden. Das Senden von Nachrichten verbraucht wiederum Strom, weshalb sich nun die Frage stellt wie viel diese Sicherheit und die Zuverlässigkeit kostet und inwieweit sie einen entsprechenden Preis besitzt, den der Nutzer bereit ist, zu zahlen.

1.2 Zielsetzung der Arbeit

Ziel dieser Arbeit ist die Untersuchung des Energieverbrauches eines Wireless Local Area Network (WLAN)-basierten Mikrocontrollers. Zum Einsatz kommt ein Espressif ESP32, der den kompletten WLAN-Protokollstack unterstützt. Die Frage, die geklärt werden soll, ist die Auswahl und der Preis unterschiedlicher Protokolle in Bezug auf den Stromverbrauch. Es soll dabei eine Übertragung von Sensordaten an einen Server simuliert werden. Die zu übertragende Nachrichtengröße beträgt hierbei 100 Byte bzw. 1000 Byte. Zudem sollen unterschiedliche Prozessoreinstellungen untersucht werden. In der späteren Auswertung wird anhand der Auswahl der richtigen Protokolle und Prozessoreinstellungen geklärt, ob ein Sensor mit einem WLAN-Protokollstack aufgrund seines Energieverbrauches für den Langzeiteinsatz als Sensor geeignet ist.

1.3 Aufbau der Arbeit

Die Arbeit ist in sieben Teile aufgeteilt. Im zweiten Kapitel wird der Stand der Forschung erläutert. Aktuelle Forschungsergebnisse werden gesammelt und analysiert. Ein besonderer Fokus wird auf die Messergebnisse und die Art, wie die Messergebnisse erhoben wurden, gelegt. Im dritten Kapitel werden der Aufbau und der Ablauf der verschiedenen Protokolle erklärt. Zusätzlich werden die Funktionsweise und der Aufbau des genutzten Mikrocontrollers erläutert. Anschließend wird in Kapitel vier eine Analyse der Erkenntnisse aus den ersten Teilen der Arbeit vorgenommen. Ziel ist die Entwicklung einer geeigneten Messmethode. Im fünften Kapitel wird der Versuchsplan aufgestellt, in dem alle Versuche und deren durchführungsweise aufgelistet werden. Hieran werden im nächsten Kapitel die Implementierung und die dabei entstandenen Probleme der einzelnen Protokolle sowie deren Versuchsaufbauten erläutert. Im Kapitel sieben werden die verschiedenen Ergebnisse untersucht und analysiert. Ziel ist die Abschätzung der Laufzeit von ESP32-basierten Systemen. Im letzten Kapitel werden die Ergebnisse resümiert und die zentrale Fragestellung dieser Arbeit anhand der Ergebnisse kontrovers diskutiert. Den Abschluss dieser Arbeit bilden das Fazit und der anschließende Ausblick.

2 Stand der Forschung

Während der Recherche zum Stand der Forschung wurden acht relevante wissenschaftliche Arbeiten ermittelt, die sich mit den Themen Energieverbrauch von Mikrocontrollern und den Protokollen Message Queuing Telemetry Transport (MQTT), MQTT For Sensor Networks (MQTT-SN) und Constrained Application Protocol (CoAP) in Bezug auf Übertragungsgeschwindigkeit und Stromverbrauch befassen.

Die Autoren in [37] zeigen eine allgemeine Herangehensweise für den Energieverbrauch eines IdD-Gerätes auf. Es werden zunächst die Klassen des Request For Comments (RFC) 7228 beschrieben. Diese beinhaltet die Klassen E0 „Event energy-limited“, E1 „Period energy-limited“, E2 „Liftetime energy-limited“und E9 „No direct quantitative limitations to available energy“ [10]. Zudem wird ein Schema mit fünf unterschiedlichen Klassifizierungen von IdD-Geräten erwähnt. Dieses beinhaltet eine Zeitangabe für die minimale Laufzeit. [16] Die Autoren bemängeln sowohl am RFC 7228 als auch am Schema, dass zwar eine Einteilung in Klassen vorgenommen wird, jedoch die Konstanten des Energieverbrauches nicht näher betrachtet werden. Dies würde der Analyse des Energieverbrauches eines IdD-Gerätes nicht weiterhelfen. Aus diesem Grund gehen die Autoren einen anderen Weg und teilen die verschiedenen Zustände eines IdD-Gerätes in die Zeit, in der das Gerät aktiv ist, und die Zeit, in der es schläft, ein. Zusätzlich wird eine Differenzierung von eventbasierten und zeitlich gesteuerten Phasenwechseln beschrieben. Die Autoren gehen davon aus, dass insbesondere Länge und Häufigkeit des Wechsels von Tiefschlafphasen zu Aktivphasen einen Einfluss auf den Gesamtverbrauch haben und als Messgröße dienen sollten.

In der Erhebung [5] werden Protokolle, die auf der Anwendungsschicht funktionieren, miteinander verglichen. Hierzu werden die Kategorien Latenz, Netzwerkauslastung, Stromverbrauch, Sicherheit und Wahl der Entwickler gebildet. Geeignete Untersuchungen zu diesen Themen wurden gesammelt und in einer Tabelle dargestellt. Die Auflistung zeigt, dass es schon zahlreiche Untersuchungen zum Thema MQTT und CoAP im Bereich Energieverbrauch gibt. Bei CoAP gibt es zudem auch viele Untersuchungen im Bereich Sicherheit. Dies ist bei MQTT nicht der Fall. Die Autoren kommen zu dem Schluss, dass im Bereich Energieverbrauch MQTT weniger Energie verbraucht als Hypertext Transfer Protocol (HTTP). Das Protokoll CoAP ist dabei noch effizienter als MQTT. Die Sicherheit der Protokolle wird mit Transport Layer Security (TLS) und Datagram Transport Layer Security (DTLS) realisiert. Die Autoren bemängeln, dass die Sicherheitsprotokolle TLS und DTLS nicht für beschränkte Geräte wie Mikrocontroller konzipiert wurden und zusätzlich Nachrichtenverkehr produzieren. Dabei ist TLS schneller als DTLS und birgt weniger Sicherheitslücken aufgrund seiner niedrigeren

Fehler-Toleranzschwelle. Insgesamt sehen die Autoren noch Handlungsbedarf bei der Optimierung der Sicherheit.

Eine vergleichbare Untersuchung wurde in [2] mit einem Android-Smartphone durchgeführt. Es zeigt den Anstieg des Energieverbrauches und die übertragene Datenmenge in Kombination mit zunehmenden Verlustraten. Aus der Untersuchung folgt, dass CoAP sowohl im Energieverbrauch als auch in der Datenmenge effizienter ist als MQTT Quality Of Service (QoS) 0 und MQTT QoS 2.

Die Untersuchungen in [4] zeigen, dass CoAP 10 % schneller ist als MQTT QoS 0. Wird MQTT QoS 1 verwendet, ist CoAP sogar 20 % schneller. Sobald ein Paketverlust von 20 % simuliert wird, ist der Paketverlust bei CoAP höher als bei MQTT QoS 1. Zusätzlich beschreiben die Autoren, dass CoAP durch sein Congestion Control mehr Zeit benötigt, um eine Nachricht zu übertragen, da es häufig auf eine verlorene Antwort des Servers wartet. Für den Versuchsaufbau wurde ein Android-Smartphone und ein Computer mit Windows 7 verwendet.

In der Arbeit [14] werden CoAP und MQTT-SN für Robotikanwendungen miteinander verglichen. Bei MQTT-SN handelt es sich um eine Weiterentwicklung von MQTT für energiesparende Geräte. Hierbei werden anders als bei MQTT keine Transmission Control Protocol (TCP)-Pakete, sondern User Datagram Protocol (UDP)-Pakete verwendet. Für die Untersuchung wurden mittels Raspberry Pi 10.000 Nachrichten mit MQTT-SN und CoAP versendet. Dabei wurde die Übertragungsdauer gemessen. Der Versuch ergab, dass MQTT-SN im Durchschnitt 30 % schneller ist als CoAP.

Die Autoren in [36] vergleichen den Energieverbrauch von MQTT und CoAP auf dem Espressif ESP8266. Hierzu wurde für MQTT eine TCP-Verbindung und für CoAP eine UDP-Verbindung genutzt. Es zeigte sich ein fast gleicher Energieverbrauch bei ca. 40 mA für die beiden Übertragungsraten. Die Abtastrate betrug dabei nur 333 ms. Die Versorgungsspannung wurde nicht angegeben. Zusätzlich wurde gezeigt, dass eine statische IP-Adresse den Verbrauch weiter senken kann. Diese Energieeinsparung nimmt mit der Anzahl der Nachrichten pro Stunde stetig zu.

In der Bachelorarbeit [20] wurde ein Espressif ESP32 genutzt, um die Latenzzeit und den Energieverbrauch von CoAP und MQTT miteinander zu vergleichen. Es wurde gezeigt, dass die unterschiedlichen QoS-Stufen von MQTT Einfluss auf die Latenz haben. Einen weiteren Effekt auf die Latenz hatten auftretende Interferenzen durch andere Funksender. Bei der Messung des Stromverbrauches traten Schwierigkeiten auf, da beim ESP32 die Stromstärke zwischen wenigen μA und mehreren Hundert mA schwankt. Die genutzten Geräte konnten diesen Bereich nicht erfassen.

Die Autoren von [19] haben den Einfluss von DTLS mit symmetrischen und asymmetrischen Schlüsseln unter dem Protokoll CoAP untersucht. Hierfür wurden u.a. der Energieverbrauch und die Zeit des Handshakes von DTLS gemessen. Die Autoren wiesen beim Test zum Energieverbrauch darauf hin, dass besonders in einem funkbasierten Netzwerk unkontrollierbare Effekte auftreten, die sie unter einer kabelgebundenen Verbindung nicht beobachten konnten. Deshalb haben sie sich dazu entschieden jegliche Netzwerkaktivität zu filtern und nur die arithmetischen Operationen zu messen. Die Messung wurde mit einer internen Funktion ausgeführt, die die Prozessortakte seitdem das Gerät gestartet wurde, angibt. Diese Werte wurden dann mit den Herstellerangaben zum Energieverbrauch multipliziert. Bei der Handshake-Messung wurde ein Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 Wireless Personal Area Network aufgebaut. Dieses Netzwerk hat eine kleine Maximum Transmission Unit (MTU) von 127 Bytes. IEEE 802.11 unterstützt hingegen eine MTU von 2312 Bytes. Diese kleine MTU kann schnell zur Fragmentierung der Pakete führen. Diese Fragmentierung führt laut Autoren zu einer langsameren Verarbeitungsgeschwindigkeit. Zur Belegung der These wurden die Paketgrößen beim Handshake von symmetrischen und asymmetrischen Schlüsseln gemessen. Das Ergebnis zeigte, dass unter asymmetrischen Schlüsseln größere Pakete verschickt werden müssen. Zudem findet bei dem niedrigen MTU-Wert von 127 Bytes, der eine maximale Größe der Applikationsdaten von 59 Bytes erlaubt, eine Fragmentierung der Pakete statt. Durch die an sich größeren Pakete beim asymmetrischen Verfahren und den fünf zusätzlichen Paketen, die für den Schlüsselaustausch notwendig waren, sind symmetrische Verfahren schneller beim Handshake als asymmetrische Verfahren.

Zusammenfassend lässt sich sagen, dass das Thema mit sehr unterschiedlichen Verfahrensweisen untersucht wurde. Am häufigsten wurde die Latenz vom Client zum Server gemessen. Hierbei zeigte sich, dass CoAP schneller ist als MQTT. Sobald es einen Paketverlust gibt, ändert sich dieses Bild. Inwiefern hier die Antwortzeiten des Servers eine Rolle spielen, wurde nicht in ausreichender Genauigkeit betrachtet. Eine genauere Analyse dazu, welcher Teil des Codes zu einer Verzögerung führt, ist deshalb sinnvoll. Weiterhin wurde die Energiemessung bei drei Arbeiten betrachtet. Ein Autor konnte die Nachricht nicht komplett aufnehmen und hat seine Ergebnisse deshalb als ungenau eingestuft. Nur in einer Arbeit wurde für einen ESP8266 der Energieverbrauch gemessen. Dieser wurde gemittelt angegeben und mit einer zu niedrigen Abtastrate durchgeführt. Spannungsspitzen konnten so übersehen werden. Der Aspekt der zusätzlichen Sicherheit wurde in [19] betrachtet.

Insgesamt ergibt sich der Eindruck, dass eine Untersuchung der Thematik in einer feineren Granularität durchgeführt werden sollte. Die Prüfung der unterschiedlichen wissenschaftlichen Veröffentlichungen hat kein einheitliches Bild zum Thema der Messung des

Energieverbrauches und dem Vergleich der Protokolle MQTT und CoAP ergeben. Überdies wurden unterschiedliche Bibliotheken verwendet. Sofern eine Einhaltung der RFC-spezifizierten Protokolle genutzt wurde, kann man die Arbeiten miteinander vergleichen. Jeder Mikrocontroller hat unterschiedliche Stärken aufgrund von z. B. hardwareseitiger Unterstützung von kryptografischen Operationen. Deshalb sollte die Thematik für den in dieser Arbeit zu untersuchenden Mikrocontroller erneut betrachtet werden. Die Erkenntnisse und Erfahrungen der wissenschaftlichen Arbeiten wurden mit in die Planung und Ausführung der neuen Messungen einbezogen. Insbesondere die genaue Messung der Zeit in den unterschiedlichen Phasen und die genauere Erfassung des Energieverbrauches, sofern dies möglich ist, sind dabei von Interesse.

3 Grundlagen

Zur Vorbereitung auf die Evaluation der verschiedenen Protokolle werden in diesem Kapitel die Grundlagen behandelt. Dabei werden insbesondere der Aufbau und die Besonderheiten der jeweiligen Protokolle aufgezeigt. Angefangen wird mit MQTT, MQTT-SN und CoAP. Im Fokus stehen die Teilnehmer der einzelnen Protokolle sowie die Netzwerktopologie. Eine genauere Darstellung des Aufbaus einer Nachricht wird ebenfalls betrachtet, um anschließend in einem Sequenzdiagramm den Ablauf des Nachrichtenverkehrs zwischen Server und Mikrocontroller zu analysieren. Daran schließen sich die Sicherheitsprotokolle TLS bzw. DTLS mit ihrem jeweiligen Ablauf an. Abschließend wird der in dieser Arbeit verwendete Mikrocontroller vorgestellt und seine Besonderheiten werden erklärt.

3.1 MQTT

Das Protokoll MQTT ermöglicht die Nachrichtenkommunikation von mehreren Clients zu einem Server. Es wurde speziell für den Einsatz im Machine-To-Machine (M2M) und IdD-Bereich entwickelt und bietet eine einfache Möglichkeit, Nachrichten auszutauschen. [23] Die aktuellste Version MQTT 5 wurde am 31.10.2018 veröffentlicht [24]. Aufgrund der noch nicht umgesetzten Broker- und Clientsoftware wird im Folgenden noch die Version 3.1.1 vom 29.10.2014 betrachtet [12].

3.1.1 Topologie

Die Rollen von MQTT sind sehr einfach gehalten. Das Protokoll setzt auf zwei Verfahren, die sich Publish und Subscribe nennen. Bei der Publish-Methode werden Nachrichten von einem Sender an den Server übermittelt. Der Server nimmt die Rolle eines Brokers ein und überträgt die Nachrichten an all jene die sich für den Empfang der Daten bereit erklärt haben. Die Erklärung nennt man Subscribe. Um nicht alle Nachrichten eines Gerätes zu empfangen, werden diese in einzelne Topics unterteilt. Das Topic muss eindeutig sein und darf auf dem Broker nur einmal vorkommen. [23]

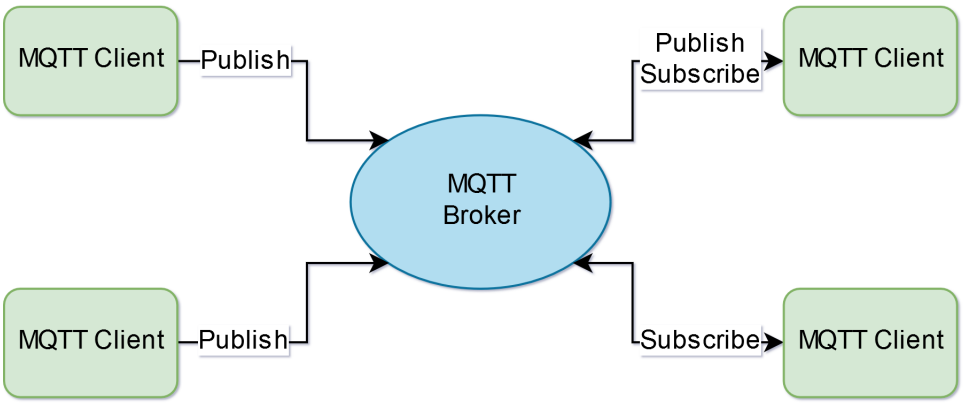


Abbildung 1: MQTT-Topologie. [23]

In der Abbildung 1 ist die Topologie von MQTT dargestellt. Der MQTT-Broker ist das Zentrum des Netzwerks. MQTT-Clients können sich mit ihm verbinden und Nachrichten schicken und empfangen, je nachdem ob sie Publisher, Subscriber oder beides sind.

3.1.2 Topics

Topics sind ein wichtiges Instrument in MQTT. Sie definieren einen Pfad, an den Nachrichten gesendet (PUBLISH) oder an dem Nachrichten empfangen (SUBSCRIBE) werden. Die Länge eines Topics muss mindestens einen Character lang sein. Die Topics sind zudem case sensitive. Die maximale Länge eines Topics ist auf 65536 Bytes begrenzt. Leerzeichen sind im Topic erlaubt. Nur der `NULL-Character` darf nicht enthalten sein. [23]

Ein Beispiel für einen Temperatursensor wäre demnach der Pfad:

```
1 /sensornetwork/sensor1/temperatur
2 /sensornetwork/sensor1/humidity
```

Abbildung 2: MQTT-Topic. [23]

Die verschiedenen Ebenen des Topics wurden mit dem Symbol / getrennt. Diese Trennung erlaubt den Aufbau einer Baumstruktur. Der Sensor würde wie in der dargestellten Struktur seine gemessene Temperatur an den ersten Pfad senden und seine Luftfeuchtigkeit an den zweiten Pfad. Alle Clients, die die Pfade durch einen Topic-Filter mitlesen und somit Subscriber sind, würden bei neuen Messwerten benachrichtigt werden.

Zum Abrufen eines Topics existieren Topic Filter. Für sie gilt dieselbe Semantik wie für Topics. Sie haben zusätzlich noch die Möglichkeit, Wildcards zu nutzen, um z. B. beide

Pfade zu empfangen. Dafür benötigt es nur ein Topic mit Wildcard. Die nachfolgenden Beispiele verdeutlichen dies. [23]

3.1.2.1 Multi-Level Wildcard

Das Symbol # wird als Multi-Level-Wildcard bezeichnet. Es selektiert die Eltern und Kinder des Pfades.

```
1 /sensornetwork/sensor1/#
```

Abbildung 3: MQTT-Topic Multi-Level-Wildcard Teil 1. [23]

In diesem Beispiel werden alle verfügbaren Werte des `sensor1` selektiert.

```
1 /sensornetwork/#/temperatur
```

Abbildung 4: MQTT-Topic Multi-Level-Wildcard Teil 2. [23]

Dieser Ausdruck hingegen selektiert in der Abbildung 4 die Temperatur aller Sensoren im `sensornetwork`.

3.1.2.2 Single-Level Wildcard

Neben der Multi-Level-Wildcard gibt es noch die Single-Level-Wildcard. Sie benutzt das Symbol +. Der Hauptunterschied ist die Selektierung von nur einem Level.

```
1 /sensornetwork/sensor1/temperature
2 /sensornetwork/sensor1/subsensor1/temperature
3 /sensornetwork/sensor1/subsensor2/temperature
```

Abbildung 5: MQTT-Topic Single-Level-Wildcard Teil 1. [23]

Das Beispiel zeigt einen Sensor, der zwei einzelne Temperaturwerte besitzt. Darüber hinaus existiert ein gemittelter Temperaturwert. Bei einer Selektierung des Sensors mit einer Multi-Level-Wildcard würden alle drei zurückgegeben werden.

```
1 /sensornetwork/+/temperatur
```

Abbildung 6: MQTT-Topic Single-Level Wildcard Teil 2. [23]

Mit der Single-Level-Wildcard wird nur der gemittelte Wert selektiert.

3.1.2.3 $-Topics

Der MQTT-Standard sieht noch ein Präfix für Systemnachrichten vor.

```
1 $SYS/server/port
```

Abbildung 7: MQTT-Topic Systemnachrichten. [23]

Dieses Präfix dient serverspezifischen Informationen oder Steruerungs-Application Programming Interface (API)s und darf vom Client nicht benutzt werden. Der Vorteil dieses Präfixes ist, dass es nicht durch Wildcards selektiert wird. [23]

3.1.3 Quality of Service

Die QoS-Stufe bestimmt die Abläufe, die beim Empfangen eines MQTT-Paketes zu absolvieren sind. Definiert sind die Stufen 0, 1 und 2. Bei der Stufe QoS 0 wird nach einer gesendeten Nachricht keine Bestätigung vom Empfänger zurückgeschickt. Die einzige Sicherheit, dass die Nachricht ankommt, wird von dem darunterliegenden TCP-Protokoll

gegeben. Dadurch wird bei einer verlorenen Nachricht auch kein erneutes Senden durch das MQTT-Protokoll veranlasst. Die QoS-Stufe 1 erhöht die Zuverlässigkeit, indem sie eine Bestätigung seitens des Empfängers der Nachricht fordert. Die Bestätigungsnachricht muss dabei dieselbe Identifikation haben wie die des empfangenen Paketes. Der Sender der Nachricht ist deshalb dazu verpflichtet eine unbenutzte Identifikationsnummer zu nutzen. Kommt keine Bestätigung, kann der Sender die Nachricht erneut senden. Er muss dann das DUP-Flag von 0 auf 1 setzen. Dies zeigt an, dass es sich um eine Retransmission handelt. [23]

Die höchste QoS-Stufe ist 2. Sie stellt die zuverlässigste Methode dar. Jede gesendete Nachricht soll genau einmal ankommen und es darf keine Duplikate geben. Dazu werden zwei Bestätigungsnachrichten genutzt. Die erste bestätigt den Empfang der Nachricht. Die zweite Nachricht bestätigt, dass alle Subscriber die Nachricht einmal empfangen haben. [23]

Ein tieferer Einblick in die verschiedenen QoS-Stufen wird später gegeben.

3.1.4 Flags

Das MQTT-Protokoll definiert zwei wichtige Flags, die für jeden Client, der eine Nachricht senden möchte, wichtig sind. Diese werden Retain- und Will-Flag genannt. [23]

Das Retain-Flag signalisiert dem Broker, dass er die letzte bekannte Nachricht eines Topics an alle neuen Subscriber zu versenden hat. Dies kann bei Messwerten helfen, die z. B. nur in mehrstündigen Abständen erhoben und gesendet werden. Die Subscriber erhalten somit immer den letzten Wert, unabhängig davon, wann sie sich angemeldet haben. [23]

Das Will-Flag kann beim Verbindungsaufbau vom Client gesetzt werden. Dieses definiert was geschehen soll, wenn der Client nicht mehr mit dem Server kommunizieren kann oder eine Verbindung ohne DISCONNECT-Nachricht beendet wurde. Das Will-Topic und die Will-Message werden dann mit dem spezifischen QoS und dem gewählten Retain-Flag verschickt. [23]

3.1.5 Aufbau einer Nachricht

Das MQTT-Protokoll nutzt verschiedene MQTT-Kontrollpakete zur Übertragung der Nachrichten. Die Nachricht ist in drei Teile aufgeteilt. Das Paket fängt mit einem Fixed Header an, der in allen Paketen enthalten ist. Anschließend folgt ein optionaler Variable Header und ein optionales Payload.

3.1.5.1 Fixed Header

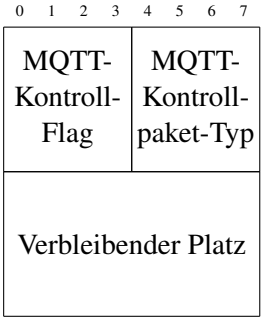


Abbildung 8: Aufbau MQTT-Nachricht. [23]

Der Fixed Header besteht aus drei Teilen. Der erste Teil definiert den Typ der Nachricht. Eine Übersicht über die Typen wird in Tabelle 1 gegeben.

Name	**Wert**	**Verbindungsrichtung**	**Beschreibung**
Reserviert	0	Verboten	Reserviert
CONNECT	1	Client -> Server	Verbindungsaufbau
CONNACK	2	Server -> Client	Verbindungs-Bestätigung
PUBLISH	3	Client <-> Server	Nachricht senden
PUBACK	4	Client <-> Server	Bestätigung QoS 1
PUBREC	5	Client <-> Server	Bestätigung QoS 2 Teil 1
PUBREL	6	Client <-> Server	Empfangen-Anfrage QoS 2
PUBCOMP	7	Client <-> Server	Bestätigung QoS 2 Teil 2
SUBSCRIBE	8	Client -> Server	Subscribe-Anfrage
SUBACK	9	Server -> Client	Subscribe-Bestätigung
UNSUBSCRIBE	10	Client -> Server	Unsubscribe-Anfrage
UNSUBACK	11	Server -> Client	Unsubscribe-Bestätigung
PINGREQ	12	Client -> Server	PING-Anfrage
PINGRESP	13	Server -> Client	PING-Antwort
DISCONNECT	14	Client -> Server	Verbindungsabbau
Reserviert	15	Verboten	Reserviert

Tabelle 1: MQTT-Pakettypen. [23]

Die Tabelle listet dabei den Namen der Methode, die dafür zu setzende Variable, den erlaubten Verbindungsweg und eine Beschreibung auf. Für MQTT 3.1.1 gibt es 16 verschiedene mögliche Werte, wobei 0 und 15 reserviert sind und nicht verwendet werden dürfen. Deshalb stehen dem Protokoll 14 verschiedene Typen zur Verfügung. Nach dem Typ der Nachricht folgt ein Flag. Es dient der näheren Beschreibung des Typs. Die Flags

werden nur von dem Typ PUBLISH genutzt. Bei allen anderen haben die 4 Bits des Typs den Wert 0. Zum Schluss wird noch der verbleibende Platz angegeben. Sie enthält die Anzahl der Bytes, die noch im selbigen Paket folgen. [23]

3.1.5.2 Variable Header

Der Variable Header wird nur in manchen MQTT-Kontrollpaketen verwendet. Seine Verwendung hängt vom Typ der Nachricht ab und dient der Identifizierung des Nachrichtenverkehrs zwischen zwei MQTT-Instanzen. Ist das Paket vom Typ PUBLISH QoS 1, PUBLISH QoS 2, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE oder UNSUBACK wird es eingebunden. Der Inhalt des Variable Header besteht aus 2 Byte, die das Paket identifizieren. Dabei nutzt das erste Byte die Bitwertigkeit Most Significant Bit (MSB) und das zweite Byte Least Significant Bit (LSB). [23]

3.1.5.3 Payload

Die MQTT-Pakete des Typs CONNECT, SUBSCRIBE, SUBACK und UNSUBSCRIBE benötigen zum Fixed und Variable Header noch einen Payload. Beim Typ PUBLISH ist die Verwendung eines Payloads optional. Die dabei eingebetteten Daten können zum einen Nutzdaten sein. Zum anderen werden verfahrensabhängige Elemente miteingefügt. Dies können bspw. beim Typ CONNECT der Username, das Passwort, das Will-Topic und die Will-Message sein. Eine Übersicht über die möglichen Elemente ist in der Tabelle 2 dargestellt. [23]

Methode	**Fixed H.**	**Variable H.**	**Payload**
CONNECT	Type	Protocol-Name, Protocol-Level, Connect-Flags, Keep-Alive	Client-Identifikator, Will-Topic, Will-Message, Username, Passwort
CONNACK	Type	Connect-Ack-Flags, Session-Present, Connect-Return-Code	-
PUBLISH	Type, DUP-Flag, QoS-Stufe, Retain	Topic-Name, Identifikator	Application Message
PUBACK	Typ	Identifikator	-
PUBREC	Typ	Identifikator	-
PUBREL	Typ	Identifikator	-
PUBCOMP	Typ	Identifikator	-
SUBSCRIBE	Typ	Identifikator	Topics mit QoS-Stufe
SUBACK	Typ	Identifikator	Return-Code-Topics
UNSUBSCRIBE	Typ	Identifikator	Topics
UNSUBACK	Typ	Identifikator	-
PINGREQ	Typ	-	-
PINGRESP	Typ	-	-
DISCONNECT	Typ	-	-

Tabelle 2: Aufbau MQTT-Pakete. [23]

3.1.6 Nachrichtenverkehr

Der Nachrichtenverkehr von MQTT richtet sich nach der verwendeten Methode. Ein Sensor, der Nachrichten an einen Broker sendet, nutzt hierfür die PUBLISH-Methode. [23]

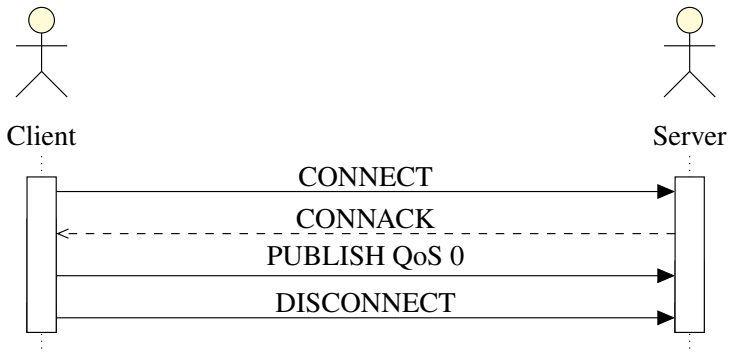


Abbildung 9: MQTT-Nachrichtenübermittlung mit QoS-Stufe 0. [23]

In der Abbildung 9 ist der Sende-Ablauf einer Nachricht mit der niedrigsten QoS-Stufe zu sehen. Der Client verbindet sich zunächst mit dem Server über eine CONNECT-Nachricht. Anschließend bestätigt der Server die Nachricht mit einem CONNACK. Sobald der Client die CONNACK-Nachricht erhält, kann er seine Daten mit einer PUBLISH-Nachricht übermitteln. Abschließend folgt eine DISCONNECT-Nachricht. [23]

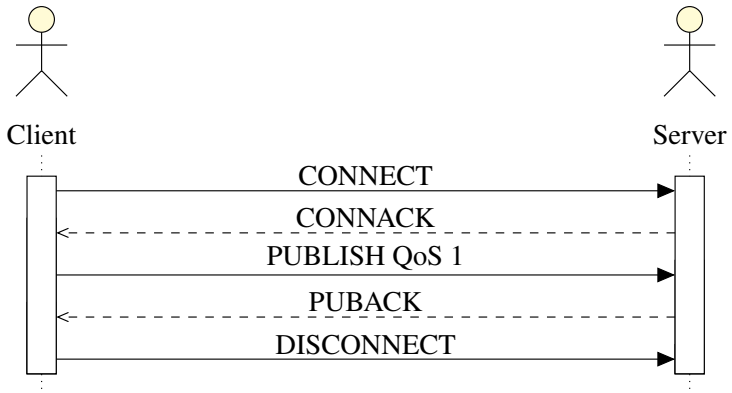


Abbildung 10: MQTT-Nachrichtenübermittlung mit QoS-Stufe 1. [23]

Sollte die Nachricht mit QoS 1 übertragen werden, muss diese wie in Abbildung 10 mit einem PUBACK vom Server bestätigt werden. Das PUBACK wird erst versendet, sobald der Server allen Subscribern die Nachricht vom Client übermittelt hat. Sollte der Client kein PUBACK empfangen, sendet er nach einer definierten Zeit eine weitere PUBLISH-Nachricht an den Broker. Diese Nachricht muss mit einem veränderten

DUP-Flag gekennzeichnet werden. Sobald das PUBACK vom Client empfangen wurde, sendet er eine DISCONNECT-Nachricht und geht in den Tiefschlaf. [23]

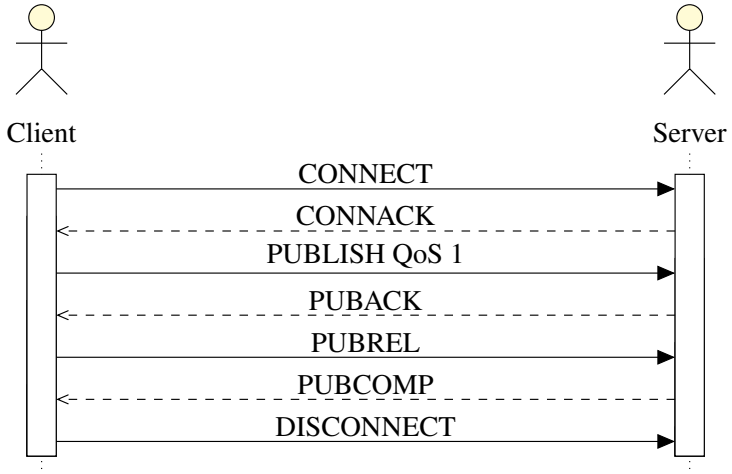


Abbildung 11: MQTT-Nachrichtenübermittlung mit QoS-Stufe 2. [23]

Eine Nachricht mit QoS 2 darf nur einmal beim Subscriber ankommen. Um dies zu garantieren, sendet der Client, wie in Abbildung 11 zu sehen ist, zunächst eine PUBLISH-Nachricht an den Server. Dieser bestätigt die Nachricht sofort mit einem PUBREC. Empfängt der Client das PUBREC nicht, sendet er erneut eine PUBLISH-Nachricht an den Server. Sollte der Client das PUBREC empfangen haben, fragt er beim Server an, ob auch alle Subscriber die Nachricht genau einmal bekommen haben. Der Server antwortet mit einem PUBCOMP wenn dieses Ereignis eintritt. Nach Empfang der PUBCOMP-Nachricht schickt der Client eine DISCONNECT-Nachricht und geht anschließend in den Tiefschlaf. [23]

3.1.7 Sicherheit

Die MQTT-Spezifikationen sehen keinen festen Standard für die Verschlüsselung von Nachrichten vor. Da MQTT auf dem TCP-Protokoll läuft, wird die Nutzung von TLS für die Verschlüsselung vorgeschlagen. Zudem besteht die Möglichkeit, einen Usernamen sowie ein Passwort in der CONNACK-Nachricht zu verschicken.

Eine weitere Möglichkeit ist der Aufbau einer Virtual Private Network (VPN)-Verbindung. Diese ermöglicht es, die Integrität und Authentifizierung der Nachrichten zu garantieren, ohne das MQTT-Protokoll zu verändern.

Für eingeschränkte Geräte wird die Portierung von Advanced Encryption Standard (AES) und Data Encryption Standard (DES) vorgeschlagen, wie sie in der International Organization for Standardization (ISO) 29192 spezifiziert wird [7]. Dabei wird keine

Ende-zu-Endeverschlüsselung aufgebaut. Nur der Payload wird verschlüsselt. Die restliche Kommunikation bleibt damit abhörbar und das Wiedereinspielen von alten Nachrichten ist möglich, sofern die Applikation dies nicht selbst verhindert. [23]

3.2 MQTT-SN

Das Protokoll MQTT-SN ist eine Abwandlung des originalen MQTT-Protokolls und setzt den Fokus auf batteriebetriebene Geräte mit limitierter Prozessorleistung und eingeschränkten Speicherressourcen. Die Abkürzung SN steht dabei für Sensor Networks. Das Besondere an der Abwandlung ist die Nutzung des UDP-Protokolls. Die erste Version von MQTT-SN wurde am 29.11.2007 spezifiziert. Mittlerweile existiert die dritte Fassung vom 20.05.2011. Der Hintergedanke bei der Entwicklung von MQTT-SN ist die Übertragung von wesentlichen Daten eines funkbasierten Sensors. Als Beispiel wird ein Global Positioning System (GPS)-Gerät genannt. Der Server und die spätere Applikation interessieren sich für die aktuelle Position des GPS-Gerätes. Die Netzwerkadresse wird jedoch nicht zwingend benötigt. Sie könnte durch eine kürzere ID ersetzt werden. Das Protokoll MQTT-SN setzt in vielen Bereichen auf verkürzte Namen und Adressen. Dies kann z. B. der Kommunikation vom Server über ein TCP/Internet Protocol (IP)-Netzwerk zu einem Sensor in einem Low Power Wide Area Network (LPWAN)-Netzwerk mit niedriger Datenrate dienen. Letztendlich können so mehr Nutzdaten in einer Nachricht übertragen werden. [1]

3.2.1 Topologie

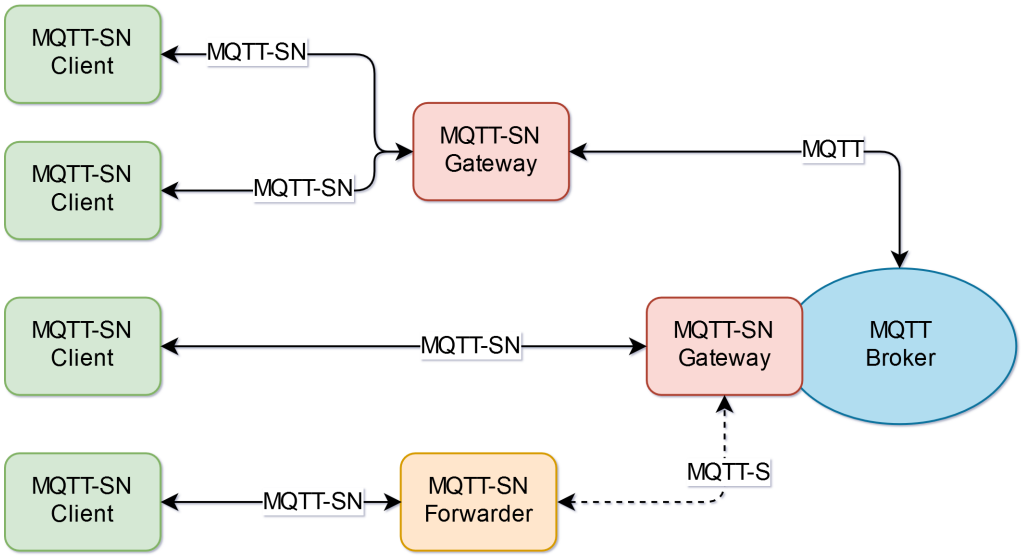


Abbildung 12: MQTT-SN-Architektur. [1]

Der Aufbau eines MQTT-SN-Netzwerks ist in der Abbildung 12 dargestellt. Clients haben die Möglichkeit, den MQTT-Broker über drei verschiedene Wege zu erreichen. Beim ersten Weg wird ein eigenständiges MQTT-SN-Gateway genutzt. Das Gateway übersetzt die Nachrichten von den Clients und leitet den Nachrichtenverkehr an den

MQTT-Broker weiter. Dieses antwortet über den gleichen Weg. Die Besonderheit von eigenständigen MQTT-SN-Gateways ist die Nutzung von MQTT zur Übertragung vom Gateway zum Broker.

Bei der zweiten Variante wird ein integriertes Gateway am Broker genutzt. Dieses übersetzt wie im ersten Fall die Nachrichten und dient den Clients als Schnittstelle.

Die letzte Möglichkeit ist die Nutzung eines MQTT-SN-Forwarders. Sie kapseln empfangene Client-Nachrichten und leiten sie an das Gateway weiter. Das Gateway kommuniziert im Anschluss über die Forwarder mit den Clients. [1]

3.2.2 Gateway

Das MQTT-SN-Gateway kann entweder transparent oder aggregierend sein. Ein transparentes Gateway leitet den Nachrichtenverkehr über eine individuelle Verbindung an den MQTT-Broker weiter. Es werden somit bei mehreren Clients mehrere MQTT-Verbindungen genutzt. Ein aggregierendes Gateway baut hingegen nur eine Verbindung zum MQTT-Broker auf. Der Nachrichtenverkehr der Clients wird über diese MQTT-Verbindung geleitet. [1]

3.2.3 Aufbau einer Nachricht

Eine MQTT-SN-Nachricht besteht aus zwei Teilen. Der erste Teil ist ein 2 bis 4 Byte großer Header. Der Header beinhaltet die Länge der Nachricht und deren Typ. An den Header schließt sich der zweite Teil an. Er wird Message Variable Part genannt. Seine Größe richtet sich nach der maximalen Paketgröße, die vom Netzwerk unterstützt wird, maximal jedoch 65535 Oktetts. Dies liegt der Tatsache zu Grunde, dass MQTT-SN keine Fragmentierung der Nachricht unterstützt. [1]

Name	Wert	Verbindungsrichtung	Beschreibung
0x00	ADVERTISE	GW -> Clients	Gateway Präsenznachricht
0x01	SEARCHGW	Client -> GW	Gatewaysuche
0x02	GWINFO	GW -> Client	Gatewayinfo
0x03	Reserviert	Verboten	Reserviert
0x04	CONNECT	Client -> GW	Verbindungsaufbau
0x05	CONNACK	GW -> Clients	Verbindungs-Bestätigung
0x06	WILLTOPICREQ	GW -> Client	Anfrage Will-Topic
0x07	WILLTOPIC	Client -> GW	Antwort Will-Topic
0x08	WILLMSGREQ	GW -> Client	Anfrage Will-Message
0x09	WILLMSG	Client -> GW	Antwort Will-Message
0x0A	REGISTER	Client <-> GW	Topic ID Registrieren/Informieren
0x0B	REGACK	Client <-> GW	Bestätigung Topic ID
0x0C	PUBLISH	Client <-> GW	Nachricht senden
0x0D	PUBACK	Client <-> GW	Bestätigung QoS 1
0x0E	PUBCOMP	Client <-> GW	Bestätigung QoS 2 Teil 1
0x0F	PUBREC	Client <-> GW	Empfangen-Anfrage QoS 2
0x10	PUBREL	Client <-> GW	Bestätigung QoS 2 Teil 2
0x11	Reserviert	Verboten	Reserviert
0x12	SUBSCRIBE	Client -> GW	Subscribe-Anfrage
0x13	SUBACK	GW -> Client	Subscribe-Bestätigung
0x14	UNSUBSCRIBE	Client -> GW	Unsubscribe-Anfrage
0x15	UNSUBACK	GW -> Client	Unsubscribe-Bestätigung
0x16	PINGREQ	Client -> GW Client -> Client	PING-Anfrage
0x17	PINGRESP	GW -> Client Client -> Client	PING-Antwort
0x18	DISCONNECT	Client -> GW	Verbindungsabbau
0x19	Reserviert	Verboten	Reserviert
0x1A	WILLTOPICUPD	Client -> GW	Update Will-Topic
0x1B	WILLTOPICRESP	GW -> Client	Bestätigung Update Will-Topic
0x1C	WILLMSGUPD	Client -> GW	Update Will-Message
0x1D	WILLMSGRESP	GW -> Client	Bestätigung Update Will-Message
0x1E-0xFD	Reserviert	Verboten	Reserviert
0xFE	Encapsulated message	Client -> Forwarder	Nachricht weiterleiten
0xFF	Reserviert	Verboten	Reserviert

Tabelle 3: MQTT-SN-Nachrichtentyp. [1]

MQTT-SN hat insgesamt 28 unterschiedliche Nachrichtentypen und damit mehr als MQTT. Deren Werte, Name und Bedeutung sind in der Tabelle 3 abgebildet.

Die Nachrichten für das Will-Topic und die Will-Message wurden in separate Pakete ausgelagert. Zudem existieren noch Möglichkeiten, diese upzudaten. [1]

3.2.4 Nachrichtenverkehr

Das Protokoll MQTT-SN hat insgesamt vier verschiedene QoS-Stufen. Somit existiert eine QoS-Stufe mehr als bei MQTT.

Die Besonderheit an QoS -1 ist das Auslassen einer CONNECT-Nachricht. Der Client sendet eine Nachricht an das Gateway, ohne zu wissen, ob die Adresse richtig ist. Diese Methode wird auch Fire-and-Forget genannt. [1]

Abbildung 13: MQTT-SN-Nachrichtenübermittlung mit QoS-Stufe -1. [1]

Die anderen QoS-Stufen erfordern eine CONNECT-Nachricht, bevor gesendet werden darf. In MQTT ist die Nachricht in nur einem Paket enthalten. Das Protokoll MQTT-SN hingegen nutzt mehrere einzelne Pakete.

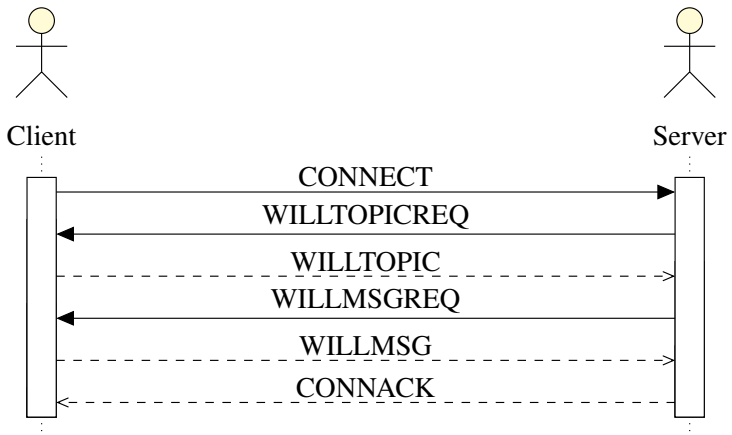


Abbildung 14: MQTT-SN-CONNECT-Nachricht. [1]

Sobald der Client verbunden ist, kann er eine Nachricht senden. Hierzu werden die gleichen Nachrichten-Abläufe wie in MQTT genutzt. Das erneute Senden einer Nachricht

überlässt das Protokoll der jeweiligen Implementierung. Es besteht keine Vorgabe nach wie vielen Sekunden eine Nachricht erneut versendet werden muss. [1]

3.2.5 Sicherheit

Die Spezifikationen sehen keine Verschlüsselung für das Protokoll vor. Da MQTT-SN UDP nutzt, könnte an eine Weiterentwicklung in Richtung IPsec oder DTLS gedacht werden. Bisher wurde dies aber noch nicht im RFC berücksichtigt, vielleicht wegen des Fokus auf beschränkte Geräte mit einer limitierten Netzwerkbandbreite. [1]

3.3 CoAP

Das CoAP wird im RFC 7252 der Internet Engineering Task Force (IETF) als spezielles Transportprotokoll für eingeschränkte Geräte und Netzwerke beschrieben. Es wurde für die M2M-Kommunikation entwickelt und soll als Kommunikationsweg für Netzwerke dienen, die eine niedrige Datenrate und hohe Verlustraten für Pakete vorweisen. Das Nachrichtenformat von CoAP setzt deshalb auf kompakte Pakete, die über UDP transportiert werden. Es wurden Mechanismen implementiert, die die Zuverlässigkeit der Kommunikation erhöhen, ohne den vollen Umfang von TCP nutzen zu müssen. Neben der üblichen Übertragung über UDP können auch Short Message Service (SMS), TCP und Stream Control Transmission Protocol (SCTP) verwendet werden. [8]

3.3.1 Topologie

Die Topologie von CoAP ist einfach aufgebaut. Es gibt einen Sender und einen Empfänger. Der Sender stellt eine Anfrage beim Empfänger, um eine Ressource zu erstellen, zu ändern oder zu löschen. Man spricht deshalb von einem Anfrage-Antwort-Modell zwischen zwei Endpunkten. [8]

3.3.2 Aufbau einer Nachricht

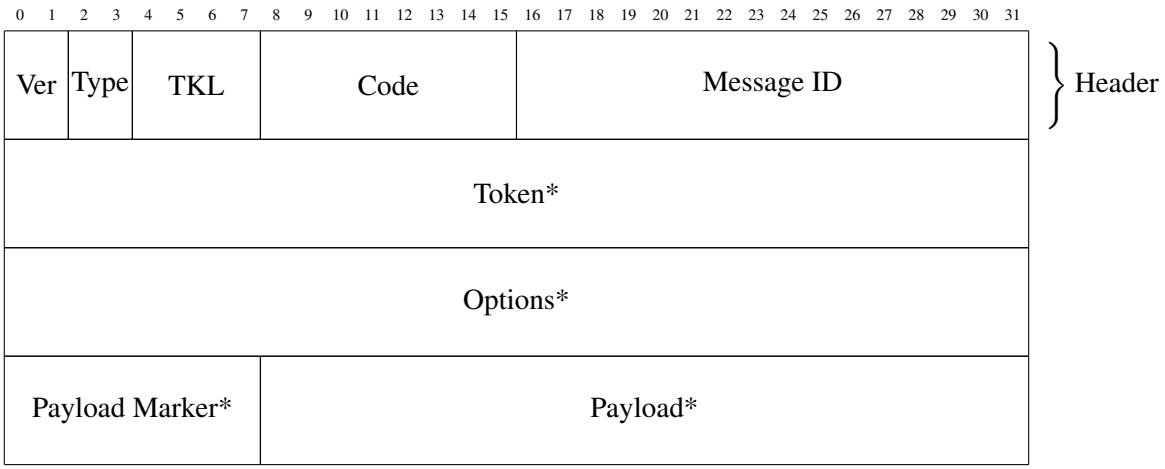


Abbildung 15: Aufbau einer CoAP-Nachricht. [8]

Eine CoAP-Nachricht besteht aus einem Header, einem Token, mehreren Optionen und einem optionalen Payload Marker mit anschließendem Payload. In diesem Unterkapitel werden die einzelnen Bestandteile einer CoAP-Nachricht, ihre Größe und der jeweilige Inhalt erläutert. [8]

3.3.2.1 Header

Die Nachricht beginnt mit einem 4 Byte großen Header. Dieser beinhaltet die CoAP-Version. Der Wert wird für RFC 7252-konforme Nachrichten auf 1 gesetzt. Andere Werte sind für spätere Versionen reserviert. Sollte eine nicht bekannte Versionsnummer benutzt werden, muss diese Nachricht verworfen werden.

Anschließend wird der Typ angegeben. Dieser bestimmt, um welche Art von Nachricht es sich handelt. Die vier unterschiedlichen Nachrichtentypen sind GET, PUT, POST und DELETE.

Der Wert Token Length (TKL) gibt die Länge des optionalen Tokens an. Maximal könnte ein Token 15 Byte groß sein. Das RFC beschränkt es auf 8 Byte. Alle Werte die darüber liegen werden als Formatfehler angesehen.

An dieser Stelle folgt der 8 Bit lange Code. Er wird unterteilt in 3 Bit MSB für die Code-Klasse und 5 Bit LSB für die Details. Die Code-Klasse gibt an, ob es sich um einen Request, eine Success-Response, eine Client-Error-Response oder eine Server-Error-Response handelt. Das Detail gibt weitere Auskunft über die Art der Anfrage oder der Antwort.

Abschließend folgt die Nachrichten-ID mit einem 16 Bit Unsigned-Integer. Der Wert ist in der Network Byte Order angeordnet, die dem Big-Endian-Format entspricht. Die Nachrichten-ID wird verwendet, um Nachrichtenduplikate zu identifizieren. Zudem dient sie der Zuordnung von Bestätigungs- und Resetnachrichten für CoAP-Confirmable Message (CON) und -Non-confirmable Message (NON). [8]

3.3.2.2 Token

Das Token kann zwischen 0 Byte und 8 Byte groß sein. Es wird für die Identifizierung der Antwort auf die Anfrage eines Clients verwendet. Deshalb muss es einzigartig sein und darf nicht mehrmals für dieselbe Anfrage auf einem Port verwendet werden. Die Nutzung des gleichen Tokens über unterschiedliche Ports ist hingegen erlaubt. Als Beispiel könnte ein Client eine Nachricht mit einem Token an zwei Server schicken, solange diese nicht dieselbe Adresse und denselben Port benutzen. Die Server sind daraufhin verpflichtet, mit demselben Token zu antworten. Das Token sollte bei ungeschützten Verbindungen zufällig und nicht trivial sein. Deshalb werden bis zu 8 Byte zur Verfügung gestellt, damit diese Vorgabe erfüllt werden kann. [8]

3.3.2.3 Optionen

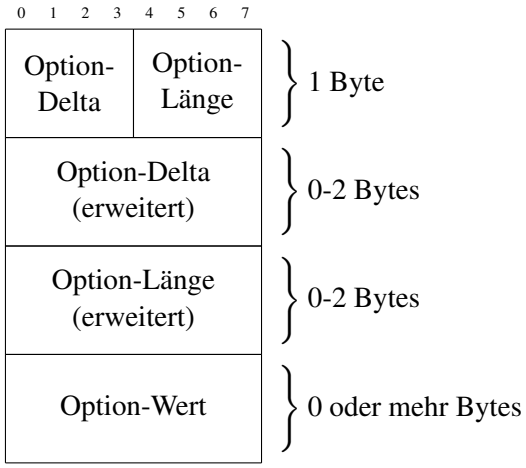


Abbildung 16: Aufbau einer CoAP-Option. [8]

Es existieren mehrere Optionen für eine CoAP-Nachricht. Jede angehängte Option enthält ihre Optionsnummer in einer Delta-Kodierung, ihre Länge und den eigentlichen Option-Wert. Das Option-Delta berechnet sich dabei aus der Differenz der Optionsnummer und der Optionsnummer aus der vorherigen Option. Handelt sich es um die erste Option einer CoAP-Nachricht wird kein Wert abgezogen. [8]

3.3.2.4 Payload Marker plus Payload

Das letzte Element einer CoAP-Nachricht ist der optionale 1 Byte lange Payload Marker mit nachfolgendem Payload. Der Payload Marker hat den Wert 0xFF. Dieser Wert zeigt das Ende der Optionen an und den Anfang des Payloads. Dabei wird darauf hingewiesen, dass das einfache Suchen von einem Byte mit 0xFF in einer CoAP-Nachricht nicht angewendet werden darf. Der Wert 0xFF kann u.a. auch in den Optionen genutzt werden. Deshalb muss eine Implementierung darauf achten, dass der Payload Marker nur auf eine Option folgen kann. Hiermit werden Verwechslungen ausgeschlossen. Weiterhin wird darauf aufmerksam gemacht, dass ein Payload Marker mit anschließendem leeren Payload als Message Format Error zu behandeln ist. [8]

3.3.3 URI-Schema

Das Uniform Resource Identifier (URI)-Schema definiert den Pfad, an dem eine bestimmte Ressource zu erreichen ist. Seine Zusammensetzung ist in der Abbildung 17 dargestellt. Der Wert für `path-abempty` kann entweder leer oder ein Pfad sein. Wird zusätzlich DTLS benutzt wird `coap` durch `coaps` ersetzt. [8]

```
1 coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?"
  ↪ query ]
```

Abbildung 17: CoAP-URI-Schema. [8]

3.3.4 Übertragung

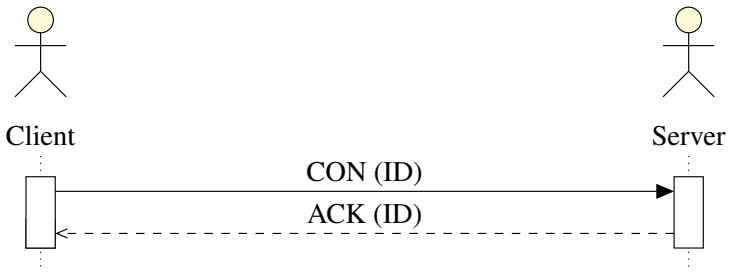


Abbildung 18: CoAP-CON-Übertragung. [8]

In der Abbildung 18 ist eine CoAP-CON veranschaulicht. Hierbei wird eine CoAP-Nachricht an den Server geschickt. Eine Acknowledgment (ACK)-Nachricht wird als Bestätigung zurückgeschickt. Sowohl die CON- als auch die ACK-Nachricht haben dieselbe ID. Sollte der Server die Nachricht vom Client nicht verarbeiten können, wird eine CoAP-Reset-Nachricht (RST)-Nachricht anstelle der ACK-Nachricht verschickt. [8]

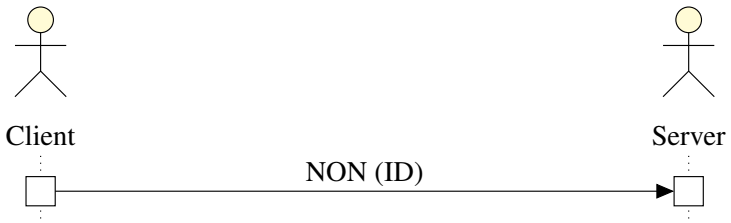


Abbildung 19: CoAP-NON-Übertragung. [8]

Bei einer CoAP-NON wird nur ein Paket übertragen. Die Abbildung 19 zeigt den Verbindungsablauf. Obwohl keine Antwort erwartet wird, hat diese Nachricht auch eine ID. [8]

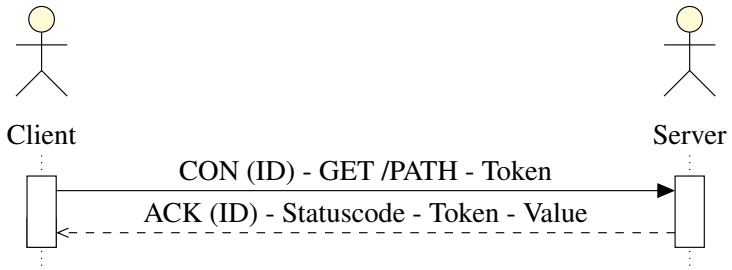


Abbildung 20: CoAP-Huckepack-Antwort. [8]

Neben dem gängigen Abschicken von Nachrichten nutzt CoAP zusätzlich das Anfrage-Antwort-Modell. Dabei werden CON/NON mit der Methode GET verschickt. Antwortet der Server sofort mit einer CON, die ein ACK enthält, spricht man von einer Huckepack-Antwort. [8]

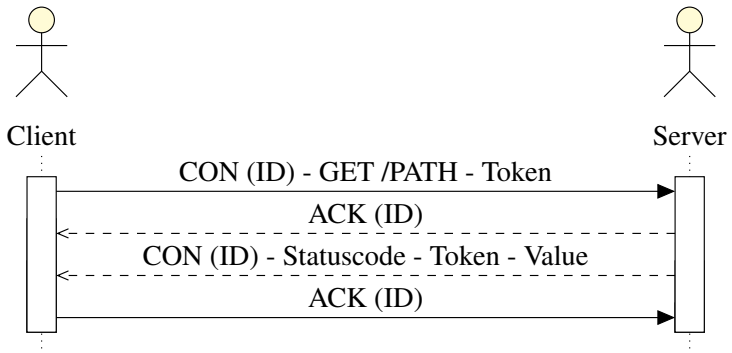


Abbildung 21: CoAP-Separate-Antwort. [8]

Daneben existiert noch die Möglichkeit, erst später zu antworten, z. B. weil der Server den Wert erst generieren muss. Dann wird sofort ein leeres ACK zurückgeschickt und später eine separate CON mit dem gewünschten Wert versendet. Die Methode wird separate Antwort genannt. [8]

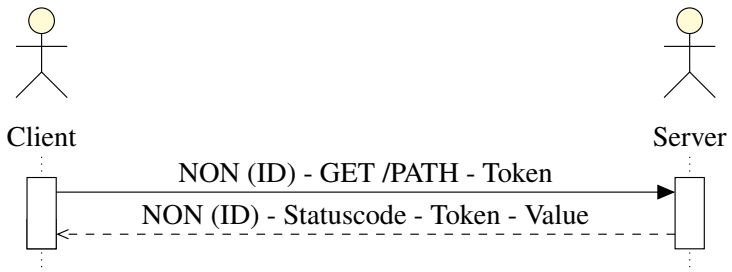


Abbildung 22: CoAP-NON-Antwort. [8]

Die letzte Möglichkeit ist die Nutzung des Anfrage-Antwort-Modells mit NON. Der Client verschickt eine Anfrage in einer NON an den Server. Dieser antwortet mit einer NON, die den entsprechenden Wert enthält. [8]

Sollte der Client keine Antwort erhalten, hat er die Möglichkeit nach einer definierten Zeit eine neue Anfrage zu stellen. Die Wartezeit setzt sich aus dem ACK-Timeout von 2 s und einem zufälligen Faktor, der größer als 1,0 ist, zusammen. Die Wartezeit kann durch die Anpassung der Parameter verändert werden. [8]

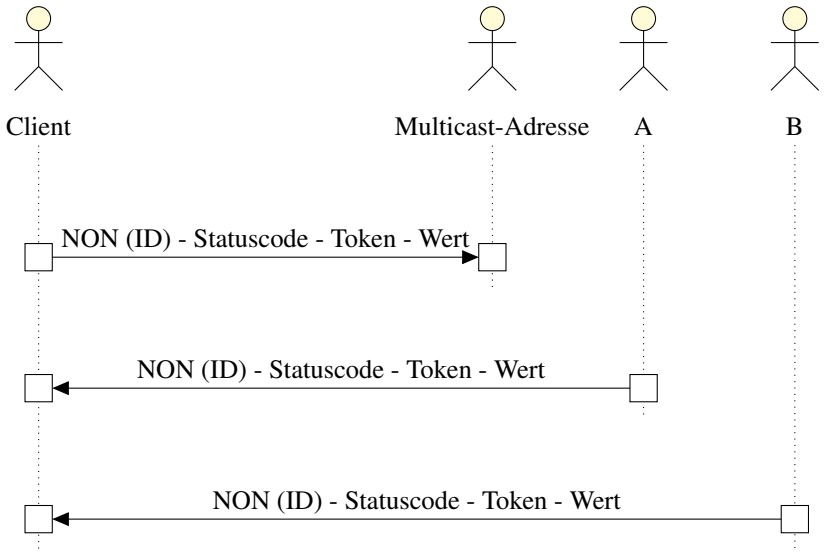


Abbildung 23: CoAP-Multicast-Antwort. [8]

Das Protokoll CoAP bietet die Möglichkeit, Anfragen an Multicast-Adressen zu versenden. Das RFC 7252 spezifiziert, dass dazu alle Empfänger, an die die Multicast-Nachricht gesendet wird, denselben Pfad besitzen. Die Empfänger senden ihre Antwort als NON an den Ursprungssender zurück. Der ganze Ablauf ist in der Abbildung 23 dargestellt. [8]

3.3.5 Sicherheit

Die Autoren des RFC beschreiben vier mögliche Methoden, um das Protokoll CoAP abzusichern. Die erste Methode nennt sich NoSec. Dabei ist DTLS deaktiviert und die Sicherheit wird in tieferen Schichten umgesetzt. Als Lösung wird IPsec vorgeschlagen. Hierzu existiert bereits ein Entwurf. In das Protokoll wurde er aber noch nicht eingebaut. [8]

Die zweite Methode nennt sich PreSharedKey. Hierbei wird eine DTLS-Verbindung aufgebaut mit bereits festen Schlüsseln. Diese werden in Listen verwaltet. Idealerweise bekommt jeder Endpunkt einen Schlüssel. Die gemeinsame Nutzung eines Schlüssels ist bei Gruppen erlaubt. Dann kann der Endpunkt jedoch nur noch als Mitglied der Gruppe authentifiziert werden. [8]

Die letzten beiden Methoden werden als RawPublicKey und Certificate bezeichnet. Bei beiden kommt ein asymmetrisches Schlüsselpaar zum Einsatz. Der große Unterschied zwischen der Certificate-Methode und der RawPublicKey-Methode ist die Signierung der Schlüssel durch eine vertrauensvolle Quelle. [8]

3.4 TLS

Das TLS-Protokoll hat das Ziel, eine sichere Verbindung zwischen zwei Instanzen aufzubauen. Das Protokoll will dabei für die Entwickler eine Schnittstelle schaffen, die Interoperabilität mitbringt. Dazu sollen neue kryptografische Verfahren in das bestehende Framework von TLS einbindbar sein, um die Notwendigkeit einer weiteren Bibliothek zu verhindern. [11]

Der Aufbau einer sicheren Verbindung zwischen einem Client und einem Server bedarf eines TLS-Handshakes. Er hat die Aufgabe die Aushandlung von Algorithmen und Schlüsseln zu übernehmen. Zusätzlich können durch ihn Teilnehmer authentifiziert und identifiziert werden. [11]

Der Verbindungsaufbau kann auf drei verschiedene Arten stattfinden.

1. Ein Full-Handshake mit Serverauthentifizierung
2. Ein Full-Handshake mit Server und Client-Authentifizierung
3. Ein Handshake der eine frühere Sitzung wieder aufnimmt

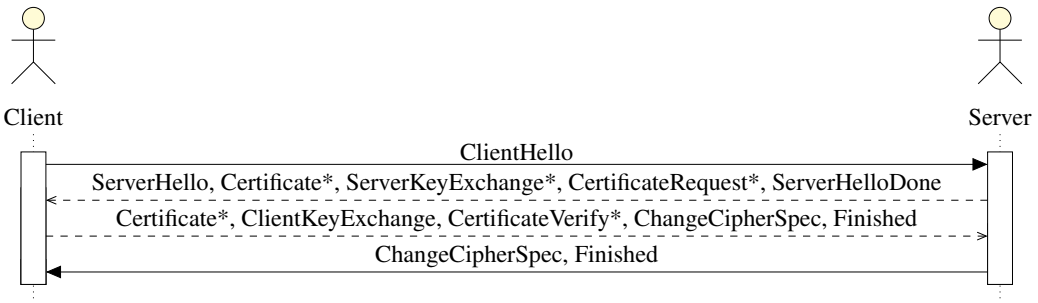


Abbildung 24: TLS-Full-Handshake. [11]

In der Abbildung 24 ist ein Full-Handshake mit Server- und Client-Authentifizierung dargestellt. Dieser beginnt mit einem ClientHello, in dem der Client dem Server seine Präferenzen mitteilt. Die Präferenzen enthalten die TLS-Version, den generierten Zufall mit einem Timestamp des Clients, eine Session-ID, die verwendbaren Cipher-Suites, eine Kompressionsart und mögliche Erweiterungen. [11]

Auf die ClientHello-Nachricht folgt vom Server aus eine ServerHello-Nachricht mit den gewählten Verbindungsparametern. Dazu zählen die Cipher-Suite und die Kompressionsart. Die Nachricht enthält wiederum den Zufall mit der Serverzeit und die Session-ID. Anschließend folgen drei optionale Elemente der Serverantwort. Dies ist zum einem das Zertifikat. Es enthält z. B. eine X.509-Zertifikatskette in Abstract Syntax Notation One (ASN.1) Distinguished Encoding Rules (DER) Kodierung. Die Zertifikate

müssen bei dem Prozess in der richtigen Reihenfolge versandt werden und zur ausgewählten Cipher-Suite passen. Es existieren Cipher-Suites die kein Certificate verwenden und somit keine Certificate-Nachricht benötigen. Auch wird die Nachricht von manchen Cipher-Suites für den Transport von Pretty Good Privacy (PGP)-Schlüsseln genutzt. [11]

Der darauffolgende ServerKeyExchange ist ebenfalls abhängig von der gewählten Cipher-Suite und kann weitere Informationen zum Schlüsselaustausch für das gewählte Verfahren enthalten. Sollten keine weiteren Daten gebraucht werden, entfällt er. Der letzte optionale Part ist der CertificateRequest. Er fordert vom Client ein Zertifikat an. Der Server beendet den Handshake mit einem ServerHelloDone und signaisiert dem Client die Bereitschaft zum Empfang des Schlüsselmaterials. [11]

Der Client übermittelt, falls dies angefordert wurde, zunächst das Zertifikat. Im Anschluss folgt der ClientKeyExchange. Dieser ist abhängig von der gewählten Cipher-Suite und überträgt für das Verfahren wichtige Information, wie z. B. das Premaster-Secret. Das CertificateVerify ist ebenfalls optional. Es verifiziert, dass der Client wirklich im Besitz des privaten Schlüssels ist. Den Abschluss der Nachricht bildet das ChangeCipherSpec. Es informiert den Server darüber, dass ab nun die Kommunikation mit den vorher festgelegten Schlüsseln startet. Das angehängte Finished bestätigt den Erfolg des Schlüsselaustausches und der Authentifizierung. [11]

Der Server antwortet ebenfalls mit einer Nachricht, die ChangeCipherSpec und Finished beinhaltet. Er bestätigt dadurch auch den Erfolg seinerseits und beginnt mit der verschlüsselten Kommunikation. [11]

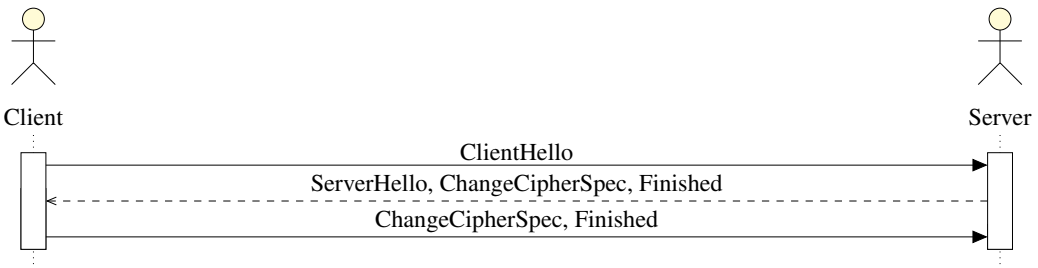


Abbildung 25: TLS-Session-Resumption. [11]

In der Abbildung 25 ist eine Wiederaufnahme einer Session dargestellt. In diesem Fall müssen deutlich weniger Parameter ausgetauscht werden. Der Sinn des Verfahrens ist es, nur einmal einen Full-Handshake durchführen zu müssen. Anschließend kann die gespeicherte Session wieder aufgebaut werden. [11]

3.5 DTLS

Das DTLS-Protokoll ist das Äquivalent zu TLS. Es hat deshalb die Philosophie einen Weg zu finden, um das TLS-Protokoll mit dem UDP-Protokoll zu vereinen. Beim UDP-Protokoll wird die Ankunft der Daten nicht sichergestellt. Dadurch entstehen zwei Probleme. Einerseits erlaubt es TLS nicht, Pakete in falscher Reihenfolge zu verarbeiten. Geht ein UDP-Paket verloren, fällt dies nicht auf. Erst wenn der Empfänger die Sequenznummer der Pakete vergleicht, fragt er nach dem fehlenden Paket. Durch diese Verzögerung kann es dazu kommen, dass ein späteres Paket zuerst beim Empfänger landet und die Reihenfolge der Pakete somit nicht stimmt. Die Integritätsprüfung von TLS würde die Verbindung verwerfen, sobald dieses Ereignis eintritt. [9]

Ein weiteres Problem ist das Handshake-Verfahren von TLS. Dieses setzt voraus, dass Handshake-Nachrichten zuverlässig zugestellt werden. Sollte ein Paket verloren gehen, wird der Vorgang abgebrochen. [9]

Für das Handshake-Problem nutzt DTLS einen einfachen Retransmission Timer. Sollte eine Nachricht beim Handshake verloren gehen, müssen Client oder Server diese nach einer bestimmten Zeit erneut senden. Bisher empfangene Nachrichten des Protokolls werden solange gespeichert bis die passende Nachricht empfangen wurde. Die empfangenen Nachrichten können später an passender Stelle benutzt werden und brauchen nicht erneut gesendet werden. [9]

```
1  struct {
2     ContentType type;
3     ProtocolVersion version;
4     uint16 epoch;                                  // New field
5     uint48 sequence_number;                        // New field
6     uint16 length;
7     opaque fragment[DTLSPlaintext.length];
8   } DTLSPlaintext;
```

Abbildung 26: DTLS-Record-Layer. [9]

Das Problem der Paketreihenfolge löst DTLS durch ein verändertes Record Layer. In der Abbildung 26 werden zwei neue Werte miteingebunden. Zum einen `epoch` mit dem initalen Wert von 0. Der Wert wird inkrementiert, sobald eine ChangeCipherSpec-Nachricht gesendet wurde. [9]

Der zweite Wert `sequence_number` inkrementiert sich pro Paket. Er ist zudem fest mit `epoch` verbunden. Wird `epoch` inkrementiert, fällt `sequence_number` zurück auf seinen initialen Wert von 0. [9]

Das Protokoll gibt vor, dass Pakete aus vorherigen `epoch`-Werten verworfen werden sollen. Andererseits erlaubt es bei zu großem Paketverlust, das Verwerfen von Paketen aufgrund des `epoch`-Werts zu unterlassen und die Pakete zu speichern und an passender Stelle zu benutzen. [9]

3.6 ESP32

Der ESP32 ist ein Mikrocontroller, der für die drahtlose und drahtgebundene Kommunikation im IdD-Umfeld entwickelt wurde. Er unterstützt 802.11 b/g/n/e/i und das IP. Zusätzlich werden auch Bluetooth 4.2 und Bluetooth Low Energy unterstützt. Die Basis des Controllers bilden die beiden Harvard Architecture Xtensa LX6 Central Processing Unit (CPU)s. Diese sind entweder mit 80, 160 oder 240 MHz getaktet. Es wird dabei standardmäßig von einer Trennung der beiden Kerne ausgegangen. Der erste Kern wird Pro_CPU genannt und soll die Initialisierung des Systems und die Steuerung der Protokolle übernehmen. Der zweite Kern wird als APP_CPU bezeichnet. Er verarbeitet die Befehle der Applikation. Eine feste Einteilung der Aufgaben wurde nicht vorgenommen, sodass Entwickler ihre Applikation jedem Kern frei zuordnen können.

Es existieren viele verschiedene Modelle vom ESP32. Bei allen beträgt der interne Speicher 448 KB für das Read-only Memory (ROM). Daneben existieren noch 520 KB Static Random-Access Memory (SRAM) und zwei Real-Time Clock (RTC)-Speicher mit 8 KB. Der Anschluss von einem externen Speichermedium wird mit bis zu 16 MB unterstützt. Die Größe variiert je nach Modell. [31]

Categories	Items	Specifications
Wi-Fi	Protocols	802.11 b/g/n (802.11n up to 150 Mbps)
		A-MPDU and A-MSDU aggregation and 0.4 μs guard interval support
	Frequency range	2.4 ~ 2.5 GHz
Bluetooth	Protocols	Bluetooth V4.2 BR/EDR and BLE specification
	Radio	NZIF receiver with –97 dBm sensitivity
		Class-1, class-2 and class-3 transmitter
		AFH
	Audio	CVSD and SBC
Hardware	Module interfaces	ADC, DAC, touch sensor, SD/SDIO/MMC Host Controller, SPI, SDIO/SPI Slave Controller, EMAC, motor PWM, LED PWM, UART, I^2C, I^2S, infrared remote controller, GPIO, pulse counter
	On-chip sensor	Hall sensor
	Integrated crystal	40 MHz crystal
	Integrated SPI flash	4 MB
	Operating voltage/Power supply	2.7 V ~ 3.6 V
	Operating current	Average: 80 mA
	Minimum current delivered by power supply	500 mA
	Operating temperature range	–40 °C ~ 85 °C
	Package size	(7.000±0.100) mm×(7.000±0.100) mm×(0.940±0.100) mm

Abbildung 27: Espressif ESP32 PICO D4 Spezifikation. [31]

In dieser Arbeit wird das Modell Pico D4 verwendet. Die Abbildung 27 stellt eine Übersicht der weiteren Funktionen des Modells dar. Hervorzuheben sind die verschiede-

nen Module Interfaces wie Analog-in-Digital-Wandler (ADC), Serial Peripheral Interface (SPI), Universal Asynchronous Receiver Transmitter (UART) und Inter-Integrated Circuit (I2C), die beim Bau eines Sensors eine zentrale Rolle spielen. Zusätzlich ist der bereits verbaute interne Speicher mit 4 MB und der interne Quarz mit 40 MHz angegeben. Das Einbetten dieser beiden Elemente verringert die benötigte Fläche auf einer Platine. [31]

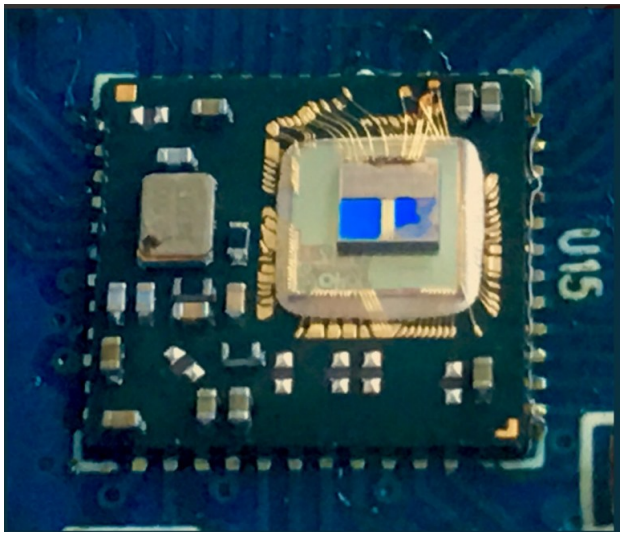


Abbildung 28: Unvergossener ESP32 PICO D4. [26]

In Abbildung 28 ist das Innenleben des Pico D4 dargestellt. Neben dem Chip auf der rechten Seite, ist links von ihm der Quarz positioniert. Ergänzend werden noch einige externe Kondensatoren mit untergebracht.

Die Firmware des ESP32 basiert auf einer modifizierten Version von FreeRTOS v8.2.0. FreeRTOS wurde für Geräte mit nur einem Kern entwickelt. Deshalb musste der Quellcode an die Gegebenheiten des ESP32 angepasst werden. Eine dieser Anpassungen ist die Unterstützung von Prozesszugehörigkeiten. Bisher wurden Prozesse immer auf dem ersten Prozessor ausgeführt. Ein neu eingeführter Parameter bei der Erstellung von Prozessen erlaubt es jedem Kern einen Prozess zu zuordnen. [29]

Zusätzlich zur Auswahl des Kerns unterstützt der ESP32 Symmetric Multiprocessing (SMP). Mit Hilfe des Rundlauf-Verfahrens werden Aufgaben durch die begrenzten Ressourcen des ESP32 effizient abgearbeitet. Dabei wird jeder Aufgabe eine Priorität und eine Prozessorzugehörigkeit zugewiesen. [29]

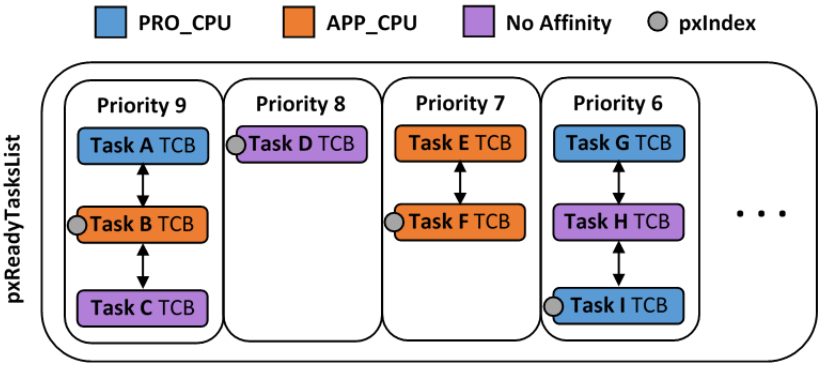


Abbildung 29: ESP-IDF Rundlauf-Verfahren. [29]

Die Prozesse in der Abbildung 29 werden je nach ihrer Farbe einer CPU zugeordnet. Der Wert pxIndex zeigt auf den gerade zu bearbeitenden Prozess. Für jede Priorität existiert genau ein pxIndex. Die Pro_CPU darf nur Aufgaben übernehmen, die die Farbe Blau oder Lila haben. Die APP_CPU nur diejenigen mit der Farbe Orange oder Lila. Sobald eine CPU den Prozess-Scheduler aufruft, wird ihr ein Prozess mit der passenden Farbe zugeordnet. Dabei besteht ein Problem, wenn zu viele Prozesse derselben Priorität zugeordnet wurden.

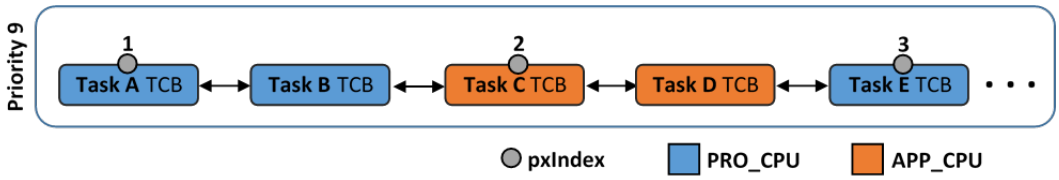


Abbildung 30: ESP-IDF Rundlauf-Verfahren Problem. [29]

In der Abbildung 30 bearbeitet die Pro_CPU den Prozess A. Die APP_CPU kann nicht den Prozess B übernehmen, da ihr dieser nicht zugeordnet wurde. Deshalb bearbeitet sie den Prozess C. Der pxIndex überspringt den Prozess B und landet direkt bei Prozess C. Der Prozess D kann nicht von der Pro_CPU bearbeitet werden. Deshalb geht er zu Prozess E. [29]

Durch schlechtes Setzen der Prioritäten kann es vorkommen, dass eine Anwendung, die beide Kerne nutzt, regelmäßig Prozesse überspringt [29].

Eine Lösung kann sein, den weiteren Ablauf zu blockieren, bis der aktuelle Prozess abgeschlossen wurde. Die bessere Alternative ist die Nutzung von vielen verschiedenen Prioritäten für jeden einzelnen Prozess. [29]

Für die Entwicklung von Applikationen wird die Entwicklungsumgebung ESP-IDF verwendet. Die Umgebung umfasst den gesamten Source-Code für die einzelnen Bestandteile des ESP32. Sie wird in einem offiziellen Github-Repository verwaltet und ständig von den Espressif-Entwicklern und der Community aktualisiert. [32]

4 Analyse

Die Aufarbeitung des Standes der Forschung zeigten drei mögliche Messmethoden. Im folgenden Abschnitt werden die Messmethoden diskutiert und es wird Stellung zur Umsetzbarkeit und Nutzbarkeit genommen. Im Anschluss findet eine Analyse der verschiedenen Phasen des Mikrocontrollers statt.

4.1 Messmethode

4.1.1 Nachrichtengröße

Jedes Protokoll definiert den Inhalt seiner Nachrichten und somit auch deren Größe. Die Messung der Nachrichtengröße kann helfen Erkenntnisse über die Fragmentierung und den Header eines Protokolls in einem verlustfreien Netzwerk zu sammeln. Zur Optimierung der Nutzung von Netzwerkressourcen eignet sich die Methode. Bei einem Mikrocontroller spielt neben der Größe der Nachricht auch die Zeit zur Bestätigung des Empfangs eine Rolle. Diese kann mit der bloßen Nachrichtengröße nicht erfasst werden, weshalb die Messmethode nur eine geringe Aussagekraft für das zu untersuchende Thema hat.

Die notwendigen Werkzeuge zur Erfassung der Nachrichtengröße sind vorhanden. Zum Einsatz kommt meist ein sogenannter Paket-Sniffer. Dieser listet Pakete und deren Größe auf. Die Installation und Einrichtung eines Paket-Sniffers, wie z. B. Wireshark, lassen sich auf fast jedem modernen System mit wenig Zeitaufwand realisieren.

Auch wenn die Nachrichtengröße nicht die richtige Wahl ist, um den Stromverbrauch zu ermitteln, können die eingesetzten Werkzeuge Verwendung beim Verstehen und Analysieren von Abläufen geben. Neben der Paketgröße werden z. B. auch die ungefähre Zeit und die Anzahl der ausgetauschten Pakete gespeichert.

4.1.2 Stromverbrauch

Die Messung des Stromverbrauchs ist mit geeigneten Werkzeugen die ideale Methode zur Untersuchung der Thematik. Durch die sehr großen Energieschwankungen des ESP32, von wenigen μA bis zu hunderten mA [31], war es jedoch schwer, ein Messinstrument zu finden, dass die notwendige Präzision besitzt. Deshalb wurde im Zuge der Arbeit der Fachbereich Elektrotechnik der Hochschule Bonn-Rhein-Sieg als beratende Stelle hinzugezogen. Die dort genutzten Messinstrumente bestanden aus einem sehr präzisen Shunt namens uCurrent und einem hochauflösenden Oszilloskop vom Hersteller

National Instruments. Das Messsystem heißt NI-CompactDAQ und nutzt ein NI9205-Modul. Dieses besitzt eine Auflösung von 16 Bit mit einer maximalen Abtastrate von 250 kS/s. [15]

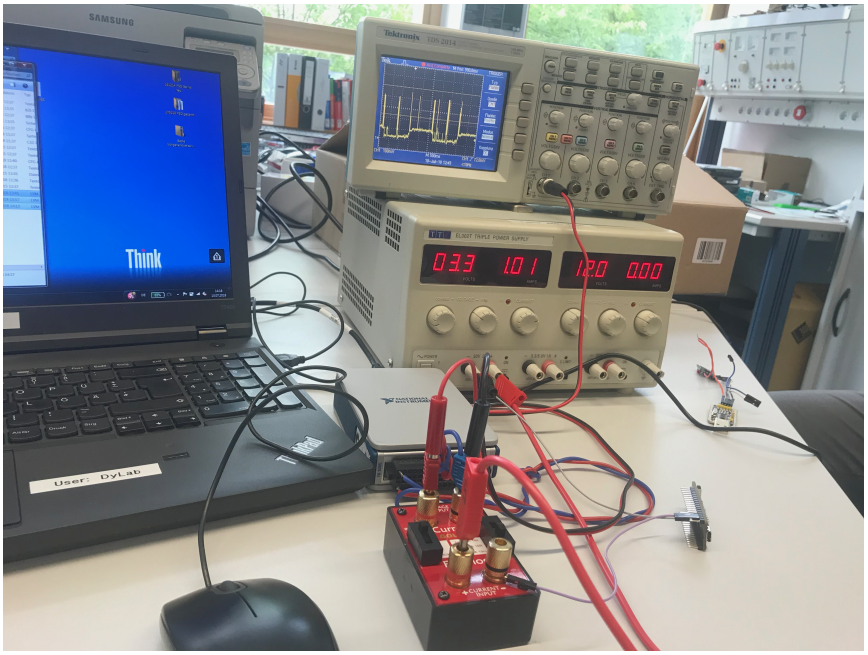


Abbildung 31: NI-CompactDAQ mit NI9205 Modul und uCurrent.

Während der Arbeit bot sich durch eine Neuanschaffung die Möglichkeit, einen Logikanalysator der Firma Saleae zu nutzen. Der Analysator trägt den Namen Logic Pro 16. Er bietet mit 12 Bit eine ähnlich hohe Auflösung wie die des Moduls aus dem Fachbereich Elektrotechnik. Der Vorteil dieser Neuanschaffung war die gesteigerte Abtastrate. Sie beträgt bei analogen Messungen bis zu 50 MSa/s. [27] Durch die großen Schwankungen im Energieverbrauch können Sende- und Empfangsspitzen mit diesem Messinstrument noch präziser gemessen werden. Es wurde weiterhin derselbe Shunt benutzt mit einer Genauigkeit von $\pm$ 0,1% im mA-Bereich und $\pm$ 0,05% im μA-Bereich [17]. Zusätzlich können durch den Logikanalysator noch General Purpose Input Output (GPIO)-Pins aufgezeichnet werden, die der Analyse von Verläufen von Stromverbräuchen dienlich sind [27].

4.1.3 Zeit

Eine Reihe wissenschaftlicher Arbeiten aus dem Stand der Forschung nutzen die Zeit als Messgröße. Für den Gebrauch der Zeit spricht zum einen die einfachere Durchführbarkeit. Einzelne Phasen der Protokolle können zeitlich genau erfasst und analysiert werden. Die Umsetzbarkeit hängt von der korrekten Auflösung der verstrichenen Zeit ab.

Der zu untersuchende Mikrocontroller besitzt einen sehr genauen Quarz mit 40 MHz und einer Frequenzabweichung von ± 10 *ppm* [6]. Geeignete Funktionen zum Abruf der Zeit sind ebenfalls vorhanden und können leicht in Programmabläufe implementiert werden [32].

Die Funktionen können jedoch nicht für alle Tests benutzt werden. Sie benötigen Routinen, die beim Booten des Mikrocontrollers noch nicht zur Verfügung stehen [32]. Aus diesem Grund nutzt die erste Methode das Setzen von digitalen Ausgängen. Ein externer Logikanalysator misst kontinuierlich die Spannung an einem oder mehreren digitalen Ausgängen. Sobald eine Veränderung auftritt, wird diese erkannt. Aus den Abständen zweier Veränderungen lässt sich anschließend die Zeit ermitteln. Wichtig war zunächst herauszufinden, ob das Setzen der Zeit mittels digitaler Ausgänge zusätzlich Zeit verbraucht. Für diesen Versuch wurde eine Verzögerung zwischen dem Setzen zweier digitaler Ausgänge eingeplant. Anschließend maß der Logikanalysator die Abstände und damit auch die Genauigkeit der Funktion. Die gesetzten Zeitabstände von 1 ms und 1 s wurden vom Logikanalysator bestätigt. Somit können Messungen in diesem Zeitbereich durchgeführt werden.

Die zweite Methode nutzt die internen Funktionen des ESP32. Es werden ein Start und ein Endpunkt festgelegt. Die Differenz beider bildet später die vergangene Zeit ab. Die Schwierigkeit bei dieser Methode ist die Nutzung geeigneter Funktionen, die keinen zu großen Overhead beim Ermitteln der Zeit generieren. Grundsätzlich kann hierzu ein internes Register genutzt werden. Dieses zählt die Takte des Prozessors. Mit der Taktfrequenz lässt sich anschließend die vergangene Zeit in μs errechnen. Das Problem des Registers ist der zu kleine Speicherplatz. Der Zähler wird in einer uint32_t gespeichert. Diese kann den Maximalwert von $2^{32}-1$ annehmen. Bei einer niedrigen Taktrate von 80 MHz ist eine Zeitmessung von maximal 53,68 s möglich. Steigt die Taktfrequenz auf 240 MHz an, sind es nur 17,89 s, bis der Zähler überläuft. Deshalb wird eine Funktion gebraucht, die das Überlaufen des Registers erkennt und entsprechend miteinberechnet. Das Espressif Framework bietet mit der Funktion esp_timer_get_time() die Möglichkeit, die vergangene Zeit seit dem Start in μs zu berechnen. Überläufe werden mitberücksichtigt und in einen int64_t gespeichert. Der maximale Wert ist $2^{63}-1$ μs.

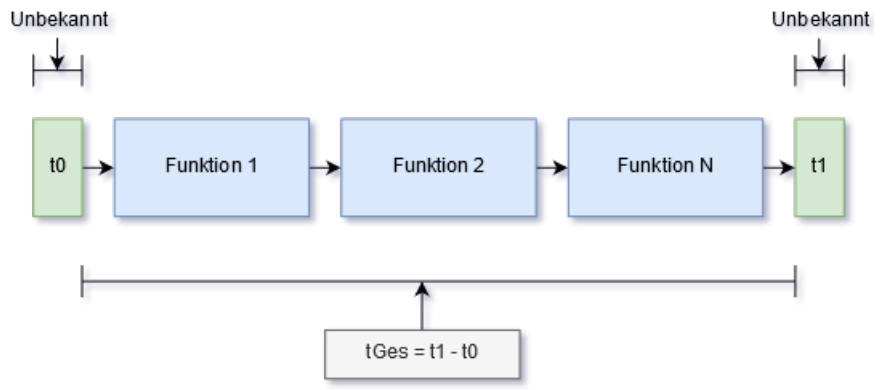


Abbildung 32: Zeitfunktion.

Die Abbildung 32 zeigt den Ablauf der Zeitmessung mit der internen Zeitfunktion. Gemessen werden sollte tGes. Die Funktionen t0 und t1 bilden den Start- und Endpunkt. Zum Testen der Genauigkeit wurde tGes mit einer definierten Verzögerung belegt. Anschließend wurde die Verzögerung von 1 ms bis zu 1000 ms gemessen. Dazu wurde diese je Iteration um 1 ms erhöht. Die spätere Analyse ergab eine Standardabweichung von 2 μs.

Die Messung der Zeit ist mit den genutzten Mitteln mit einer geeigneten Präzision gegeben. Zudem ist die Umsetzbarkeit der Methode einfach, sodass die Protokolle mit mehreren verschiedenen Faktoren leicht gemessen werden können.

4.1.4 Abwägung der richtigen Messmethode

Letztendlich muss eine Mischung aus allen verfügbaren Methoden gewählt werden. Die Messung der Größe und dem damit verbundenen Mithören der Verbindung dient der Überprüfung der korrekten Implementierung des Protokolls. Zudem ermöglicht es bei eigenen Portierungen, Fehler ausfindig zu machen. Die Zeitmessung mittels digitaler Ausgänge oder durch die interne Funktion des Mikrocontrollers erlaubt die Identifikation von schnellen und langsamen Protokollen. Die Ursachenfindung der Zeitdifferenzen kann anschließend durch die Strommessung weitergehend untersucht werden. Ausschläge beim Stromverbrauch können auf Sende- oder Empfangssituationen hinweisen.

4.2 Phasen des Mikrocontrollers

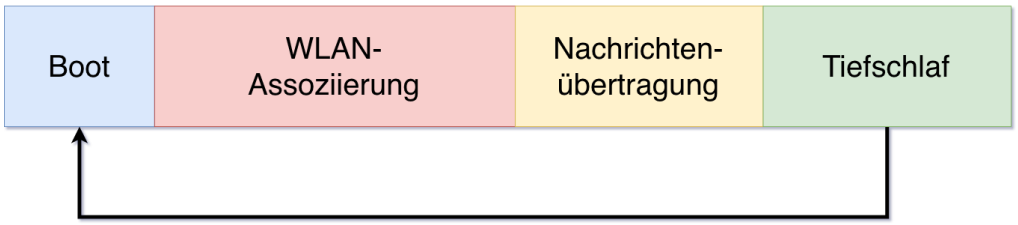


Abbildung 33: ESP32-Phasen.

Der ESP32 durchläuft während seines Betriebes unterschiedliche Phasen. In der Abbildung 33 sind die vier wichtigsten Phasen dargestellt.

Der Mikrocontroller beginnt seinen Betrieb in der Bootphase mit dem Starten der ersten Stufe seines Bootloaders. Dieser lädt die zweite Stufe des Bootloaders in den Random-Access Memory (RAM). Während dieser Phase läuft nur der erste Kern. Der zweite Kern wird nicht verwendet. Nachdem dieser Schritt erledigt ist, werden in der zweiten Stufe des Bootloaders Zwischenschritte durchgeführt. Hierzu zählen u.a. Flash-Encryption, Secure-Boot und Over-The-Air (OTA)-Updates. Der Quellcode der zweiten Bootloader-Stufe ist anders als der der ersten Bootloader-Stufe Open-Source. Dies erlaubt es eigene Anpassungen für z. B. ein eigenes OTA-Verfahren einzuarbeiten. Anschließend wird die auszuführende Applikation geladen und zum Einsprungpunkt gewechselt. Dieser Wechsel beendet die Bootphase. [33]

Die nächste Phase initialisiert den Non-Volatile Memory (NVS)-Speicher. Danach wird mittels der Funktion tcpip_adapter_init() ein Lightweight IP (LwIP)-Prozess gestartet. Der Prozess sendet automatisch eine Dynamic Host Configuration Protocol (DHCP)-Request-Nachricht aus, sollte bei der Konfiguration der DHCP-Client nicht gestoppt werden. Neben der Angabe der Service Set Identifier (SSID) und der Pre-Shared Key (PSK) des gewünschten Netzwerkes, kann auch der Channel vorgegeben werden. Sollte kein Kanal angegeben sein oder der angegebene Kanal ein Timeout erzeugen, wird von Kanal 1 bis 13 das WLAN-Spektrum nach Beacons durchsucht. Das Durchsuchen eines Kanals dauert dabei 30ms. Im besten Fall wird der Kanal sofort gefunden. Im schlechtesten verbraucht die Suche 13 mal 30 ms, also insgesamt 390 ms. [35]

Sobald eine Verbindung besteht und dem Mikrocontroller eine IP-Adresse zugewiesen wurde, setzt er ein Connected-Bit, das dem anschließenden Protokoll signalisiert, dass die Nachrichtenübertragung beginnen kann. [35]

In der dritten Phase wird die Nachricht übertragen. Hierzu werden TCP- oder UDP-Verbindungen aufgebaut und protokoll-spezifische Nachrichten ausgetauscht. Optional

wird die Verbindung vorher mit TLS oder DTLS verschlüsselt. In den späteren Implementierungen wird näher auf die einzelnen Besonderheiten der Protokolle eingegangen.

Die letzte Phase versetzt den Mikrocontroller in den Tiefschlaf. Entweder für eine bestimmte Zeit oder bis ein externes Ereignis ihn wieder aufweckt. Beim Aufwecken werden alle Systeme neu initialisiert. Bei der Initialisierung können alte Konfigurationen vom Flash genutzt werden [34]. So ist es möglich, nur einmal den geeigneten Kanal für ein Netzwerk zu suchen und diesen später erneut zu verwenden.

4.3 Protokolle

Zur Durchführung der Messungen müssen alle Protokolle getestet und implementiert werden. Das ESP-IDF bietet für zahlreiche Protokolle bereits vorgefertigte Komponenten. Die Tabelle 4 zeigt die bereits implementierten Protokolle an. Fehlende werden mit einem - gekennzeichnet.

Implementierung	**Name**	**Link**	**Sicherheit**
MQTT	ESP32 MQTT Library	[30]	TLS
MQTT-SN	-	-	-
CoAP	libcoap (16 Dec 2016)	[3]	-

Tabelle 4: Übersicht implementierte Protokolle.

Für MQTT bietet Espressif eine selbst verwaltete Bibliothek an. Diese basiert auf einer MQTT-Implementierung für Contiki. Sie wurde für den Einsatz auf dem ESP32 portiert und speziell angepasst. Die Unterstützung für mbedTLS ist bereits eingebaut und benötigt keine weiteren Arbeitsschritte. [30]

MQTT-SN wurde noch nicht für den ESP32 portiert. Es existieren schon zahlreiche Implementierungen für andere Plattformen. Deren Einbindung sollte durch den generischen IP-Stack möglich sein.

Für CoAP existiert bereits eine Umsetzung für den ESP32. Die verwendete Bibliothek libcoap ist auf dem Stand vom 16.12.2016. Mittlerweile existiert die zweite Version von libcoap. Bisher wird noch die erste Version für den ESP32 verwendet. Das Verschlüsselungsprotokoll DTLS wird von dieser Version nicht unterstützt [28].

5 Planung der Versuche

Die Erkenntnisse aus den Kapiteln Stand der Forschung, Grundlagen und Analyse fließen in die Planung der Messungen mit ein. In diesem Kapitel wird die Planung der durchzuführenden Tests beschrieben. Es wurden bei allen Versuchen das ESP-IDF in der Version vom 26.11.2018 verwendet.

5.1 Boot

Der Mikrocontroller bietet die Möglichkeit die Stufen der Log-Ausgabe der beiden Bootphasen zu beeinflussen. Zu untersuchen sind die Auswirkung der Ausgabe. Andere Einflüsse mit Auswirkungen auf die Laufzeit sollten ebenfalls mit in die Versuche einfließen.

5.2 WLAN-Assoziierung

Das Starten des TCP/IP-Stacks und die anschließende Assoziierung mit einem WLAN-Access Point (AP) verbraucht Zeit und Strom. Die Speicherung der Verbindungsdaten und das Setzen einer festen IP-Adresse können den Zeitbedarf senken. Zu untersuchen sind die Einstellung einer festen IP- bzw. die Anfrage einer IP-Adresse mittels DHCP-Anfrage.

5.3 Nachrichtenübertragung

Die Analyse der Nachrichtenübertragung ergab fehlende Funktionen bei den Protokollen MQTT-SN und CoAP. Nach der erfolgreichen Implementierung konnte ein Vergleich aller drei Protokolle unter verschiedenen Bedingungen stattfinden. Hierzu war geplant, den Test mit einem Payload von 100 Byte und 1000 Byte durchzuführen. Weiterhin wurde jedes Protokoll mit unterschiedlichen Prozessoreinstellungen getestet. Dazu gehörte der Vergleich vom niedrigsten Prozessortakt (80 MHz) mit dem höchsten (240 MHz). Zudem sollte die Auswirkung der Nutzung von zwei Kernen getestet werden. Insgesamt ergibt sich daraus die folgende Tabelle 5. Alle Teilversuche wurden mit jeder Möglichkeit getestet, die die Zuverlässigkeit beeinflusst. Bei MQTT sind das QoS 0, QoS 1 und QoS 2. Für CoAP existieren CON und NON. MQTT-SN unterstützt die Verfahren QoS -1, QoS 0, QoS 1 und QoS 2, jedoch softwarebedingt nur mit 100 Byte. Rechnet man alle Varianten zusammen, wird diese Versuchsreihe in 96 Einzelversuche unterteilt.

Payload	**MQTT**	**MQTT-SN**	**CoAP**	**MQTT TLS**	**CoAP DTLS**
100	80 MHz, 1 Kern	80 MHz, 1 Kern	80 MHz, 1 Kern	80 MHz, 1 Kern	80 MHz, 1 Kern
100	80 MHz, 2 Kerne	80 MHz, 2 Kerne	80 MHz, 2 Kerne	80 MHz, 2 Kerne	80 MHz, 2 Kerne
100	240 MHz, 1 Kern	240 MHz, 1 Kern	240 MHz, 1 Kern	240 MHz, 1 Kern	240 MHz, 1 Kern
100	240 MHz, 2 Kerne	240 MHz, 2 Kerne	240 MHz, 2 Kerne	240 MHz, 2 Kerne	240 MHz, 2 Kerne
1k	80 MHz, 1 Kern	-	80 MHz, 1 Kern	80 MHz, 1 Kern	80 MHz, 1 Kern
1k	80 MHz, 2 Kerne	-	80 MHz, 2 Kerne	80 MHz, 2 Kerne	80 MHz, 2 Kerne
1k	240 MHz, 1 Kern	-	240 MHz, 1 Kern	240 MHz, 1 Kern	240 MHz, 1 Kern
1k	240 MHz, 2 Kerne	-	240 MHz, 2 Kerne	240 MHz, 2 Kerne	240 MHz, 2 Kerne

Tabelle 5: Nachrichtenübertragungsplan.

5.4 Energieverbrauch Idle/Last

Die zwei Prozessorkerne können auf Taktraten von 80, 160 und 240 MHz gesetzt werden. Der Einfluss der Taktrate auf den Stromverbrauch ist eine wichtige Information für die Auswahl des richtigen Protokolls. Sollte das Protokoll nur mit einer hohen Taktfrequenz effizient arbeiten können, stellt sich die Frage, wie viel die Mehrleistung kostet.

Der Test wurde in zwei Stadien durchgeführt. Zum einen wurde eine Endlosschleife erzeugt, in der die Prozessorkerne keine Rechenoperationen durchführen mussten. Auf diese Weise konnte der Grundverbrauch im Idle-Modus ermittelt werden. Zum anderen musste der Verbrauch unter Last geprüft werden. Hierzu wurden Rechenoperationen genutzt, die der Prozessor in einer Endlosschleife lösen musste. Ein Oszilloskop zeichnete für die spätere Analyse den verbrauchten Strom auf.

6 Implementierung

In diesem Kapitel liegt der Fokus auf der Implementierung, dem Aufbau und der Versuchsdurchführung der einzelnen Messungen. Beschrieben werden dabei die Schritte, die durchgeführt wurden.

6.1 Entfernung von Komponenten

Das Entwicklerboard, mit dem zu untersuchenden Mikrocontroller ESP32 Pico D4, besitzt Hardware-Komponenten, die die Stromverbrauchs-Messergebnisse verfälschen würden. Hierbei handelt es sich um den seriellen Wandler CP2104 und den Spannungswandler AMS1117. Der CP2104 verbraucht im normalen Modus gemäß Datenblatt zusätzlich 17 bis 18,5 mA bei einer Eingangsspannung von 3,0 bis 3,6 V. Der Low-Dropout Regulator (LDO) AMS1117 hat daneben noch einen Ruhestrom von 5 bis 11 mA, wenn die Differenz zwischen Eingangsspannung und Ausgangsspannung 1,5V beträgt. Da diese Werte je nach Einstellung variieren und nicht in den Stromverbrauch des Mikrocontrollers mit eingerechnet werden sollten, mussten diese entfernt werden. Der Spannungswandler AMS1117 ist im Small-Outline Transistor (SOT)-23 Format und ließ sich auslöten. Der CP2104 hat das Format Quad Flat No Leads Package (QFN)24. Der Chip ließ sich nicht durch das Erhitzen der Pads entfernen, ohne Nachbarkomponenten gegebenenfalls zu beschädigen. Deshalb wurde er durch mechanische Einwirkungen einer Zange und eines Skalpells vorsichtig entfernt.

Zusätzlich zum seriellen Chipsatz und dem Spannungswandler wurde die Antenne ausgebaut. Am Antennenausgang des Mikrocontrollers wurde ein Pigtail angelötet. Dieser verbindet die beiden Transceiver über zwei Dämpfungsglieder mit 20 dB Dämpfung. Dies dient der Verbesserung der Übertragung und soll äußere Einflüsse minimieren.

6.2 Störquellen

Beim Testen des Aufbaus wurden bei verschiedenen Versuchen unerklärliche Messungen bemerkt. Deshalb wurde eine Suche nach Störquellen durchgeführt.

Wichtige Elemente, die vor den Tests ausgetauscht wurden, waren der zu benutzende Computer und die Stromquelle des Mikrocontrollers. Bei Messungen mit einem Computer konnten im Oszilloskop Spitzen erkannt werden. Diese verschwanden teilweise durch den Einsatz eines akkubetriebenen Computers. Auch bei der Stromversorgung des Mikrocontrollers wurde von einem 230 V Netzteil auf einen Akku umgestiegen.

Die Spitzen verschwanden darauf hin und ließen sich somit der dem Problem mit einer Brummschleife zuordnen.

Zusätzlich zum Wechsel auf akkubetriebene Geräte ist die Wahl des richtigen WLAN-Kanals eine große Fehlerquelle. Trotz direkter Verbindung der Transceiver und schwacher fremder WLAN-Signale, sollte immer ein Kanal verwendet werden, in dem kein anderes bekanntes Netzwerk aktiv sendet. Da an stark frequentierten Orten wie der Hochschule Bonn-Rhein-Sieg die Anzahl der WLAN-AP und der Clients es nicht erlaubte, einen freien Kanal zu finden, wurde dies in privaten Räumlichkeiten durchgeführt. Die störungsfreie Übertragung wurde mittels WLAN-Sniffer überprüft. Fortan kam es auch zu keinen weiteren Störungen bei der Übertragung.

Zum Abschluss wurde zwischen der 3,3 V-Stromversorgung und GND noch ein Kondensator eingefügt, um Stromschwankungen auszugleichen. Zudem wurde darauf geachtet, alle Kabel so kurz wie möglich zu halten.

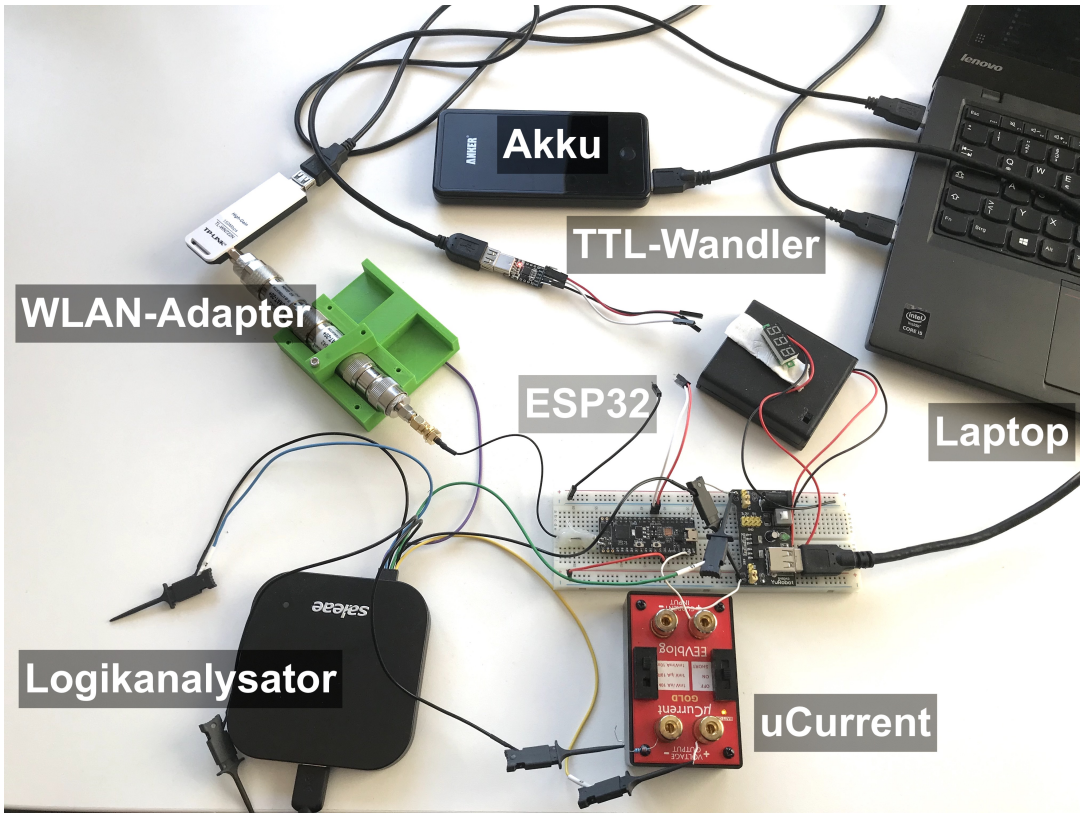


Abbildung 34: Messaufbau.

Die Abbildung 34 stellt den Messaufbau mit allen Umbauten dar.

6.3 Zeitmessung

Die Zeitmessung wird mit Hilfe von zwei Methoden ermittelt. Die erste Methode nutzt das Setzen von GPIO-Pins zur Bestimmung eines Zeitintervalls. Für die Bedienung der GPIO-Schnittstelle kann die interne Funktion des ESP-IDF genutzt werden. Diese wählt zunächst einen geeigneten GPIO-Pin aus. Anschließend wird bestimmt, ob der GPIO-Pin ein Aus- oder ein Eingang ist. Zum Schluss kann der Pegel des GPIO-Pins gesetzt werden, sofern er als Ausgang deklariert wurde.

```
1 gpio_pad_select_gpio(gpio_num);
2 gpio_set_direction(gpio_num, GPIO_MODE_OUTPUT);
3 gpio_set_level(gpio_num, 1); // HIGH
4 gpio_set_level(gpio_num, 0); // LOW
```

Abbildung 35: Ansteuerung der GPIO-Schnittstelle Teil 1. [32]

Eine Analyse der Funktion gpio_set_level() ergab eine weitere IF-Klausel. Diese dient der Prüfung, ob es sich um einen reinen digitalen Eingang oder einen digitalen Ein- und Ausgang handelt. Für den Test wurde die Routine umgangen und nur GPIO-Pins verwendet, die als Ausgang genutzt werden können.

Die folgenden beiden Befehle setzen bspw. den benötigten Pegel des GPIO_21 und umgehen die zusätzliche Abfrage.

```
1 gpio_pad_select_gpio(gpio_num);
2 gpio_set_direction(gpio_num, GPIO_MODE_OUTPUT);
3 GPIO.out_w1ts = (1 << gpio_num); //HIGH
4 GPIO.out_w1tc = (1 << gpio_num); //LOW
```

Abbildung 36: Ansteuerung der GPIO-Schnittstelle Teil 2. [32]

Ein angeschlossener Logikanalysator zeichnete die Veränderungen der GPIO-Pins auf. Dabei wurde definiert, dass solange der GPIO-Pin einen niedrigen Pegel hat, kein Ereignis eingetreten ist. Sollte der Pegel kurzzeitig einen hohen Status annehmen, wurde eine neue Aufgabe begonnen.

Neben der Messung der Zeit mit digitalen Ausgängen wurde auch die interne Zeitmessung für Protokolle der Transport- und Anwendungsschicht verwendet. Die interne Zeitmessung wurde mit Hilfe der Funktion esp_timer_get_time() realisiert. Diese gab die Zeit seit der Initialisierung des Mikrocontrollers aus. Der zurückgegebene Wert wurde in μs angegeben und war vom Typ INT64. Der Timer wurde nach jedem Tiefschlaf zurückgesetzt.

Die Differenz aus Start- und Endzeit wurde in der Funktion sendtime() berechnet. Im Anschluss wurde sie mit dem Transportprotokoll TCP an einen Python-Server versendet. Dieser nahm den Wert in Empfang und schrieb diesen in eine Comma-separated Values (CSV)-Datei. Zusätzlich zur Differenz wurde auch noch ein Name mit dem zu untersuchenden Protokoll und der Konfiguration des Mikrocontrollers übermittelt, um Verwechselungen auszuschließen.

6.4 Strommessung

Der Stromverbrauch des Mikrocontrollers wurde mit dem bereits genannten uCurrent und einem Logikanalysator durchgeführt. Der Testaufbau sah dabei die Trennung der seriellen Schnittstelle zum Laptop vor. Beim Test wurde die GND-Verbindung des Mikrocontrollers mit dem uCurrent verbunden. Dieser leitete die Verbindung über einen Shunt an die GND-Schnittstelle des Akkus weiter. Der Logikanalysator wurde mit den beiden Messpunkten des uCurrents verbunden. Anschließend wurde ein Universal Serial Bus (USB) 3.0 Kabel verwendet, um den Logikanalysator mit dem Computer zu verbinden, auf dem das Aufzeichnungswerkzeug des Analysators lief. Zu beachten war der Arbeitsspeicherbedarf bei Messungen mit hoher Abtastrate. Bei langen Messungen von mehr als 5 s ist es empfehlenswert einen Arbeitsspeicher von mehr als 8 GB zu verwenden. Der Laptop verfügt nicht über diese Kapazität. Deshalb wurde ein Computer mit 24 GB-Arbeitsspeicher für diesen Schritt verwendet. Eine schnelle CPU hilft zudem bei der späteren Analyse und dem Export der Aufzeichnung.

Eine Messung startete 10 ms vor der ersten Zeitmarke. Die Aufzeichnung des Logikanalysators musste manuell gestartet werden und konnte nicht in Echtzeit in der Computeranwendung angesehen werden. Deshalb bedurfte dieser Schritt ein paar Wiederholungen.

Die Computeranwendung des Analysators benötigte zunächst eine Angabe dazu, mit welcher Abtastrate und für welches Zeitintervall eine Messung angefertigt werden soll. Die Werte mit den gewünschten Angaben wurden nach dem manuellen Starten auf den Computer übertragen. Dieser Vorgang benötigte ein paar Sekunden. Sobald die Messung fertig war, konnten die Werte als CSV-Datei exportiert werden und mit Hilfe der Python3-Bibliothek Panda geplottet werden. Dabei ergaben sich Schwierigkeiten mit dem Arbeitsspeicher. Eine Zeitmessung von 2 s mit einer Abtastrate von 50 MSa/s besaß 250 Millionen Einträge, die eine CSV-Datei mit ca. 4 GB belegten. Das Importieren dieser Datei, vor allem für mehrere verschiedene Messungen, verbrauchte viel vom vorhandenen Arbeitsspeicher. Deshalb sollte bei diesem Schritt immer ein Blick auf die freien Ressourcen des Systems geworfen werden, um Abstürze zu vermeiden.

6.5 Messung der Bootphase

Der ESP32 durchläuft während des Bootens zwei Phasen. Die Analyse der ersten Phase ergab nur einen Faktor, der einen Einfluss auf diese nehmen kann. Das Verbinden des GPIO 15 mit GND deaktivierte die Ausgabe der ersten Bootphase. In der zweiten Bootphase konnte die Logging-Stufe in der Software deaktiviert werden.

Als definierter Zeitbereich wurden der Start des Mikrocontrollers und die erste Funktion der Applikation genutzt. Diese beiden Punkte wurden ab nun als Start- und Endtrigger bezeichnet. Zur Messung der Zeit eignete sich in diesem Fall die Zeitmessung mit dem Logikanalysator. Dieser maß den GPIO EN als Starttrigger. Der GPIO EN besaß vor dem Start einen hohen Pegel. Mit der Änderung dieses Pegels wurde der Start signalisiert. Anschließend setzte der Mikrocontroller den GPIO 21 auf einen hohen Pegel, sobald die beiden Bootphasen beendet wurden. Die verstrichene Zeit wurde mit dem Logikanalysator aufgezeichnet und später analysiert.

6.6 Messung der Netzwerkinitialisierung

Die Netzwerkinitialisierung kann mit der internen Zeitfunktion ermittelt werden. Hierfür wurde vor dem Starten des WLANs eine Zeitmarke gesetzt. Anschließend wurde das WLAN konfiguriert und gestartet. Sobald eine Verbindung zum Netzwerk bestand, setzte das ESP-IDF ein Connected-Bit. Zu diesem Zeitpunkt könnte ein Protokoll zur Nachrichtenübertragung mit der Arbeit beginnen. Nach dem Setzen dieses Bits wurde eine zweite Zeitmarke gesetzt. Die Differenz beider Zeitmarken ergab anschließend die verbrauchte Zeit. Zur Übertragung der Zeit wurde die bereits vorgestellte Methode mit einem Python-Server genutzt. Die daraufhin gesammelten Daten wurden anschließend analysiert.

6.7 Messung der Protokolle

Für die Nachrichtenübertragung wurden MQTT, MQTT-SN und CoAP ausgewählt. Die Analyse zeigte, dass nicht alle Protokolle eine eigene Bibliothek im ESP-IDF besitzen. Für MQTT sind alle benötigten Features bereits umgesetzt. CoAP hat in der jetzigen Form keine Unterstützung für DTLS und MQTT-SN hat keine bekannte Bibliothek für das ESP-IDF. Im folgenden Unterkapitel wird die Implementierung der Protokolle behandelt mitsamt den noch nicht vorhandenen Bibliotheken oder Eigenschaften, die noch hinzugefügt werden müssen. Für die Messungen wurde im Anschluss die interne Zeitmessung genutzt.

6.7.1 MQTT

Die MQTT-Bibliothek vom ESP-IDF verweist auf ein eigenes Github Repository. Dieses wurde von der Community geschrieben und nachträglich als Komponente eingegliedert. Alle MQTT-Funktionen sind enthalten und können sofort genutzt werden. Die Implementierung ist damit für den ESP32 ohne großen Zeitaufwand umsetzbar.

Der MQTT-Broker musste auf dem Computer installiert und gestartet werden. Dazu wurde die Software Mosquitto von Eclipse ausgewählt.

Die Implementierung von MQTT auf dem ESP32 startet, nachdem das WLAN initialisiert wurde. Durch ein Connected-Bit wird der MQTT-Bibliothek signalisiert, dass sie die Nachrichtenübertragung starten kann.

Das Senden einer Nachricht wird durch das Erstellen einer MQTT-Konfiguration initialisiert. Übergeben werden die Adresse des MQTT-Brokers sowie der Event Handler für MQTT-Ereignisse.

Anschließend wird der MQTT-Client gestartet. Über den Event Handler werden alle MQTT-Ereignisse gesteuert. Die Abbildung 37 listet alle Ereignisse auf, die im Switch-Case behandelt werden.

```
1 case MQTT_EVENT_CONNECTED:
2 case MQTT_EVENT_DISCONNECTED:
3 case MQTT_EVENT_SUBSCRIBED:
4 case MQTT_EVENT_UNSUBSCRIBED:
5 case MQTT_EVENT_PUBLISHED:
6 case MQTT_EVENT_DATA:
7 case MQTT_EVENT_ERROR:
```

Abbildung 37: MQTT-Ereignisse. [30]

Sobald der Client das Event CONNECTED auslöst, können verschiedene MQTT-Methoden ausgeführt werden. Zu ihnen zählen:

```
esp_err_t esp_mqtt_client_subscribe(...);
esp_err_t esp_mqtt_client_unsubscribe(...);
int esp_mqtt_client_publish(...);
esp_err_t esp_mqtt_client_destroy(...);
```

Abbildung 38: MQTT-Methoden. [30]

Für die Implementierung des Versuchs muss die Funktion esp_mqtt_client_publish() im MQTT_EVENT_CONNECTED-Ereignis aufgerufen werden. Übergeben werden der

MQTT-Client, ein Topic `/test` und der Payload. Dieser besteht aus der entsprechenden Anzahl an Bytes. Der Inhalt des Char-Arrays wird mit „A“ aufgefüllt. Zusätzlich müssen noch die QoS-Stufe und das Retain-Flag angegeben werden. Das Retain-Flag hat den Wert 0. Die QoS-Stufe wird dem entsprechenden Versuchsdurchlauf angepasst. Sobald die Funktion abgeschlossen ist, wird eine Zeitmarke für das Ende der Übertragung gesetzt. Nachdem die Start- und Endzeit gemessen wurden, übergibt die MQTT-Bibliothek an die implementierte Methode zur Übertragung der Zeitdifferenz. Diese baut eine Verbindung zum Computer auf und überträgt den Wert. Im Anschluss übergibt sie, nach einer kleinen Verzögerung von 50 ms, an die Tiefschlaf-Funktion. Der Tiefschlaf beträgt 10 μs. Anschließend wird das Prozedere bis zur gewünschten Anzahl an Einzelmessungen wiederholt.

6.7.2 MQTT-TLS

Die MQTT-Bibliothek unterstützt den Aufbau und die Verwaltung einer TLS-gesicherten Verbindung. Hierfür werden die vom ESP-IDF bereitgestellten Funktionen verwendet.

Die Umsetzung von MQTT-TLS beginnt mit der Erstellung selbst signierter Zertifikate mit RSA-Verschlüsselung. Verwendet wurde dafür das Programm OpenSSL. Es erstellt eine Central Authority (CA) und das notwendige Schlüsselmaterial für den Server. Optional besteht die Möglichkeit eine Client-Authentifizierung durchzuführen. Dann müsste zusätzlich noch ein Schlüsselpaar für den Client erzeugt werden.

```
#!/bin/bash
HOST="HLK-T440"
cd "/etc/mosquitto/certs"

###RSA
openssl genrsa -out "ca-key.pem" 2048
openssl req -new -x509 -days 365 -key "ca-key.pem" -sha256 -out
 ↪  "ca.pem"
openssl genrsa -out "server-key.pem" 2048

openssl req -subj "/CN=$HOST" -sha256 -new -key "server-key.pem"
 ↪  -out "server.csr"
echo subjectAltName = DNS:$HOST,IP:10.42.0.1,IP:127.0.0.1 >>
 ↪  "extfile.cnf"
echo extendedKeyUsage = serverAuth >> "extfile.cnf"

openssl x509 -req -days 365 -sha256 -in "server.csr" -CA
 ↪  "ca.pem" -CAkey "ca-key.pem" -CAcreateserial -out
 ↪  "server-cert.pem" -extfile "extfile.cnf"
```

Abbildung 39: Generierung von selbstsigniertem Zertifikat.

Die erstellten Dateien aus der Abbildung 39 sind im Ordner `/etc/mosquitto/certs` gespeichert. Für den Mosquitto-Broker wurde eine Konfiguration erstellt. Diese beinhaltet den Pfad der jeweiligen Schlüssel und Zertifikate.

```
listener 1883

listener 8883
cafile /etc/mosquitto/certs/ca.pem
certfile /etc/mosquitto/certs/server-cert.pem
keyfile /etc/mosquitto/certs/server-key.pem
require_certificate false
```

Abbildung 40: Mosquitto-Konfiguration.

Die Konfiguration aus Abbildung 40 teilt dem Mosquitto Broker mit, dass er auf zwei Ports MQTT-Nachrichten annehmen soll. Dabei besteht auf Port 8883 die Besonderheit, dass eine TLS geschützte Verbindung mit den vorher generierten Zertifikaten genutzt wird.

Auf der Seite des Mikrocontrollers muss die `ca.pem` in die Applikation mit eingebunden werden. Dies geschieht durch die Dateien `CMakeList.txt` und `main/component.mk`. Anschließend kann im Programm das Zertifikat aufgerufen und der MQTT-

Konfiguration übergeben werden. Die Adresse wird auf den Port 8883 geändert. Zusätzlich ist es noch wichtig die Überprüfung des Zertifikats auf optional zu stellen, da sie sonst beim selbstsignierten Zertifikat fehlschlägt. Dazu wird in der `esp_tls.c` der Wert `MBEDTLS_SSL_VERIFY_REQUIRED` auf
`MBEDTLS_SSL_VERIFY_OPTIONAL` umgeändert.

Weitere Schritte sind nicht notwendig. Zum Schluss werden dieselben Zeitmarken und Übertragungswege für die Zeitdifferenz mit eingebunden und die Versuche nach Plan durchgeführt.

6.7.3 MQTT-SN

Das ESP-IDF besitzt keine Bibliothek für MQTT-SN. Deshalb wurde im Verlauf der Arbeit eine Lösung gesucht, diese Lücke zu schließen. Die Eclipse Foundation, von der bereits der Mosquitto-Broker verwendet wird, stellt auch eine Lösung für MQTT-SN bereit. Zu dieser Lösung gehören ein MQTT-SN-Gateway und ein MQTT-SN-Client. Die Portierung des Clients war aufgrund der Nutzung von Standardfunktionen einfach. Durch die Implementierung von LwIP in das ESP-IDF waren alle Übertragungsfunktionen vorhanden. Somit musste die Bibliothek nur eingebunden werden.

```
1 options.clientID.cstring = "ESP";
2 topic.type = MQTTSN_TOPIC_TYPE_SHORT;
3 memcpy(topic.data.short_name, "te", 2);
4 len = MQTTSNSerialize_publish(buf, buflen, dup, qos, retained,
   ↪  packetid, topic, payload, payloadlen);
5 rc = lowlevel_sendPacketBuffer(host, port, buf, len);
```

Abbildung 41: MQTT-SN-Client-Konfiguration. [1]

Die Konfiguration des Clients läuft annähernd gleich zum MQTT-Client ab. Neben den Daten des MQTT-SN-Gateways und der QoS-Konfiguration wurde noch ein kurzer Name für das Topic ausgewählt. In diesem Fall wurde der Wert mit `te` belegt. Payloads von 1000 Byte werden jedoch vom Quellcode nicht unterstützt. Sie wurden deshalb vom Test ausgeschlossen.

Nachdem die MQTT-SN-Bibliothek auf dem ESP32 lief, konnte das Gateway installiert werden. Hierzu wurde es auf dem Computer kompiliert und konfiguriert. Der wichtigste einzutragende Parameter ist der MQTT-Broker, der die Nachrichten erhalten soll. Des Weiteren war die Angabe, ob es sich um ein Gateway mit QoS -1 Unterstützung handelt, wichtig für den Betrieb dieser QoS-Stufe. War dies der Fall, musste zusätzlich die

Client-ID und die IP-Adresse des Clients in eine separate Datei namens `clients.conf` eingetragen werden.

Der anschließende Test zeigte, dass die Portierung funktioniert und die Daten übertragen wurden. Es entstanden jedoch in regelmäßigen Abständen lange Wartezeiten. Durch die Überprüfung des Netzwerkverkehrs konnte ein Fehler seitens des ESP32 ausgeschlossen werden. Deshalb musste ein Problem in der Software des Gateways vorliegen, welche diese Wartezeit hätte erklären können. Eine Überprüfung des Quellcodes des Gateways nach Begriffen wie `delay` oder `sleep` ergab einen Treffer in der Datei `MQTTSNGWBrokerRecvTask.cpp`.

```
1 if (maxSock == 0)
2 {
3         usleep(500 * 1000);
4 }
```

Abbildung 42: MQTT-SN-Gateway-Verzögerung. [1]

Nachdem der Wert auf 0 gesetzt wurde, traten die langen Wartezeiten nicht mehr auf. Weitere Nachforschungen ergaben, dass die Schlafphase für den Raspberry Pi eingebaut wurde, um die CPU-Last zu verringern [18]. Diese Änderung verursachte ein Problem, sobald der Client die Verbindung trennte und eine neue aufbauen wollte.

Abschließend wurden die Zeitmarken und die Übertragung der Zeitdifferenz implementiert und die Versuche anhand des Plans durchgeführt.

6.7.4 CoAP

Die Analyse von CoAP zeigte, dass bereits eine fertige Bibliothek im ESP-IDF vorhanden ist. Die Bibliothek basiert auf einer etwas älteren libcoap-Version vom 16.12.2016. Bis auf die Unterstützung von DTLS bietet sie alle RFC-spezifizierten CoAP-Funktionen. Deshalb konnte sofort mit der Umsetzung gestartet werden.

Wie bereits in den vorherigen Protokollen übergibt der ESP32, nach dem erfolgreichen Verbindungsaufbau, die Kontrolle an das Nachrichtenprotokoll. Sobald dies geschieht, wurde die Messung der Zeit gestartet.

Die Konfiguration von CoAP hat sich etwas schwieriger gestaltet als bei den beiden vorherigen Protokollen. Es beginnt mit dem Setzen der Zieladresse. Dabei wurde der Pfad mit angehängt. Zusätzlich muss die Methode angegeben werden. Für die Versuche wurde die Methode PUT verwendet. Anschließend konnte eine Anfrage erstellt und der Typ der Nachricht festgelegt werden. Der Typ definiert, ob es sich um eine CON oder

NON handelt. Der nächste Schritt war das Hinzufügen des Payloads. Handelte es sich um eine CON, wurde zusätzlich ein Response Handler registriert.

Die Bibliothek libcoap nutzt zwei verschiedene Funktionen für das Senden von Nachrichten. Obwohl vorher definiert wurde, um was für einen Typ es sich handelt, gab es die Funktionen `coap_send` und `coap_send_confirmed`. Die Implementierung enthält die jeweils passende Funktion. Nach dem Senden der Nachricht kann die zweite Zeitmarke gesetzt werden. Dies ist entweder direkt nach dem Senden oder beim Empfangen der Bestätigung der Fall. Anschließend wurde die Zeitdifferenz an den Computer übermittelt.

6.7.5 CoAP-DTLS

Die Umsetzung von CoAP mit DTLS-Unterstützung ist ein größeres Unterfangen, aufgrund der tiefgreifenden Eingriffe in die Bibliothek. Zudem existiert noch keine vollständige Unterstützung des ESP-IDFs. Da die bereits für TLS verwendete Bibliothek mbedTLS auch DTLS unterstützt, wurde der Versuch gewagt, eine Implementierung vorzunehmen.

Das ESP-IDF verfügt über keine Dokumentation zum Thema DTLS. Die Bibliothek mbedTLS bietet hingegen eine umfangreiche Dokumentation mit Beispielquellcode. Deshalb wurde zunächst versucht, eine Verbindung von einem ESP32 zu einem OpenSSL-Server aufzubauen. Dafür musste im Menü von mbedTLS die DTLS-Unterstützung aktiviert werden. Die Kompilierung des Beispielcodes funktionierte aber aufgrund nicht vorhandener Teile von mbedTLS nicht. Es fehlten die in der Abbildung 43 dargestellten Methoden.

```
1 struct _hr_time
2 unsigned long mbedtls_timing_get_timer( struct
   ↪ mbedtls_timing_hr_time *val, int reset )
3 void mbedtls_timing_set_delay( void *data, uint32_t int_ms,
   ↪ uint32_t fin_ms )
4 int mbedtls_timing_get_delay( void *data )
```

Abbildung 43: DTLS-Funktionen. [21]

Nachdem diese aus dem Quellcode von mbedTLS Repository in den Quellcode des Beispiels kopiert wurden, konnte eine Verbindung mit dem OpenSSL-Server aufgebaut werden.

Nun musste der Quellcode von libcoap analysiert werden. Ziel war es, entsprechende Befehle zu finden, die Pakete schicken oder empfangen.

Die Datei `coap_io_socket.c` ist in der Portierung des ESP-IDF enthalten und besaß die entsprechenden Funktionen. Die Analyse der Pakete zeigte Schwierigkeiten beim Ersetzen bestimmter Strukturen auf. Zwar konnte der Befehl `sendto()` einfach durch ein `mbedtls_ssl_write()` ersetzt werden. Jedoch fehlten der mbedTLS-Funktion die notwendigen DTLS-Parameter und der Ablauf von libcoap wurde gestört.

```
1 extern mbedtls_ssl_context ssl;
2 extern mbedtls_net_context server_fd;
```

Abbildung 44: mbedTLS-Variablen.

Das erste Problem wurde durch das Erstellen von externen Variablen gelöst. Die in der Abbildung 44 gezeigten Variablen können sowohl von der Bibliothek als auch von der Anwendung verändert werden.

Die Bibliothek libcoap definiert für jede Verbindung einen Endpunkt. Ohne diesen Endpunkt und die Übergabe bestimmter Parameter, werden entsprechende Funktionen im Ablauf nicht ausgeführt. Deshalb musste ein Endpunkt erstellt werden, der gewisse Parameter bei der Übergabe an die nächste Funktion bereitstellt.

```
1  coap_endpoint_t *
2  coap_new_endpoint(const coap_address_t *addr, int flags) {
3    //emulate EP
4    struct coap_endpoint_t *ep;
5    ep = coap_malloc_posix_endpoint();
6    ep->handle.fd = 55;
7    ep->flags = flags;
8    ep->addr.size = addr->size;
9    return (coap_endpoint_t *)ep;
10 }
```

Abbildung 45: Endpunkt einer libcoap-Bibliothek.

Hierzu wurde, wie in der Abbildung 45, ein Endpunkt emuliert. Die notwendigen Parameter für spätere Funktionen wurden gesetzt und getestet.

Nachdem ein Endpunkt erstellt wurde, kann mit ihm gesendet werden. Es muss aber eine Verbindung bestehen. Deshalb musste zuvor mit mbedTLS eine DTLS-Verbindung aufgebaut werden. Dies musste in der Applikation stattfinden. Der Aufbau einer DTLS-Verbindung direkt in der libcoap-Bibliothek konnte nicht durchgeführt werden.

Sobald die Verbindung und der Endpunkt stehen, können Pakete mit libcoap über DTLS gesendet und empfangen werden. Hierfür war im Versuch nur der Austausch der sendto- und recvfrom-Funktionen mit deren mbedTLS-Äquivalenten notwendig.

Zum Abschluss der Verbindung wird der Endpunkt gelöscht. Zusätzlich muss die DTLS-Verbindung beendet werden. Aus diesem Grund wurde die Funktion bearbeitet und eine CloseNotify-Nachricht verschickt.

```
1  void coap_free_endpoint(coap_endpoint_t *ep) {
2    if(ep) {
3      if (ep->handle.fd >= 0){
4        //close(ep->handle.fd);
5        mbedtls_ssl_close_notify(&ssl);
6        mbedtls_ssl_session_reset(&ssl);
7        mbedtls_net_free(&server_fd);
8      }
9      coap_free_posix_endpoint((struct coap_endpoint_t *)ep);
10   }
11 }
```

Abbildung 46: Schließung einer mbedTLS-DTLS-Verbindung.

Am Ende der Implementierung wurden alle notwendigen Funktionen getestet. Dabei fiel auf, dass eine Verbindung zum OpenSSL-Server möglich war. Die Verbindung zu einem libcoap-Server wurde aber beim Handshake abgebrochen. Nachforschungen ergaben, dass der Server standardmäßig eine Client-Authentifizierung erwartet. Da nur der Server authentifiziert werden sollte, konnte eine Änderung der Parameter das Problem lösen.

Die anschließenden Tests nach Versuchsplan verliefen ohne weitere Komplikationen.

7 Ergebnisse und Auswertung

In diesem Kapitel werden die gemessenen Phasen des ESP32 präsentiert und analysiert. Die dargestellten Werte in den Abbildungen haben aufgrund der genutzten Grafik-Bibliothek einen Punkt als Dezimaltrennzeichen.

7.1 Bootzeit

Die erste Phase, die der ESP32 durchläuft, ist die Bootphase. Sie ist in zwei Teile eingeteilt. Diese Teile wurden mit Hilfe der GPIO-Schnittstelle und dem Logikanalysator gemessen. Das Ergebnis der Messung ist in der Abbildung 47 dargestellt.

Die Standardbootzeit des ESP32 beträgt 234,5 ms. Durch die einfache Maßnahme die Ausgabe von Log-Nachrichten in der zweiten Bootphase zu unterbinden, kann diese Zeit auf 91,8 ms reduziert werden. Das ist eine Verminderung der Zeit um 60 %. Sobald die Ausgabe der ersten Bootphase deaktiviert wird, verkürzt sich die Zeit um weitere 23 ms. Somit wird eine Bootzeit von 69,8 ms erreicht, ohne große Eingriffe vorzunehmen. Die Analyse der zweiten Bootphase ergab, dass die Frequenz initial auf 80 MHz gesetzt wird. Erst später wird die gewünschte Taktrate genutzt. Durch die Erhöhung der Taktrate in der zweiten Bootphase, kann die Zeit zum Starten auf 51,2 ms bzw. 45,2 ms reduziert werden. Die Bootzeit kann somit um 80 % beschleunigt werden.

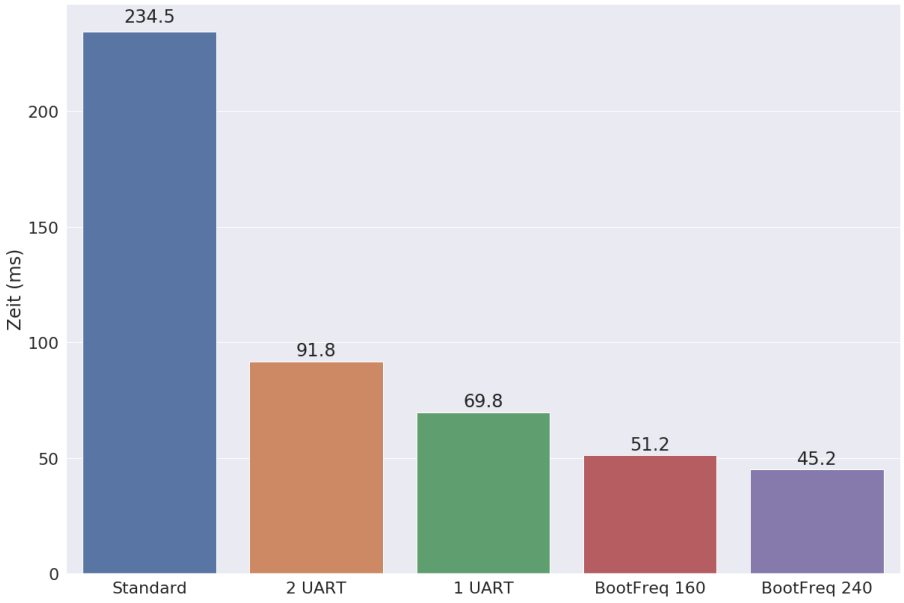


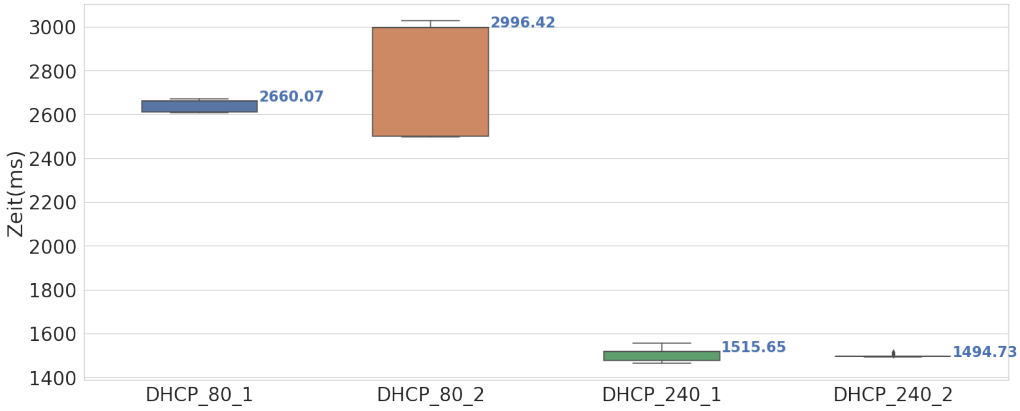
Abbildung 47: Versuchsergebnisse Bootzeit.

7.2 WLAN

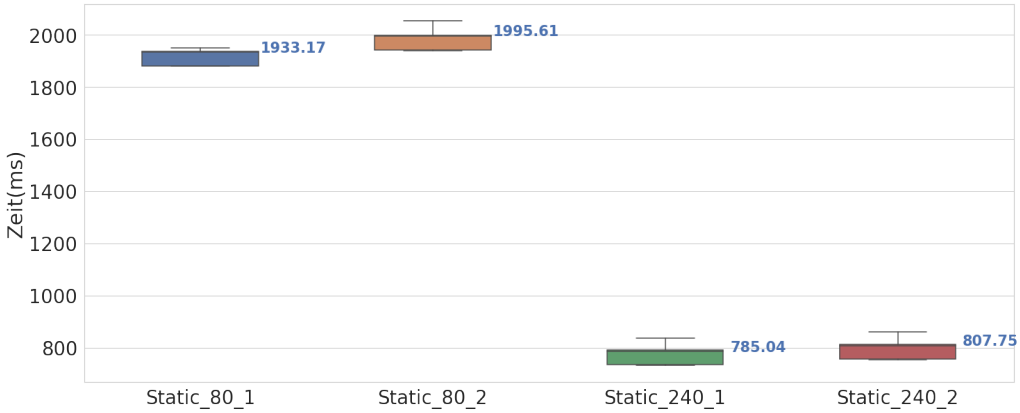
Die Zeit, die bei einer Assoziierung mit einem WLAN-AP benötigt wird, sollte so kurz wie möglich sein. Da dies eine unumgängliche Funktion des Systems ist, sollte ein besonderes Augenmerk auf deren Optimierung gelegt werden.

Die gemessenen Zeiten starten bei der Initialisierung des WLANs. Die Messung endet nach dem Setzen des Connected-Bit. Dabei wurden die Varianten Ein-Kern oder Zwei-Kerne mit jeweils 80 MHz oder 240 MHz 1000-mal getestet. Die Konfiguration steht unter dem jeweiligen Boxplot. Der Median ist in blauer Schrift rechts neben dem Boxplot dargestellt.

7.2.1 Assoziierung



(a) DHCP.



(b) Statische IP-Adresse.

Abbildung 48: Versuchsergebnisse Assoziierung mit einem gesicherten WLAN-AP.

Bei der Assoziierung mit dem WLAN-AP besteht die Wahl, eine IP-Adresse über einen DHCP-Server zu beziehen. In der Abbildung 48 ist die benötigte Zeit unter verschiede-

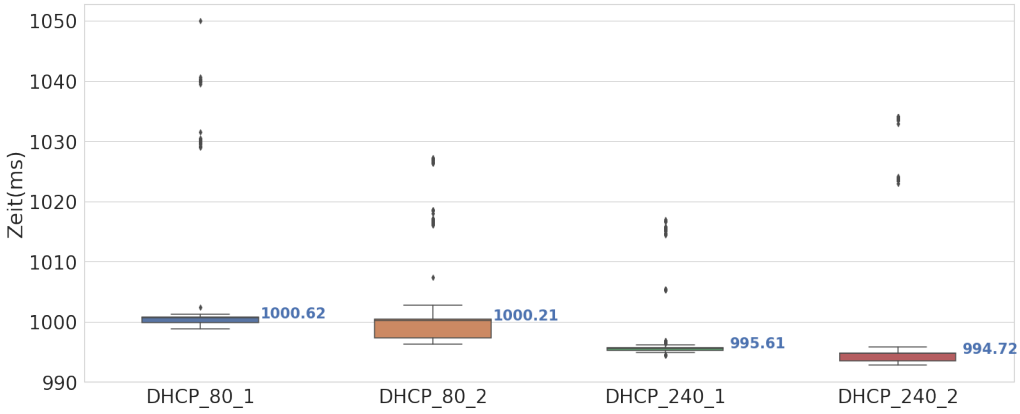
nen Prozessoreinstellungen ersichtlich. Zu sehen ist, dass der Median bei 80 MHz und einem Kern bei 2660,07 ms liegt. Werden zwei Kerne verwendet, steigt die Zeit auf 2996,42 ms. Der Verlust beträgt 11%.

Wird die Taktrate auf 240 MHz erhöht, liegt der Median bei einem Kern bei 1515,65 ms. Damit werden 43% der Zeit eingespart. Bei zwei Kernen liegt der Median bei 1494,73 ms, eine Steigerung von 50% zu 80 MHz und einem Kern.

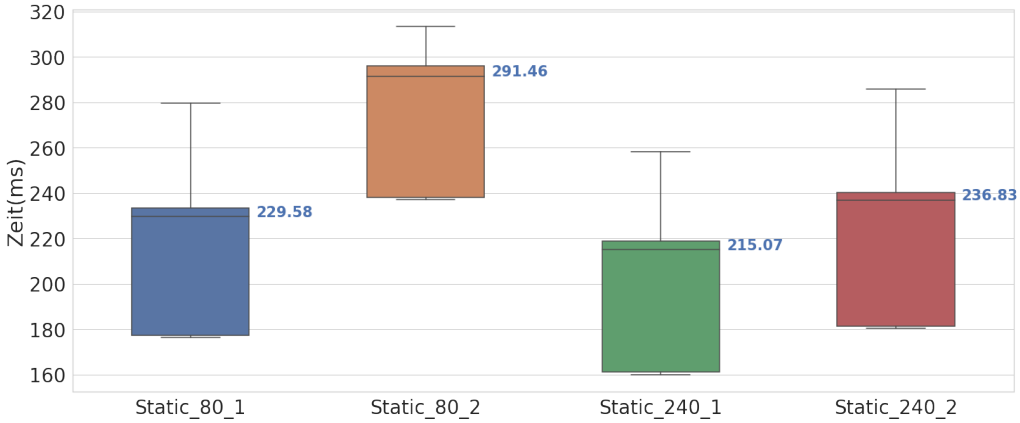
Die alternative Methode ist das Setzen einer statischen IP-Adresse mitsamt der IP-Adresse des Gateways. Dabei werden bei 80 MHz und einem Kern 1933,17 ms benötigt. Bei zwei Kernen verschlechtert sich der Wert um 3,23% auf 1995,61 ms.

Taktet man die Kerne auf 240 MHz, verbessert sich auch wieder die benötigte Zeit. Mit einem Kern werden 784,04 ms benötigt. Schaltet man den zweiten Kern hinzu, sind es 807,75 ms.

Insgesamt zeigt die Versuchsreihe, dass das DHCP-Protokoll deutlich langsamer ist. Eine statische IP-Adresse spart im Vergleich viel Zeit ein. Inwieweit hier noch Optimierungspotenzial vorhanden ist, wurde in einem Versuch ohne Verschlüsselung getestet.



(a) DHCP.



(b) Statische IP-Adresse.

Abbildung 49: Versuchsergebnisse Assoziierung mit einem ungesicherten WLAN-AP.

Der Verbindungsaufbau mit einem offenen WLAN-AP reduziert die benötigte Zeit. Die Mediane des DHCP-Versuchs liegen nah beieinander. Die Werte befinden sich zwischen 1000,62 ms und 994,72 ms. Die Steigerung durch den Einsatz von zwei Kernen beträgt bei beiden Prozessortakten weniger als 1 ms.

Wird eine statische IP-Adresse samt Gateway-Adresse gesetzt, liegen die Werte zwischen 291,46 ms und 215,07 ms. Die Abweichung durch unterschiedliche Prozessoreinstellungen ist hier wieder größer.

Die Versuche mit einer statischen IP-Adresse weisen einen interessanten Sachverhalt auf. Der Abstand zwischen dem gesicherten WLAN-AP mit 240 MHz und einem Kern zu einem gesicherten WLAN-AP mit 240 MHz und zwei Kernen beträgt 22,71 ms. Bei einer unverschlüsselten Verbindung sind es im selben Fall 21,76 ms. Ähnliches lässt sich bei 80 MHz beobachten. Daraus kann geschlossen werden, dass bei einer statischen IP-Adresse der zweite Kern eine konstante Verzögerung bewirkt.

Es stellt sich deshalb die Frage, warum ein negativer Effekt bei der Nutzung des zweiten Kerns auftritt. Analysiert man den Quellcode wird klar, warum ein positiver Effekt kaum zustande kommen kann. Nach der zweiten Bootphase wird der Quellcode aus cpu_start.c ausgeführt. Der Einsprungpunkt ist die Funktion `call_start_cpu0`. Diese verwendet zunächst nur die PRO_CPU. Später wird auch die APP_CPU gestartet.

```
portBASE_TYPE res = xTaskCreatePinnedToCore(&main_task, "main",
  ↪ ESP_TASK_MAIN_STACK, NULL, ESP_TASK_MAIN_PRIO, NULL, 0);
```

Abbildung 50: Prozessorzugehörigkeit. [32]

Der Startbefehl des Hauptprozesses ist in der Abbildung 50 dargestellt. Dieser ordnet die Pro_CPU dem Hauptprozess zu, der wiederum den späteren Einsprungpunkt der Applikation `app_main()` aufruft. Die Funktion `app_main()` initialisiert das WLAN. Somit hat auch jegliche WLAN-Aktivität die Prozessorzugehörigkeit zur PRO_CPU. Die APP_CPU wird also nicht belastet. Trotzdem kommt es zu Verzögerungen bei der WLAN-Initialisierung. Das könnte an der APP_CPU liegen, die den Prozess-Scheduler immer wieder fragt, ob ein neuer Prozess zur Verfügung steht. Da der Prozess-Scheduler auf der PRO_CPU ausgeführt wird, besteht eine zusätzliche Belastung, die den Gesamtprozess verlangsamen könnte.

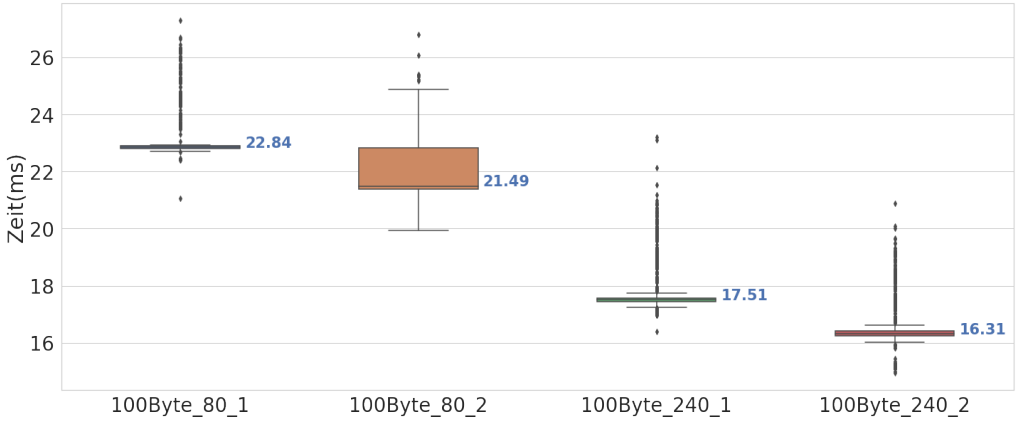
Bei der Nutzung eines DHCP-Clients verschwinden die Auswirkungen des zweiten Kerns durch die Verzögerung des Protokolls. Zudem läuft der DHCP-Client in einem eigenen Task unter dem TCP/IP-Adapter. Dieser ist nicht der PRO_CPU zugeordnet. Als Standardwert hat er keine Zugehörigkeit. Hierdurch kann bei fast allen Messungen festgestellt werden. dass mit DHCP und einem zweiten Kern ein leicht positiver Effekt erzielt werden kann, auch wenn bei einer verschlüsselten Verbindung mit 80 MHz ein Ausreißer zu finden ist.

Der zweite Versuch zeigt, dass der Großteil der Zeit für den Handshake der WLAN-Verschlüsselung gebraucht wird. Der zweite Kern bewirkt kaum einen positiven Effekt auf die beiden Methoden, weshalb hier die Wahl eines Kerns zu bevorzugen ist. Die Erhöhung des Prozessortakts zeigt hingegen das höchste Optimierungspotential dieser Phase.

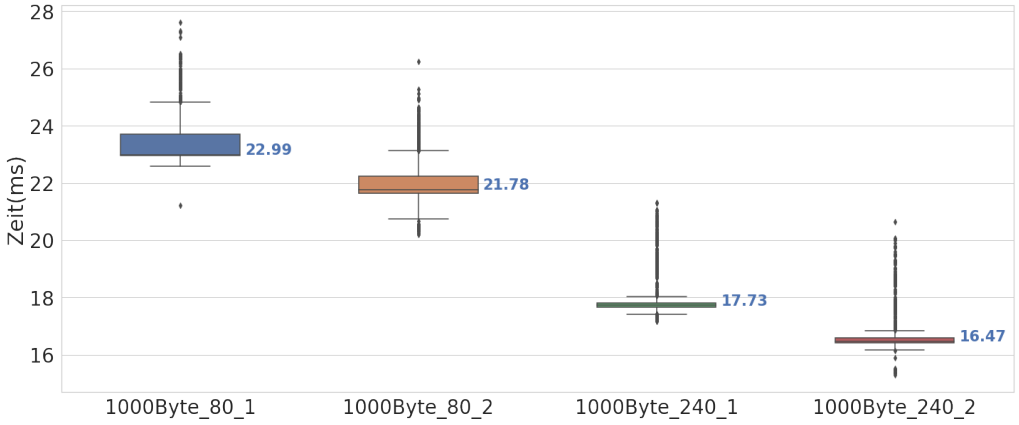
7.3 MQTT-Protokoll

In diesem Unterkapitel werden die Messergebnisse der dritten Phase des ESP32 präsentiert. Es wurden Nachrichten unterschiedlicher Größen an einen Server versandt. Hierzu wurden die Protokolle MQTT, MQTT-SN und CoAP getestet. Jeder Versuch wurde in den geplanten Einstellungen 1000-mal wiederholt. Angefangen wird mit dem MQTT-Protokoll.

7.3.1 MQTT QoS 0



(a) 100 Byte.



(b) 1000 Byte.

Abbildung 51: Versuchsergebnisse MQTT QoS 0.

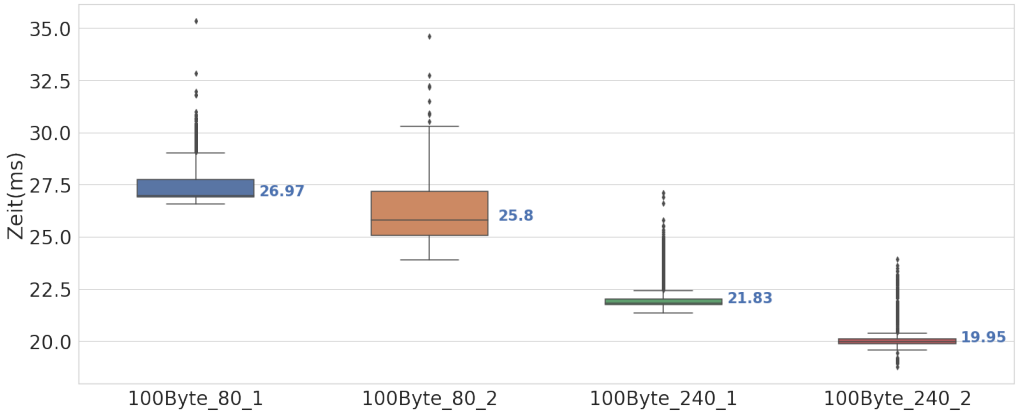
Die gemessene Zeit der Übertragung einer MQTT-Nachricht mit QoS 0 und einem Payload von 100 Byte beträgt bei 80 MHz und einem Kern 22,84 ms. Der Einsatz des zweiten Kerns senkt den Median auf 21,49 ms ab.

Wird der Prozessortakt gesteigert, beträgt der Median bei 240 MHz und einem Kern

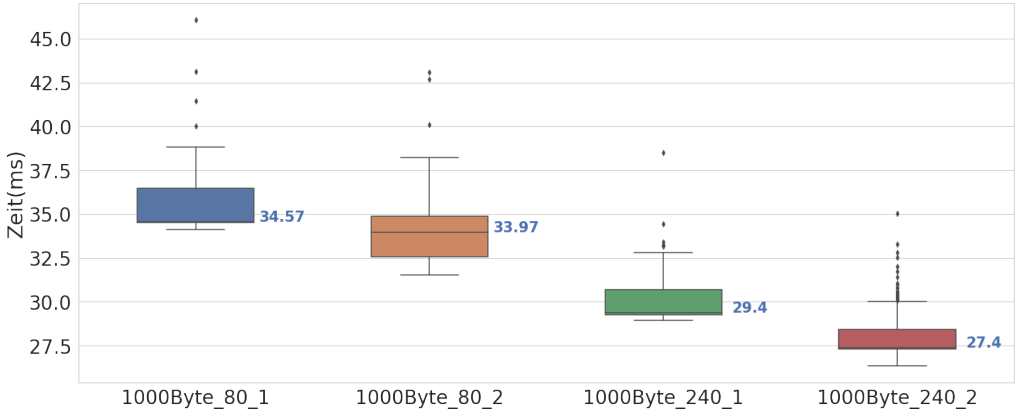
17,51 ms und bei zwei Kernen 16,31 ms. Erhöht man das Payload auf 1000 Byte, hat dies fast keine Auswirkungen auf die Übertragungszeit.

Es zeigt sich, dass bei MQTT QoS 0 der zweite Kern einen Effekt auf die Übertragungszeit hat. Die Verbesserung durch den Prozessortakt bewirkt noch einen größeren Effekt auf die Zeit. Auffällig ist, dass der Unterschied zwischen 100 Byte und 1000 Byte nur wenige Mikrosekunden beträgt.

7.3.2 MQTT QoS 1



(a) 100 Byte.



(b) 1000 Byte.

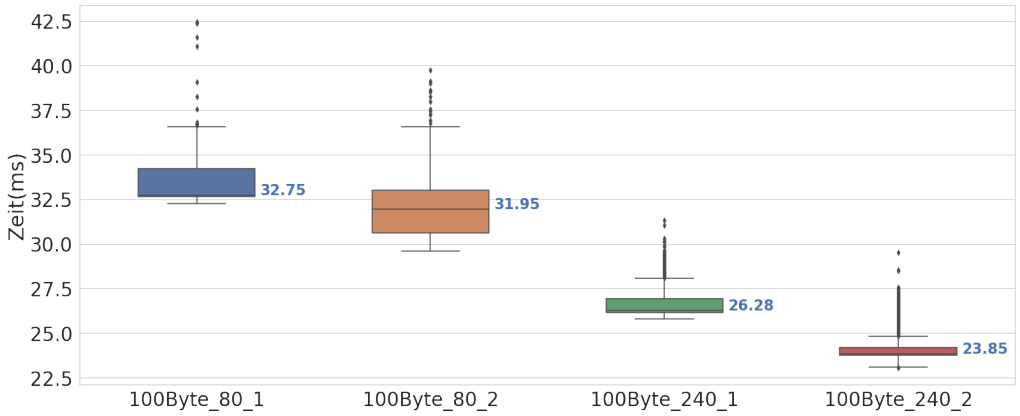
Abbildung 52: Versuchsergebnisse MQTT QoS 1.

Wird eine MQTT-Nachricht mit QoS 1 verschickt, erhöht dies die Übertragungsdauer bei 100 Byte, einem Kern und 80 MHz auf 26,97 ms. Bei zwei Kernen sind es 25,8 ms. Die Nutzung eines höheren Prozessortakts senkt die Zeit auf 21,83 ms bei einem Kern und 19,95 ms bei zwei Kernen.

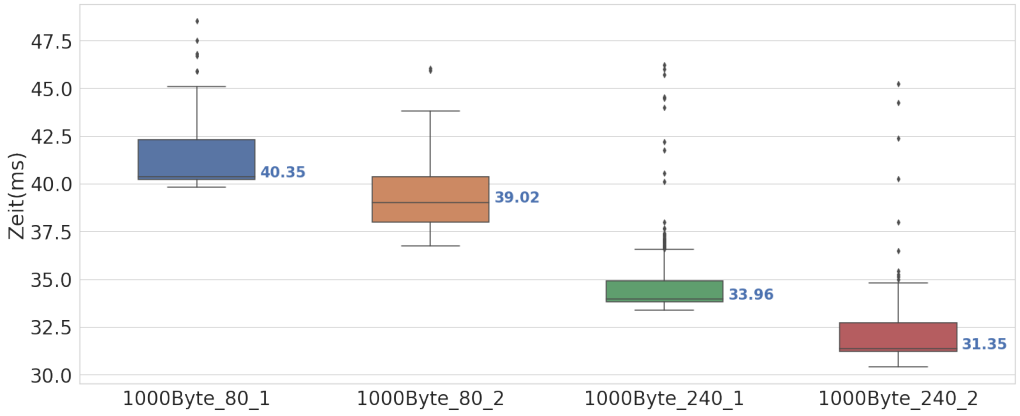
Wird die Nachrichtengröße auf 1000 Byte erhöht, steigt die Zeit auf 34,57 ms bei einem Kern und 80 MHz. Bei zwei Kernen sind es nur 0,6 ms weniger. Mit 240 MHz ergeben sich Zeiten von 29,4 ms bei einem Kern und 27,4 ms bei zwei Kernen.

Der Trend, dass zwei Kerne eine positive Auswirkung auf die Zeit haben, setzt sich hier fort. Die größte Auswirkung hatte aber wieder die Auswahl des Prozessortakts.

7.3.3 MQTT QoS 2



(a) 100 Byte.



(b) 1000 Byte.

Abbildung 53: Versuchsergebnisse MQTT QoS 2.

Die höchste QoS-Stufe für MQTT verbraucht bei 100 Byte und 80 MHz 32,75 ms. Das ist fast so viel, wie für die zehnfache Datenmenge bei QoS 1 benötigt wurde. Bei zwei Kernen sind es 31,95 ms. Die Prozessortakterhöhung vermindert die Zeit auf 26,28 ms bzw. 23,85 ms. Die Nutzung eines Payloads von 1000 Byte erhöht auch hier die Übertragungszeit. Bei 80 MHz und einem Kern sind es 40,35 ms. Schaltet man den zweiten Kern hinzu sind es nur noch 39,02 ms. Die Zeit wird am meisten durch den Prozessortakt be-

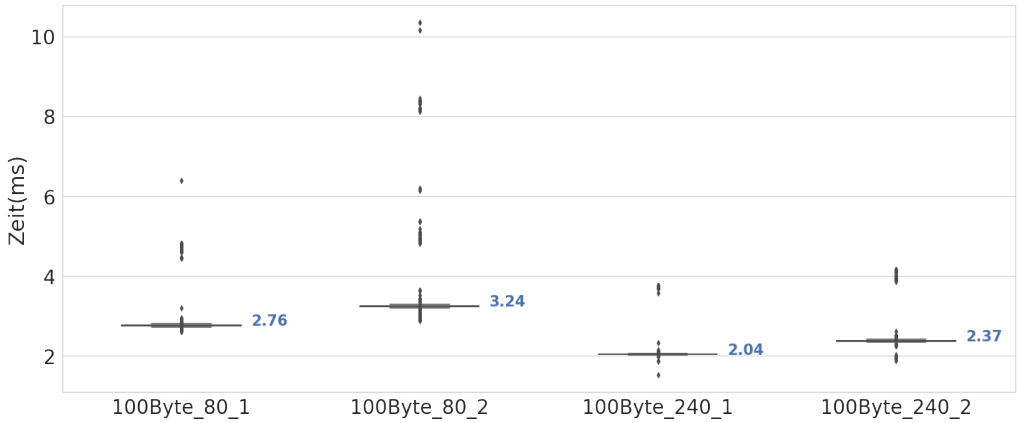
einflusst. Sie liegt bei einem 240 MHz-Kern bei 33,96 ms und bei zwei 240 MHz-Kernen beträgt sie 31,35 ms.

Insgesamt ergibt sich das Bild, dass die Auswahl der Taktrate einen entscheidenden Einfluss auf die Geschwindigkeit von MQTT hat, die Wahl der Kerne hingegen weniger. Am längsten braucht QoS 2 bei 1000 Byte mit einem Kern und einer Taktrate von 80 MHz mit 40,35 ms. Die schnellste Nachricht mit QoS 0 hat hingegen nur 16,31 ms benötigt. Auffällig ist, dass bei QoS 0 kein Unterschied zwischen 100 Byte und 1000 Byte vorhanden ist.

7.4 MQTT-SN-Protokoll

MQTT-SN nutzt, anders als MQTT, UDP-Pakete zur Übertragung. Dies hat einen direkten Einfluss auf die Geschwindigkeit.

7.4.1 MQTT-SN QoS -1

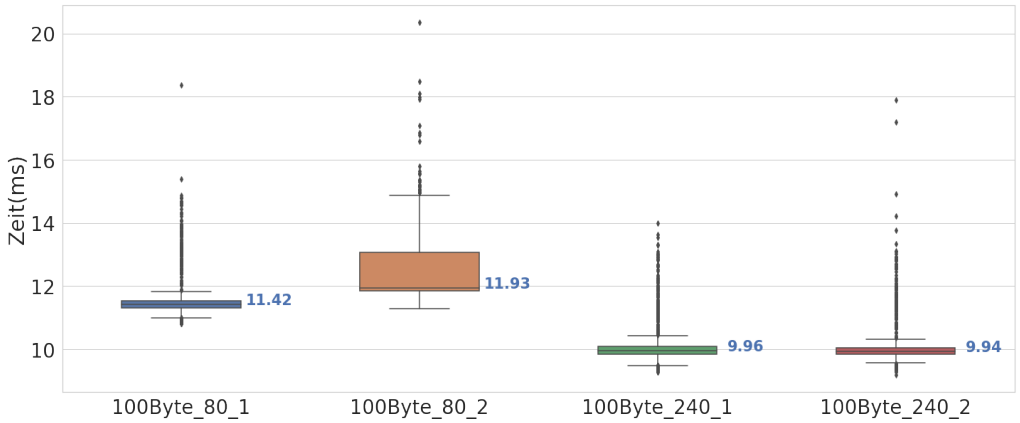


(a) 100 Byte.

Abbildung 54: Versuchsergebnisse MQTT-SN QoS -1.

Die Ergebnisse des Versuchs mit QoS -1 sind in der Abbildung 54 zu sehen. Die Nutzung eines zusätzlichen Kerns wirkt sich leicht negativ auf die benötigte Zeit aus. Trotzdem liegen die Werte ziemlich nah beieinander. Bei einem Takt von 80 MHz und einem Kern wurden 2,76 ms gemessen. Bei zwei Kernen erhöht sich die Zeit auf 3,24 ms. Ein Prozessortakt von 240 MHz braucht bei einem Kern 2,04 ms. Zwei Kerne benötigen bei diesem Takt 2,37 ms.

7.4.2 MQTT-SN QoS 0

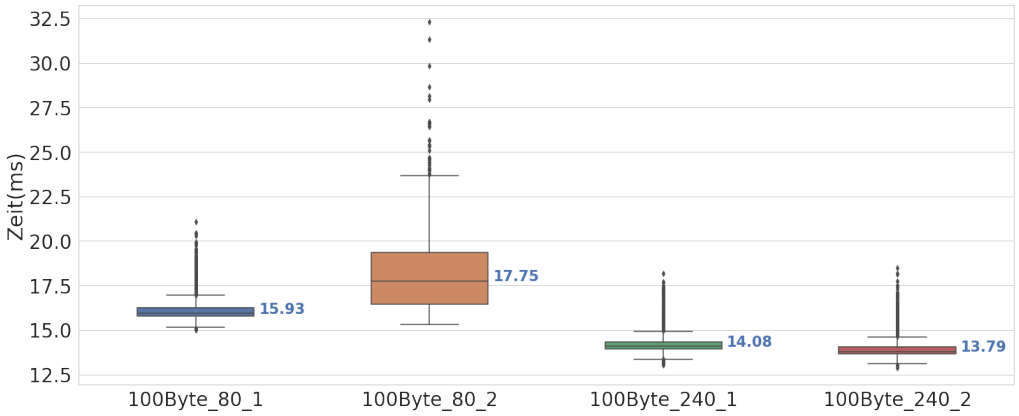


(a) 100 Byte.

Abbildung 55: Versuchsergebnisse MQTT-SN QoS 0.

Die Übertragungszeiten steigen stark an bei der Nutzung von MQTT-SN mit QoS 0. Im Vergleich zu QoS -1 um 8,66 ms auf 11,42 ms bei einem Kern und 80 MHz. Bei zwei Kernen sind es 11,93 ms. Nutzt man 240 MHz sinkt die Zeit um mehr als 1 ms. Der positive Nutzen des zweiten Kerns bei 240 MHz ist fast nicht vorhanden. Insgesamt ist es möglich eine Nachricht in 9,94 ms zu versenden.

7.4.3 MQTT-SN QoS 1



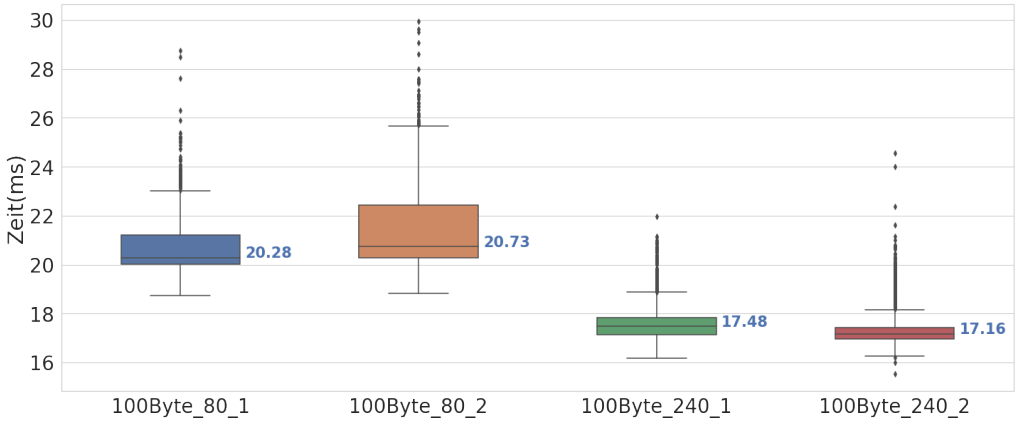
(a) 100 Byte

Abbildung 56: Versuchsergebnisse MQTT-SN QoS 1.

Die Nutzung von QoS 1 erhöht die Übertragungszeit auf 15,93 ms bei 80 MHz und einem Kern. Sein Pendant mit zwei Kernen braucht 17,75 ms. Die Nutzung eines höheren Prozessortakts bringt hier erneut einen Vorteil. Bei einem Kern werden 1,85 ms weniger benötigt. Bei zwei Kernen sind es sogar 3,96 ms.

Die Nutzung des zweiten Kerns hat einen negativen Effekt bei 80 MHz. Bei 240 MHz existiert nur ein sehr kleiner Vorteil.

7.4.4 MQTT-SN QoS 2



(a) 100 Byte.

Abbildung 57: Versuchsergebnisse MQTT-SN QoS 2.

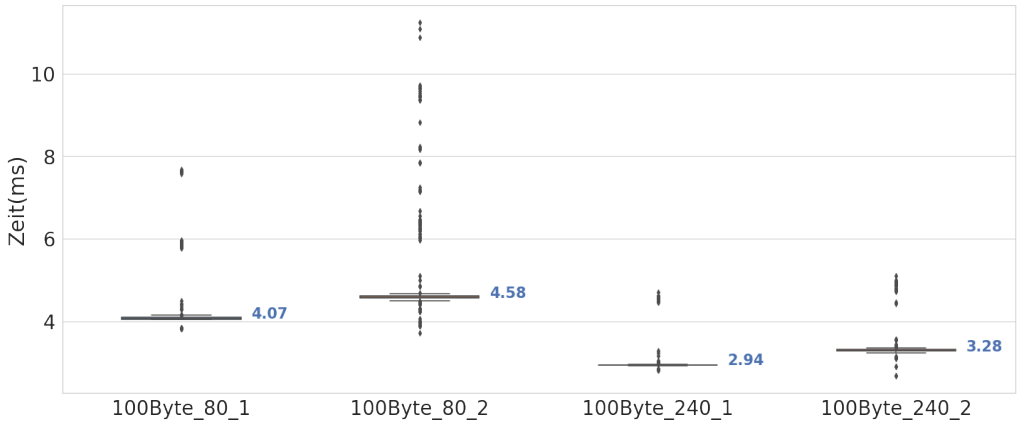
Die letzte Variante der Versuchsreihe ist MQTT-SN QoS 2. Die Übertragung benötigt in dieser Stufe am meisten Zeit im Vergleich zu allen anderen QoS-Stufen von MQTT-SN.

Erneut ist der Unterschied zwischen einem oder zwei Kernen sehr gering. Er beträgt bei 80 MHz nur 0,45 ms und bei 240 MHz 0,32 ms.

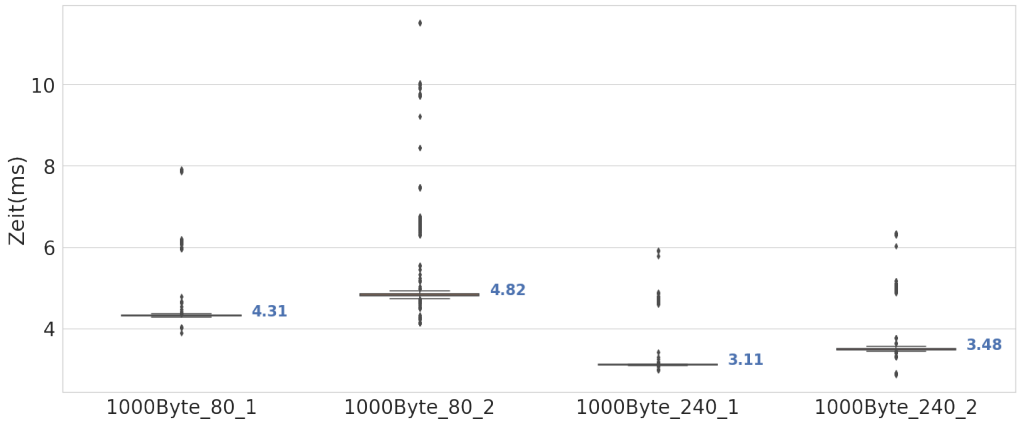
7.5 CoAP-Protokoll

Das Protokoll CoAP hat zwei verschiedene Arten von Nachrichten. Dieser Abschnitt stellt das Ergebnis der Zeitmessung beider Arten vor.

7.5.1 CoAP-NON



(a) 100 Byte.



(b) 1000 Byte.

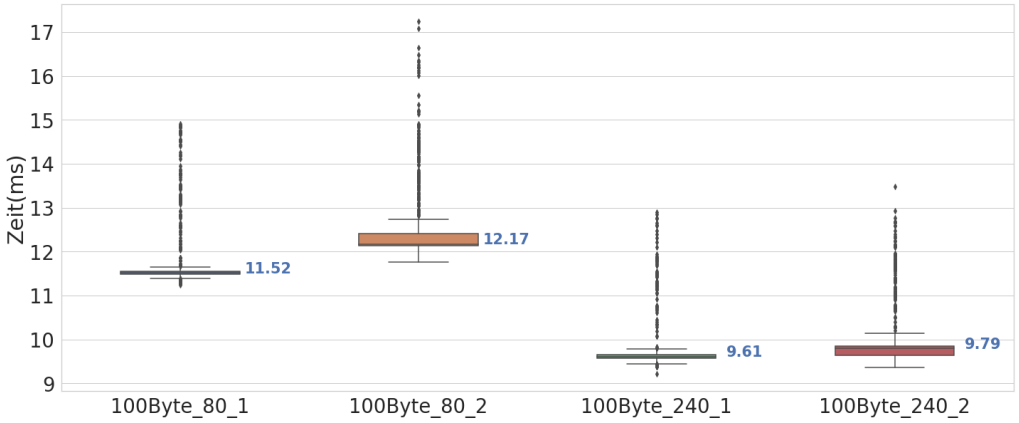
Abbildung 58: Versuchsergebnisse CoAP-NON.

Bei CoAP-NON erhält der Sender keine Bestätigung über den Empfang der Nachricht. Somit sind hier, ähnlich wie bei MQTT-SN QoS -1, sehr kurze Zeiten gemessen worden. Die Übertragungszeit beträgt 4,07 ms bei 80 MHz und einem Kern. Das Payload hatte dabei die Größe von 100 Byte. Nimmt man zwei Kerne liegt die Zeit bei 4,58 ms. Sie erhöht sich vergleichbar wie bei MQTT-SN. Nutzt man die 240 MHz des ESP32 sinkt die Zeit auf 2,94 ms bei einem Kern und 3,28 ms bei zwei Kernen.

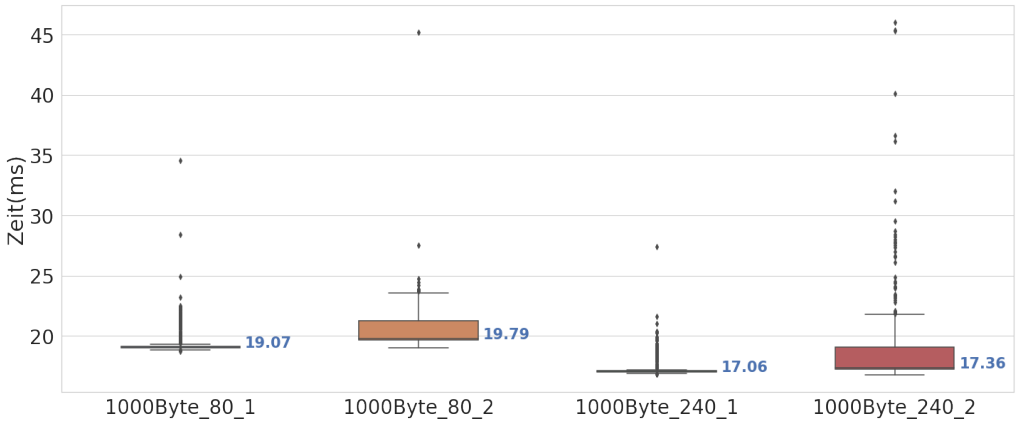
Der Unterschied zu einem Payload von 1000 Byte ist die Erhöhung der Übertragungszeit um weniger als 0,21 ms. Ein spürbarer Unterschied ist hier ebenfalls nicht vorhanden. Die Nutzung eines zweiten Kerns verlängert die Zeit bei allen Einstellungen im geringen Ausmaß. Bei 80 MHz wurden 0,24 ms weniger benötigt, unabhängig von der Anzahl der Kerne. Bei zweo 240 MHz-Kernen beträgt der Unterschied 0,20 ms und nur 0,17 ms bei einem 240 MHz-Kern.

Es zeigt sich erneut, dass bei der unzuverlässigen Nachrichtenübertragung kein großer Unterschied zwischen 100 Byte und 1000 Byte besteht. Trotzdem braucht ein CoAP-NON länger als eine MQTT-SN-Nachricht mit QoS -1. Beide Protokolle nutzen dabei nur ein UDP-Paket.

7.5.2 CoAP-CON



(a) 100 Byte.



(b) 1000 Byte.

Abbildung 59: Versuchsergebnisse CoAP-CON.

Der Erhalt von CoAP-CON wird vom Empfänger mit einer Antwort bestätigt. Die benötigte Zeit erhöht sich bei 100 Byte und 80 MHz mit einem Kern auf 11,52 ms. Zwei Kerne haben keine positive Auswirkung auf die Nachrichtenübertragung und benötigen 12,17 ms. Die bessere Optimierung ist die Anhebung des Prozessortakts. Dann werden bei einem Kern 9,61 ms erreicht. Bei zwei Kernen sind es 9,79 ms.

Interessant ist, dass die Erhöhung des Payloads auf 1000 Byte auch die Zeit linear steigert. Berechnet man von allen Zeitdifferenzen zwischen 100 Byte und 1000 Byte den Mittelwert, erhält man 7,54 ms mit einer Standardabweichung von nur 0,06 ms.

Die Messergebnisse zeigen, dass die Differenz zwischen 100 Byte und 1000 Byte bei keiner geforderten Antwort häufig keinen zeitlichen Unterschied aufweisen. Betrachtet man den Aufruf, den die Protokolle verwenden, wird klar, dass dieses Verhalten mit der Art, wie Daten versendet werden, zusammenhängt. Der Quellcode für das Versenden

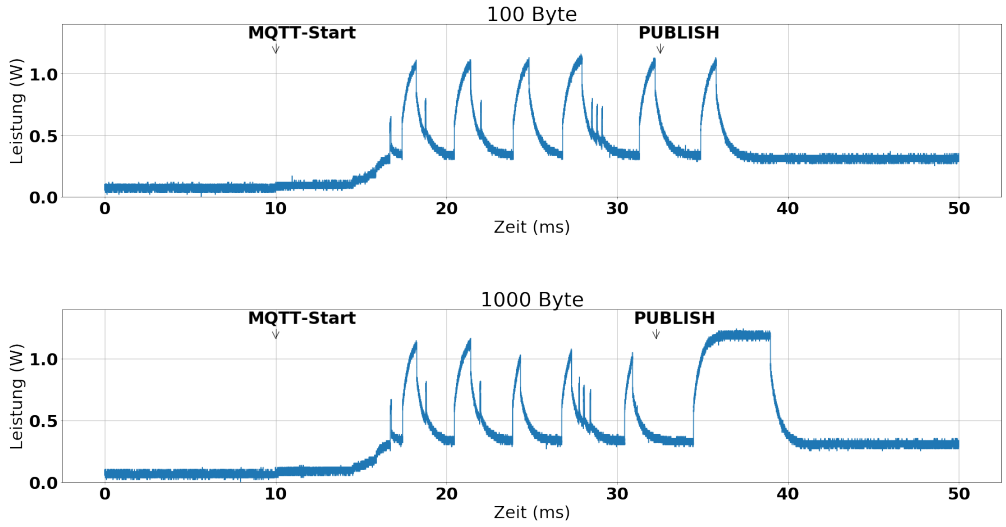


Abbildung 60: Stromverbrauch bei 100 Byte oder 1000 Byte MQTT QoS 0.

von Nachrichten auf den unteren Schichten ist nicht frei einsehbar [32]. Die Annahme war, dass die Daten versendet wurden, wenn die Übetragungsfunktion abgeschlossen ist. Untersucht man den Quellcode und den Stromverbrauch bei MQTT QoS 0 entsteht der Eindruck, dass diese Annahme falsch war. Die Strommessung in der Abbildung 60 zeigt den Stromverbrauch bei MQTT QoS 0 mit 80 MHz und einem Kern. Der Stromverbrauch ist beim Senden der 1000 Byte-Nachricht höher. Allgemein dauert die Übertragung auch länger als bei 100 Byte. Scheinbar werden Nachrichtenpakete nur an die untere Schicht weitergegeben, ohne sie direkt zu versenden. Die Funktion schließt trotzdem ab, sobald die Übergabe stattgefunden hat. Bei 1000 Byte müssten mehr Daten an die untere Schicht übergeben werden. Das müsste zu messbaren Unterschiede führen. Ein Blick in den Quellcode zeigt jedoch, dass die untere Schicht die Daten nicht sofort erhält. Vielmehr wird nur ein Pointer zu den entsprechenden Daten angehängt. Das heißt, die untere Schicht holt sich die Daten erst, sobald sie diese benötigt. Deshalb ist auch kein Unterschied bei der Payloadgröße zwischen MQTT QoS 0 oder ähnlichen Nachrichten ohne Antwort gemessen worden.

Es existiert keine Möglichkeit den Puffer des WLANs abzufragen, wie viele Nachrichten noch verschickt werden müssen. Deshalb ist es schwierig die Zeiten zu erfassen, bei denen keine Antwort benötigt wird. Vielmehr ist festzuhalten, dass dieses Verhalten des ESP32 zu Problemen führen kann. Wird der Tiefschlaf direkt nach dem Versenden aktiviert, wird im schlimmsten Fall keine Nachricht versendet.

7.5.3 Ergebnisse der Nachrichtenübertragung

Die Abbildung 61 zeigt noch einmal alle Protokolle und ihre gemessenen Zeiten in einer Übersicht. Es ist klar erkennbar, dass die UDP-basierten Protokolle die Spitze übernehmen. Das MQTT mit TCP-Paketen befindet sich eher in der unteren Hälfte der Übersicht.

Die Frage nach einem passenden Protokoll zur Übertragung von Nachrichten ist nicht einfach zu beantworten. Durch das Verhalten, Pakete nur an die untere Schicht zu übergeben, besteht immer die Gefahr, zu früh in den Tiefschlaf zu wechseln. Sollte das Risiko eingegangen werden, kann MQTT-SN mit den schnellsten Zeiten dienen. CoAP benötigt nur wenige Millisekunden mehr.

Geht es darum sicher zu sein, dass die Nachricht auch verschickt wurde, sollten CoAP-CON versandt werden. Sie benötigen nur 9,61 ms und beinhalten eine Empfangsbestätigung, dass die Nachricht angekommen ist. Das MQTT-Protokoll braucht bei QoS 0 mindestens 16,31 ms. Eine MQTT-Nachricht mit QoS 1 benötigt bei einem 240 MHz-Kern 17,51 ms. Dabei kann sich das Programm auch hier sicher sein, dass die Nachricht versandt und empfangen wurde.

Beim Vergleich der Mediane der Nachrichten, die eine Bestätigung erhalten, fällt ein möglicher linearer Zusammenhang zwischen benötigter Zeit und Payload-Größe auf. An einem einfachen Beispiel lässt sich die Vermutung überprüfen:

Eine CoAP-CON braucht bei einem 240 MHz-Kern und einem Payload von 100 Byte 9,61 ms. Verschickt man eine Nachricht mit 1000 Byte erhöht sich die Zeit auf 17,06 ms. Berechnet man nun die wahrscheinlich benötigte Zeit für 550 Byte, ergibt sich eine Zeit von 13,71 ms. Wird die reale Messung mit 550 Byte wiederholt, erhält man einen Median von 13,42 ms. Es besteht nur eine Abweichung von 2% zum errechneten Ergebnis. Deshalb kann die Annahme, dass die Payloadgröße die Übertragungszeit linear beeinflusst, bestätigt werden.

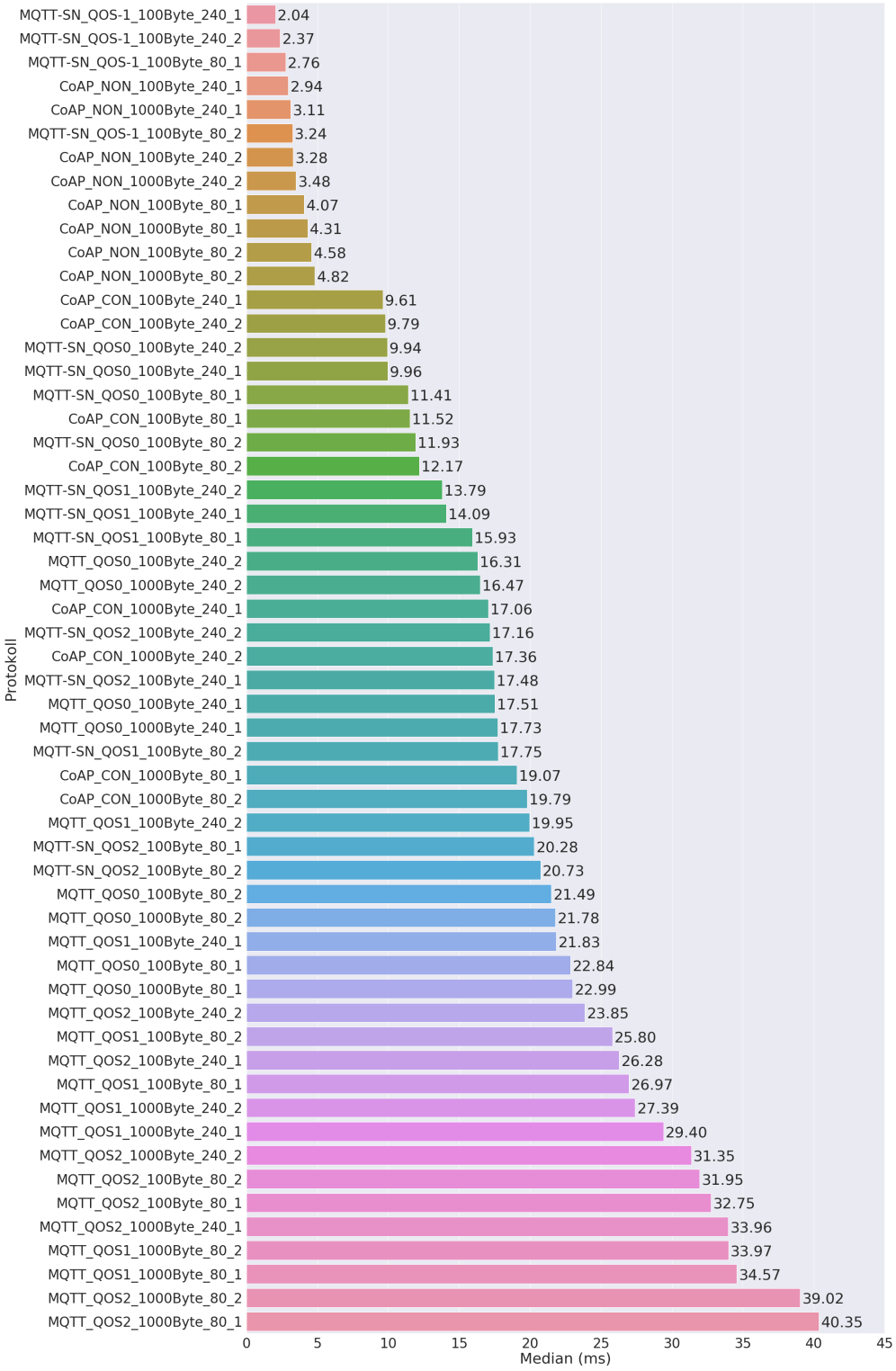
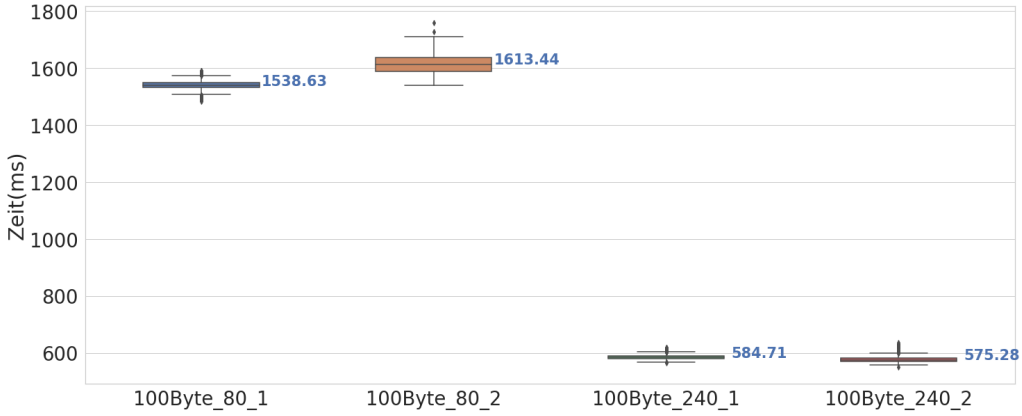


Abbildung 61: Ergebnisse der Nachrichtenübertragung.

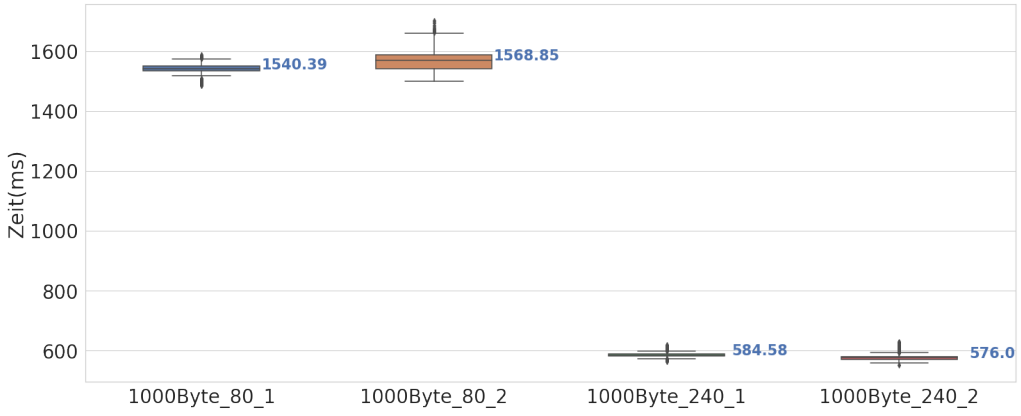
7.6 Sicherheitsprotokolle

Im folgenden Abschnitt wird die Übertragungszeit der Sicherheitsprotokolle TLS und DTLS betrachtet. Hierzu wird eine MQTT- oder CoAP-Nachricht mit dem entsprechenden Sicherheitsprotokoll verschickt und die Zeit gemessen.

7.6.1 MQTT-TLS QoS 0.



(a) 100 Byte.

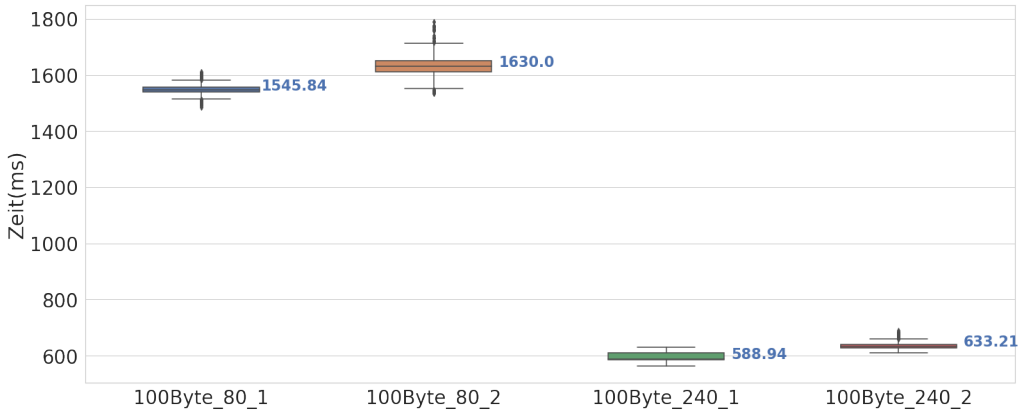


(b) 1000 Byte.

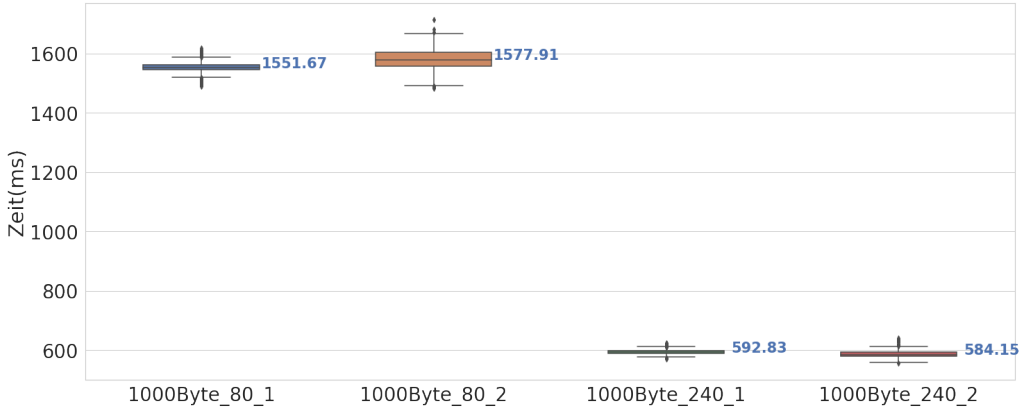
Abbildung 62: Versuchsergebnisse MQTT-TLS QoS 0.

Bei MQTT-TLS mit QoS 0 steigt die Übertragungszeit deutlich an. Wurde zuvor ohne TLS eine Zeit von 22,84 ms mit einem 80 MHz-Kern bei 100 Byte benötigt, liegt diese nun bei 1513,63 ms. Zwei Kerne brauchen 1613,44 ms. Die Nutzung von 240 MHz senkt auch hier die Zeit, bei einem Kern auf 584,71 ms. Bei zwei Kernen sind es 575,28 ms. Der Unterschied zwischen 100 Byte und 1000 Byte ist sehr gering.

7.6.2 MQTT-TLS QoS 1



(a) 100 Byte.

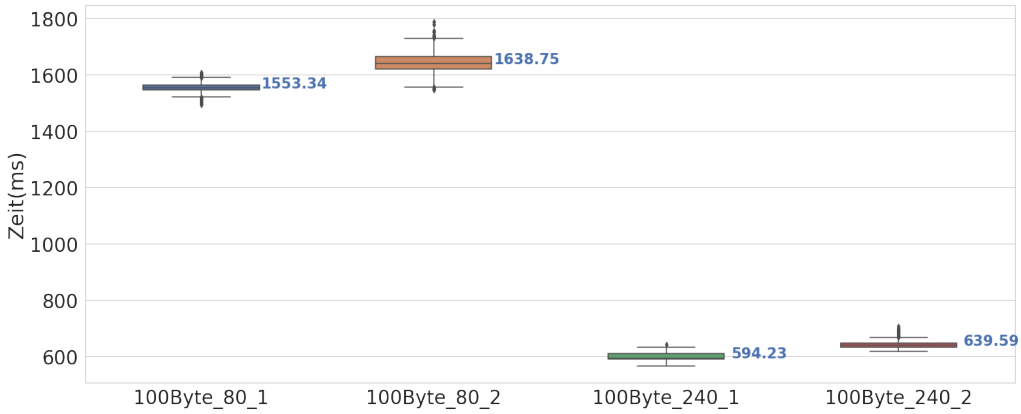


(b) 1000 Byte.

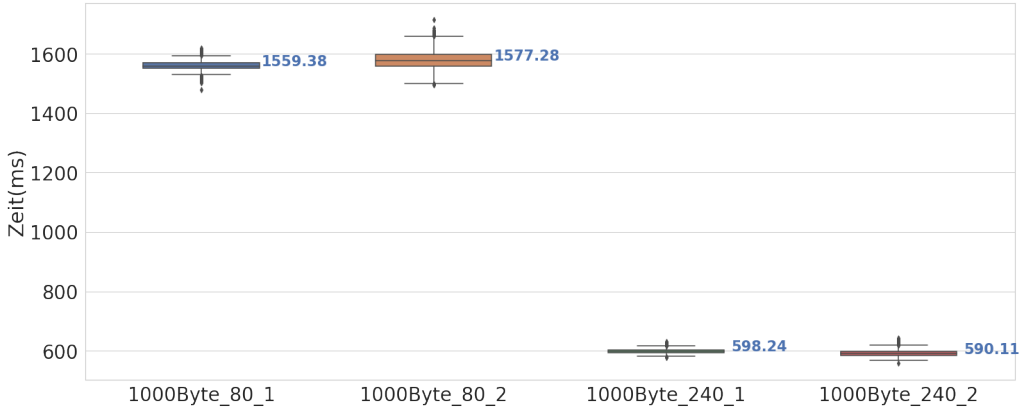
Abbildung 63: Versuchsergebnisse MQTT-TLS QoS 1.

Die Nutzung der höheren Stufe 1 verlängert die Zeit im Vergleich zu QoS 0 nur sehr gering. Bei 100 Byte mit einem 80 MHz-Kern sind es nur 7,21 ms. Der größte Zeitzuwachs liegt bei 100 Byte mit zwei Kernen vor. Hier werden 16,56 ms bei 80 MHz und 57,93 ms bei 240 MHz mehr benötigt.

7.6.3 MQTT-TLS QoS 2



(a) 100 Byte.

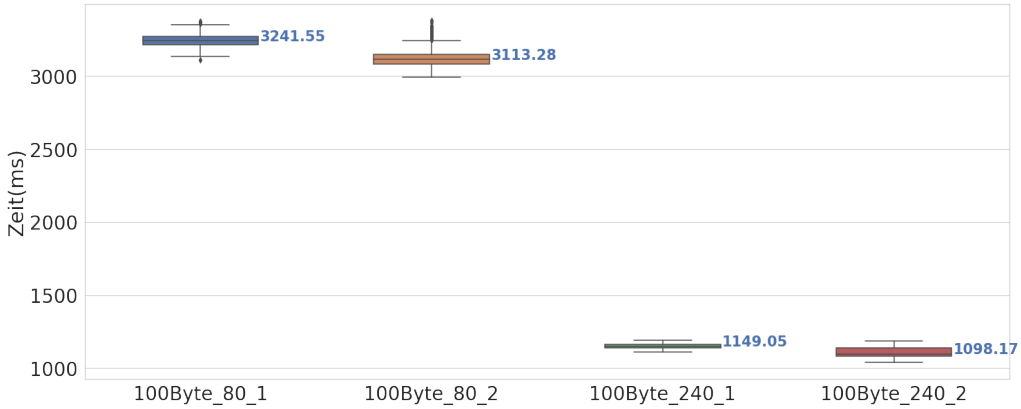


(b) 1000 Byte.

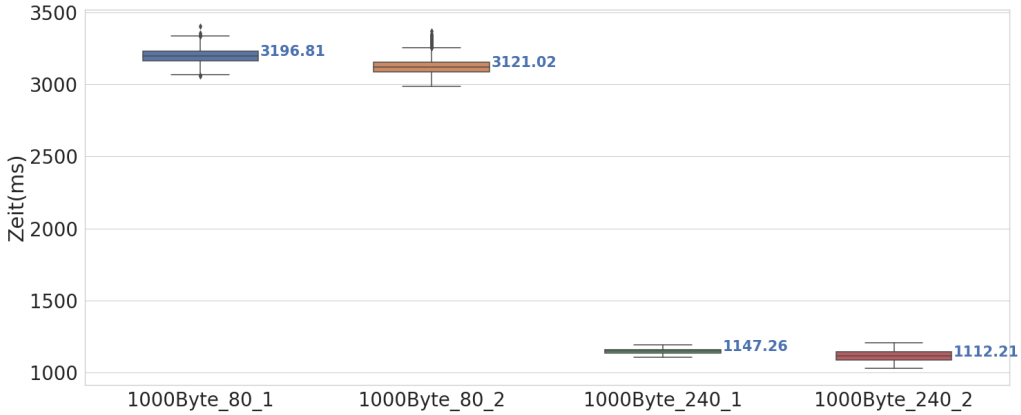
Abbildung 64: Versuchsergebnisse MQTT-TLS QoS 2.

Die letzte QoS-Stufe benötigt nicht viel mehr Zeit als MQTT-TLS QoS 1. Bis auf den Ausreißer von 100 Byte mit einem 80 MHz-Kern, werden nur wenige Millisekunden mehr benötigt. Es zeigt sich, dass nicht die Übertragung der Nachricht den größten Einfluss auf die benötigte Zeit hat. Vielmehr müssen die Verfahren von TLS diese enorme Steigerung der benötigten Zeit verursachen. Dabei spielt die Payloadgröße eine untergeordnete Rolle. Erkennbar ist dies daran, dass oftmals 100 Byte schneller sind. Es liegt somit eine große Varianz beim Aufbau einer TLS-Verbindung vor

7.6.4 CoAP-NON mit DTLS



(a) 100 Byte.

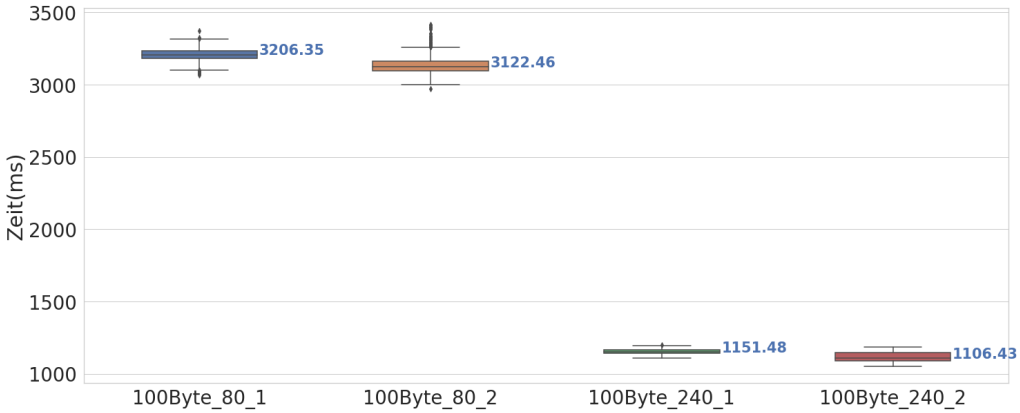


(b) 1000 Byte.

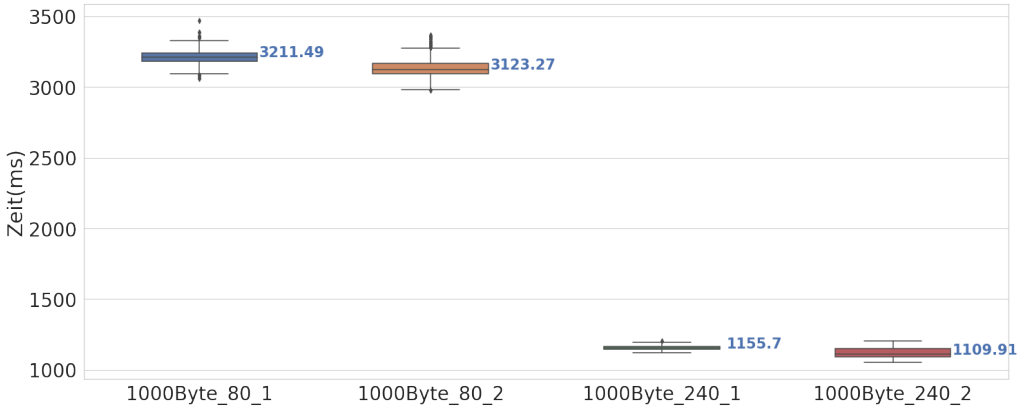
Abbildung 65: Versuchsergebnisse CoAP-NON mit DTLS.

Die längste Übertragungszeit lässt sich bei DTLS beobachten. Bei den CoAP-NON werden mit 80 MHz mehr als 3 s gemessen. Sobald die Taktrate des Prozessors auf 240 MHz erhöht wird, fällt die benötigte Zeit auf knapp 1,13 s. Eine große Differenz zwischen 100 Byte und 1000 Byte ist wie beim Pendant ohne DTLS, nicht gegeben.

7.6.5 CoAP-CON mit DTLS



(a) 100 Byte.



(b) 1000 Byte.

Abbildung 66: Versuchsergebnisse CoAP-CON mit DTLS.

CoAP-CON mit DTLS benötigen fast die gleiche Zeit zur Übertragung ihrer Nachricht wie CoAP-NON. In Abbildung 66 liegen die Werte mit 80 MHz bei mehr als 3 s. Bei 240 MHz sinken die Übertragungszeiten auf 1,1 s.

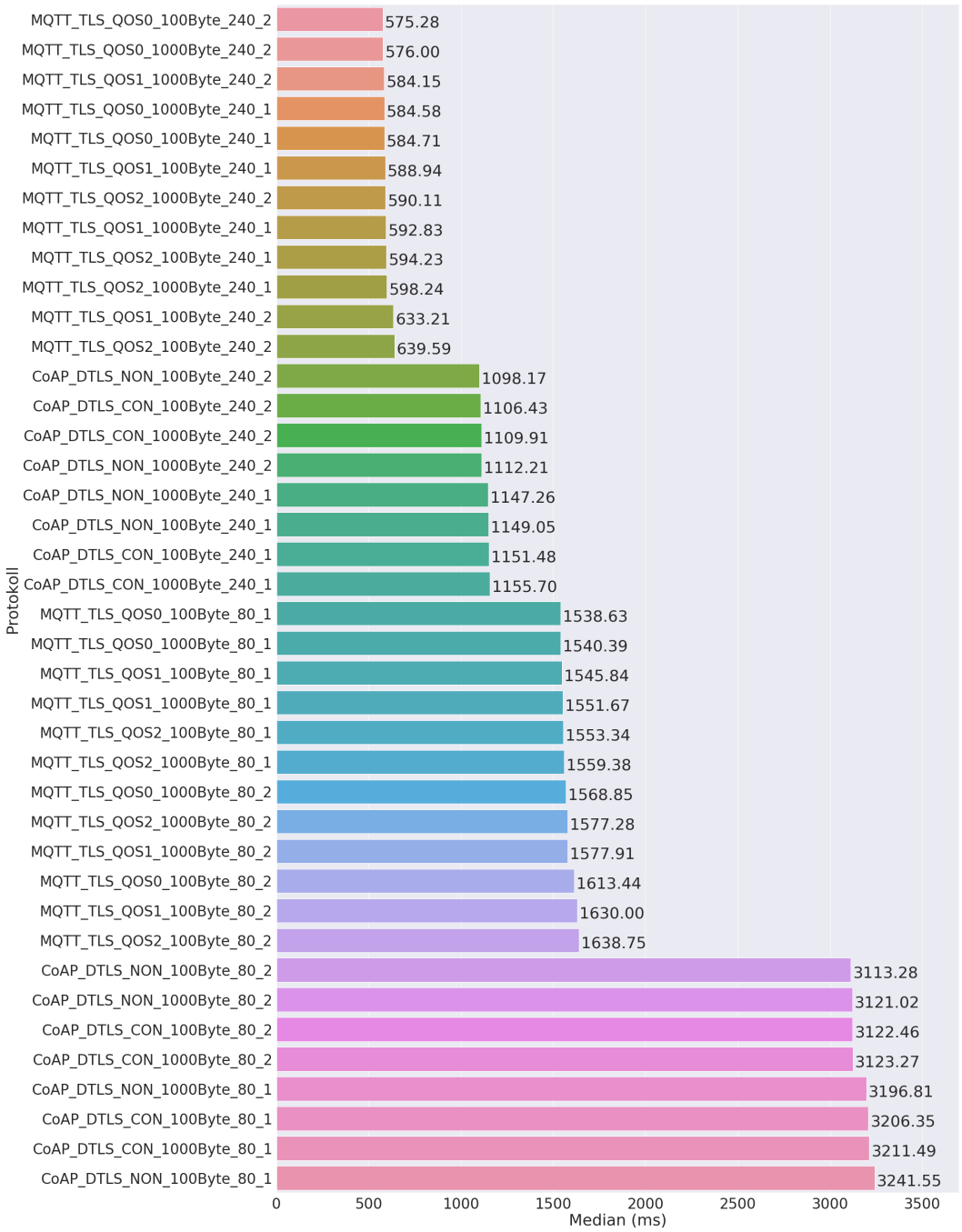


Abbildung 67: Ergebnisse der Nachrichtenübertragung mit Verschlüsselung.

Bereits im Kapitel Stand der Forschung wurden TLS und DTLS für eingeschränkte Geräte als ungeeignet bezeichnet [5]. Die Abbildung 67 zeigt deutlich, warum die Autoren zu dieser Einschätzung kommen. Die beste Zeit erreicht MQTT-TLS QoS 0.

Des Weiteren ist ersichtlich, dass ein Prozessortakt von nur 80 MHz das Verfahren maßgeblich verlängert. Die schlechteste Messung wurde mit CoAP mit DTLS bei 80 MHz erreicht. Die Zeit betrug mehr als 3,1 s.

Zur Überprüfung des Hintergrunds der enormen Zunahme der benötigten Zeit wurde eine Messung mit dem Logikanalysator durchgeführt. Hierzu wurde das Beispiel MQTT mit TLS und zwei 240 MHz-Kernen genutzt.

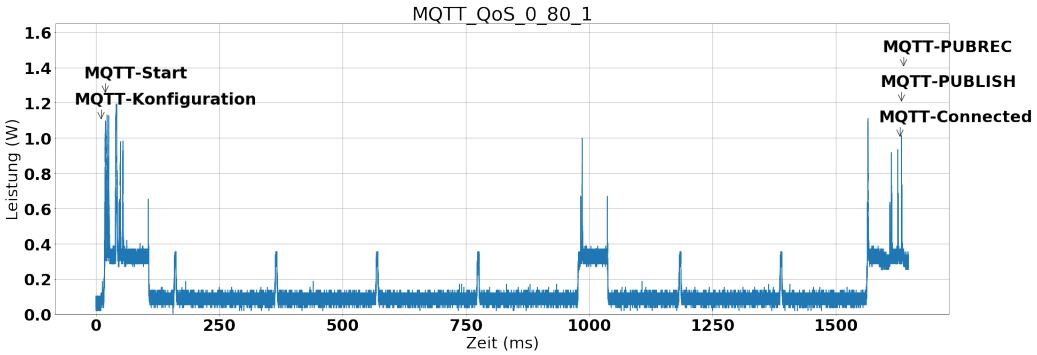


Abbildung 68: Strommessung einer MQTT-Nachricht mit TLS.

Die Abbildung 68 zeigt, dass die meiste Zeit für den TLS-Handshake benötigt wird. Das Senden der Nachricht verbraucht weniger als 10 ms.

Das TLS-Protokoll definiert eine Alternative zu der hier verwendeten Variante. Diese muss nicht immer wieder einen kompletten Handshake durchführen. Die Aufnahme einer alten Sitzung ist eine Möglichkeit, viel Zeit zu sparen. Hierbei muss nur einmal der komplette Handshake durchgeführt werden. Anschließend wird die Sitzung gespeichert und bei einer erneuten Verbindungsaufnahme genutzt. Der ESP32 besitzt noch nicht die Möglichkeit, diese Funktion zu nutzen.

In [22] führen die Autoren bspw. einen Versuch an einem Sensor durch, der eine DTLS-Verbindung aufbauen muss. Die Zeit zum Aufbau eines Full-Handshake betrug 5690 ms. Mit dem Einsatz der Wiederaufnahme der Sitzung konnte die Zeit auf 160 ms gesenkt werden. Das entspricht einer Verbesserung von 97%.

Zusammenfassend lässt sich sagen, dass der Einsatz von TLS oder DTLS in der jetzigen Form schwierig für ein batteriebetriebenes Gerät ist. Die meiste Zeit der aktiven Phase verbringt der Mikrocontroller mit dem Handshake. Sobald jedoch die benötigte Funktion der Wiederaufnahme einer Sitzung umgesetzt wurde, kann die Zeit deutlich verbessert werden.

7.7 Prozessoreinstellung

Der Prozessor des ESP32 kann unterschiedlich konfiguriert werden. Zur Auswahl stehen ein oder zwei Kerne mit entweder 80, 160 oder 240 MHz. In dieser Arbeit wurden die unterschiedlichen Konfigurationen untersucht. In diesem Unterkapitel soll die Frage geklärt werden, welche Einstellung für die Verwendung eines Systems geeignet ist.

In der Bootphase wird nur der erste Kern mit einer Taktrate von 80 MHz genutzt. Es wurde gezeigt, dass das Erhöhen der Taktrate auf 240 MHz eine Verbesserung um 35% ergibt.

Die Assoziierung mit einem WLAN-AP zeigt deutlich, dass der Prozessortakt einen starken Einfluss auf die benötigte Zeit hat. So konnte die Zeit in der höchsten Einstellung bei einer verschlüsselten Verbindung stark verringert werden. Bei einem Kern und 240 MHz lag die Verbesserung bei 59%.

Die letzte Untersuchung zeigte den Einfluss der Konfiguration auf MQTT, MQTT-SN und CoAP. Für den Vergleich wurde die durchschnittliche Zeit aller Mediane der verschiedenen Prozessoreinstellungen gebildet. Anschließend wurde die prozentuale Abweichung von der langsamsten Konfiguration gebildet.

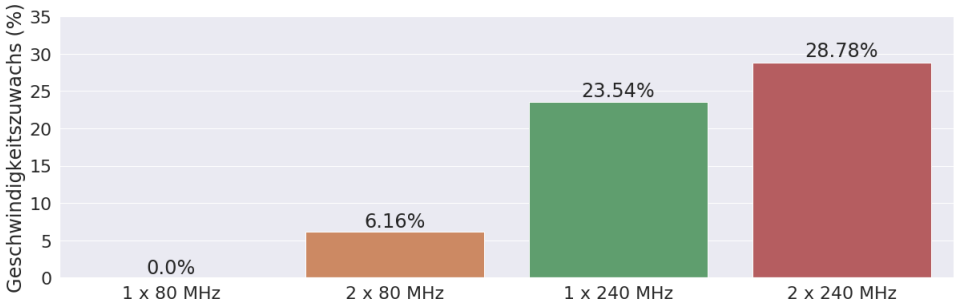


Abbildung 69: Geschwindigkeitszuwachs einer Nachrichtenübertragung.

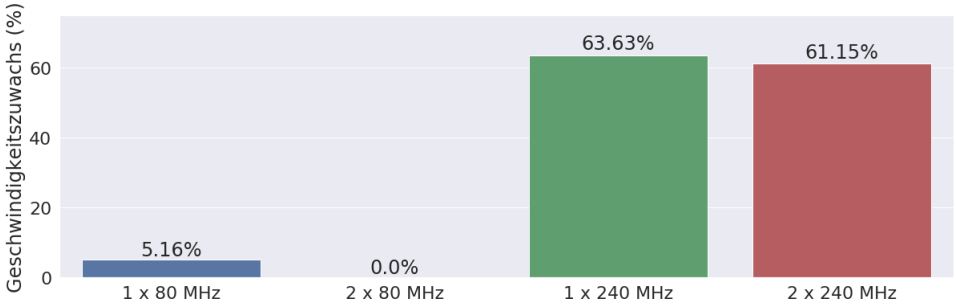


Abbildung 70: Geschwindigkeitszuwachs einer verschlüsselten Nachrichtenübertragung.

Die Abbildungen 69 und 70 zeigen, dass bei einer nicht verschlüsselten Verbindung die besten Zeiten durch zwei 240 MHz-Kerne erreicht werden. Die Alternative mit nur

einem Kern liegt dicht dahinter. Handelt es sich um eine verschlüsselte Verbindung, ist der prozentuale Geschwindigkeitszuwachs mit einem 240 MHz-Kern am größten.

Zusammenfassend lässt sich sagen, dass die Wahl eines hohen Prozessortakts förderlich für die Phasen des Mikrocontrollers ist. Folglich sollte die Einstellung mit 240 MHz der mit 80 MHz vorgezogen werden. Der positive Nutzen des zweiten Kerns ist hingegen nur bei einer unverschlüsselten Nachrichtenübertragung messbar. Alle anderen Testergebnisse zeigten keine oder nur leichte negative Auswirkungen auf die benötigte Zeit. Demzufolge wird, wenn es rein nach dem Zeitaspekt geht, ein 240 MHz-Kern empfohlen.

7.8 Energiebedarf

Der Energiebedarf des ESP32 wurde in unterschiedlichen Szenarien getestet. Ziel war es herauszufinden, wie viel Strom bei unterschiedlichen Prozessortakten und variierender Kernanzahl verbraucht wird.

Beim ersten Versuch wurde der Stromverbrauch im Idle-Modus getestet. Dabei wurde jedem aktiven Kern aufgetragen, eine unbegrenzte Zeit zu warten.

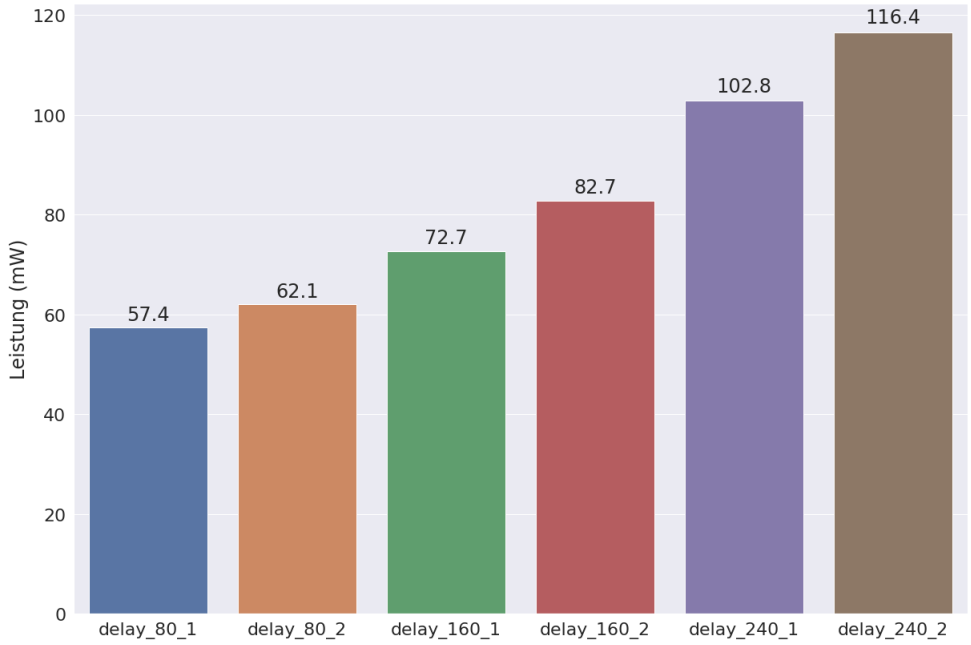


Abbildung 71: Stromverbrauch im Idle.

Die Messergebnisse zeigen, dass bei 80 MHz und einem aktiven Kern mit 57,4 mW am wenigsten Strom verbraucht wird. Das Hinzuschalten des zweiten Kerns erhöht den Strombedarf auf 62,1 mW. Prozentual wird 7,5% mehr Strom benötigt. Nutzt man einen Prozessortakt von 160 MHz, werden bei einem aktiven Kern 72,7 mW gebraucht. Bei zwei aktiven Kernen sind es 82,7 mW. Die Nutzung beider Kerne erhöht den Strombedarf um 12%. Wird die höchste Taktrate mit 240 MHz genutzt, werden bei einem Kern 102,8 mW benötigt. Das ist eine Verdoppelung des Strombedarfs im Vergleich zu einem 80 MHz-Kern. Das Einschalten des zweiten 240 MHz-Kern bewirkt eine Erhöhung des Strombedarfs um 12%. Es werden dann 116,4 mW verbraucht.

Im zweiten Versuch wurden die Kerne mit arithmetischen Aufgaben in einer Dauerschleife belastet. Der dabei gemessene Strombedarf ist in der Abbildung 72 zu sehen. Ein 80 MHz-Kern benötigt mit 82,5 mW 30% mehr Strom bei voller Auslastung. Bei zwei Kernen werden mit 113,7 mW 37% mehr Strom benötigt. Taktet man einen Kern

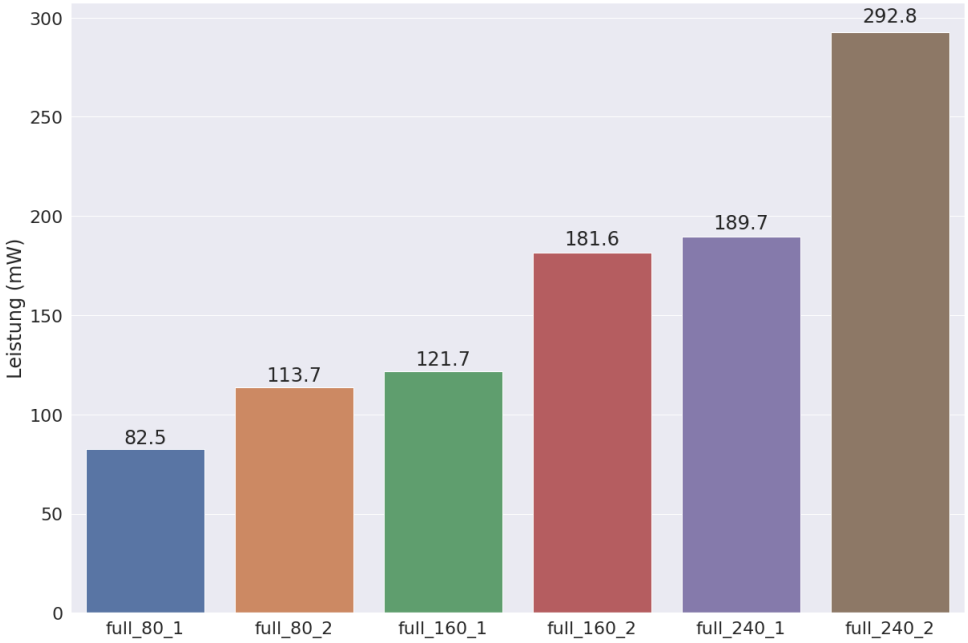


Abbildung 72: Stromverbrauch unter Last.

auf 160 MHz, verbraucht dieser fast so viel Strom wie zwei 80 MHz-Kerne. Der Strombedarf liegt bei 121,1 mW. Die Nutzung von zwei 160 MHz-Kernen erhöht den Strombedarf auf 181,6 mW. Das ist eine Steigerung von 32%. Ein Kern mit 240 MHz verbraucht 189,7 mW. Wird die volle Leistung des ESP32 mit zwei 240 MHz-Kernen abgerufen, werden 292,8 mW benötigt. Eine Steigerung von 35% zu einem Kern.

Insgesamt zeigt sich, dass der Strombedarf nicht linear zum Prozessortakt steigt. Zusätzlich verbraucht ein zweiter Kern unter Belastung zwischen 32% und 37% mehr Strom.

Für den Test des Energiebedarfs eines Jahres kann durch die Zeitmessung und den Strombedarf der verschiedenen Konfigurationen folgende Einschränkung getroffen werden: Zum einen war der positive Nutzen des zweiten Kerns nur in einer Phase ohne Nachrichten-Verschlüsselung messbar. Die Boot- und die Assoziierungsphase, die deutlich mehr Zeit benötigten, profitieren davon nicht. Meist wurde sogar mehr Zeit benötigt. Deshalb wurden die Versuche mit einem Kern durchgeführt. Zum anderen wurde eine statische IP-Adresse benutzt. Dabei wurde ein verschlüsselter WLAN-AP aufgebaut. Ein offener WLAN-AP wird vom BSI nicht empfohlen [13] und wurde dementsprechend nicht berücksichtigt. Der Versuch wurde mit einem Prozessortakt von 240 MHz durchgeführt, da bei 80 MHz über 1,9s benötigt wurden und damit doppelt so viel wie bei 240 MHz.

Für die Übertragung wurde eine CoAP-CON ausgewählt. Sie hat die niedrigste Zeit im Bereich der Nachrichtenübermittlung mit Antwort erreicht. Die Entscheidung eine

Nachricht mit Antwort zu versenden, ergab sich aufgrund des undefinierten Zustands einer Nachricht ohne Antwort. Diese besitzt das Problem, dass nicht festgestellt werden kann, ob die Nachricht verschickt wurde. Mit einer Antwort besteht diese Gefahr nicht. Zusätzlich zu CoAP-CON wird auch eine verschlüsselte Nachricht untersucht. Die schnellste verschlüsselte Nachricht mit Antwort wurde von MQTT-TLS QoS 1 erreicht.

Im Anschluss wurden der durchschnittliche Strombedarf und die Gesamtzeit ermittelt. Dabei brauchte der Versuch mit CoAP-CON 984 ms bei einem durchschnittlichen Strombedarf von 447 mW. Das Protokoll MQTT-TLS QoS 1 benötigte 1684 ms. Der durchschnittliche Strombedarf lag bei 397 mW. Zusätzlich zu den Protokollen wurde auch die Tiefschlafphase gemessen. Hierbei wurden 0,0165 mW verbraucht.

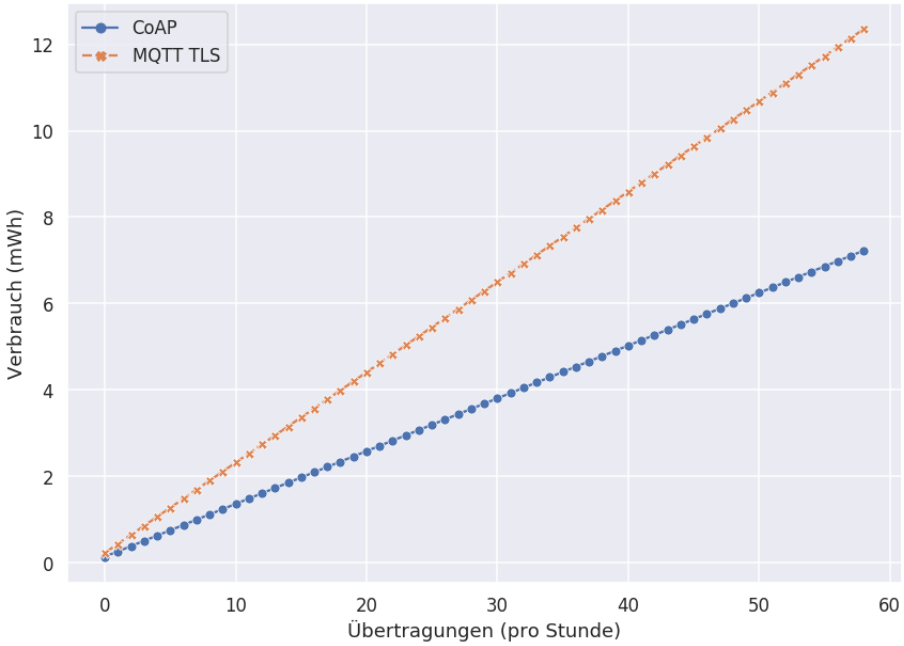


Abbildung 73: Stromverbrauch in Abhängigkeit von der Anzahl der Übertragungen.

Die Abbildung 73 zeigt den wachsenden Strombedarf bei zunehmender Anzahl der Übertragungen pro Stunde. Dieser verhält sich proportional. Die Steigung bei CoAP-CON ist trotz höherem Durchschnittsverbrauch niedriger als bei MQTT-TLS QoS 1. Das liegt an der kürzeren Zeit, die CoAP-CON für eine Übertragung benötigt.

Zusätzlich zum Strombedarf konnte eine Abschätzung der Laufzeit des Mikrocontrollers vorgenommen werden. Hierbei wurde ein einfaches Modell benutzt. Dieses berücksichtigte bestimmte Effekte des Akkus nicht, wie z. B. Temperatur und Selbstentladung.

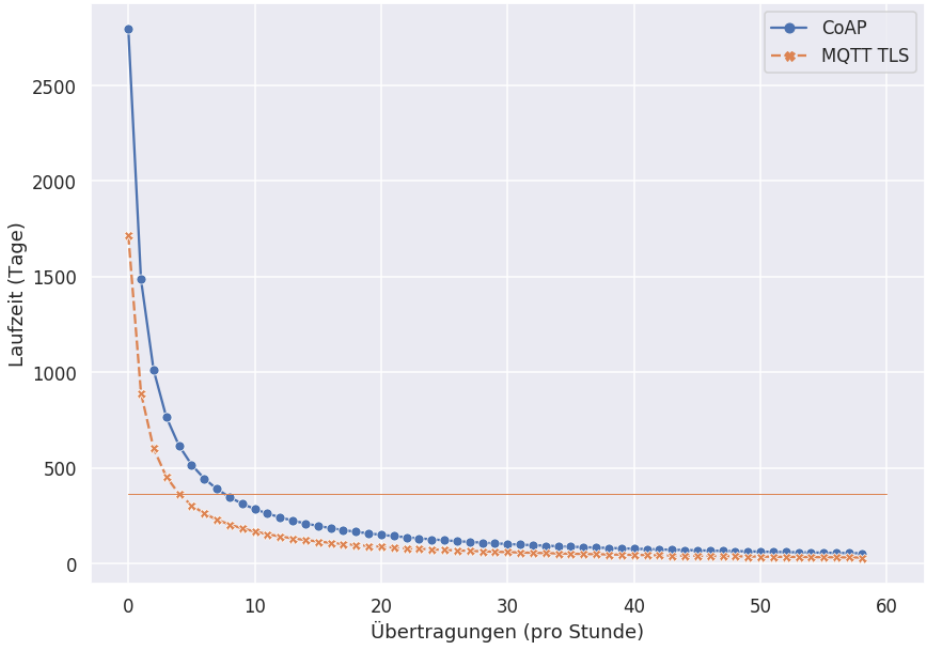


Abbildung 74: Laufzeit in Abhängigkeit von der Anzahl der Übertragungen.

Für das Worst Case-Szenario wurden die Werte eines Panasonic-18650-Lithium-Ion-Akku benutzt. Dieser besitzt eine typische Kapazität von 3350 mAh. Gemäß dem Datenblatt sollte bei 3V und einer Entladungsrate von 0,5 A eine Kapazität von mindestens 3100 mAh vorhanden sein. [25] Es stehen folglich 9,3 mWh zur Verfügung. Anschließend wurde der Stromverbrauch des Mikrocontrollers berechnet. Die Abbildung 74 zeigt die Laufzeit bei unterschiedlicher Anzahl der Übertragungen pro Stunde. Die dargestellten Werte geben eine grobe Richtung an, wie lange ein ESP32 laufen würde. Die horizontale Linie spiegelt die Grenze von einem Jahr Laufzeit wieder.

Mit den CoAP-CON kann eine Laufzeit von einem Jahr erreicht werden, wenn weniger als neun Nachrichten pro Stunde übertragen werden. Bei MQTT-TLS können fünf Nachrichten pro Stunde übertragen werden, um die Laufzeit eines Jahres zu erreichen. Ab zehn Nachrichten pro Stunde sinkt die Laufzeit drastisch ab.

Die Abschätzung zeigt, dass der ESP32 mit einem Panasonic-18650-Akku ein Jahr lang betrieben werden kann. Die Einhaltung der Anzahl der Übertragungen pro Stunde ist wichtig, um die gewünschte Laufzeit zu erreichen.

8 Fazit und Ausblick

Wie die Untersuchungen dieser Arbeit gezeigt haben, spielen die Protokolle und deren optimale Einstellung eine wichtige Rolle. Sie bestimmen die Laufzeit eines batteriebetriebenen Systems. Zur Erreichung dieses Ziels wurde zunächst eine passende Messmethode gesucht. Die Erkenntnisse aus dem Kapitel Stand der Forschung zeigen, dass bestimmte Schwierigkeiten beim Messen existieren. Diese Schwierigkeiten wurden analysiert und es wurden passende Methoden gefunden, die die nötige Genauigkeit besitzen.

Darauf aufbauend wurde mit der Analyse des ESP32 und der Umsetzung der Protokolle gestartet. Die Umsetzung war aufgrund des fehlenden DTLS-Moduls von CoAP etwas aufwendiger als zunächst angenommen. Zusätzlich musste das MQTT-SN-Protokoll auf das ESP-IDF portiert werden. Die anschließende Planung, Ausführung und Auswertung der Versuche führte zu wichtigen Erkenntnissen, die der Laufzeit des ESP32 zuträglich waren.

Dabei wurde gezeigt, dass die Abschaltung der Log-Ausgabe ein wichtiger Faktor ist. Dadurch wurde die Bootphase deutlich verkürzt. Ein weiterer Effekt konnte durch die Anhebung des Prozessortakts erreicht werden.

Die Assoziierung mit einem gesicherten WLAN-AP benötigt die meiste Zeit, im Vergleich zu den anderen Phasen. Es wurde nachgewiesen, dass die Verwendung von DHCP zu einem erhöhten Zeitbedarf führt. Weiterhin wurde der Einfluss der verschiedenen Prozessortakte und Kernanzahlen gezeigt. Dabei kam die Frage auf, ob der Einsatz eines höheren Prozessortakts in Verbindung mit einem zusätzlichen Kern förderlich für die Protokolle ist. Die anschließenden Versuche mit Protokollen für die Nachrichtenübertragung zeigten, dass der Prozessortakt einen maßgeblichen Einfluss auf die Ablaufgeschwindigkeit hat. Der zweite Kern verbrauchte hingegen mehr Strom und verlangsamte in den meisten Fällen den Ablauf. Es ist jedoch zu beachten, dass ein höherer Takt auch einen doppelt so hohen Stromverbrauch bei nur einem Kern bedeutet.

Abschließend wurde eine Abschätzung der Laufzeit mit den optimalen Einstellungen vorgenommen. Hierzu wurde der Strombedarf ermittelt und mit einem einfachen Modell die zu erwartende Laufzeit errechnet. Die Berechnungen wiesen auf eine Laufzeit von weniger als einem Jahr mit einem Panasonic Lithium-Ionen-Akku hin, wenn mehr als acht unverschlüsselte Nachrichten pro Stunde übermittelt werden. Bei einer verschlüsselten Verbindung sind es nur noch fünf Nachrichten pro Stunde, die versendet werden können.

Zusammenfassend lässt sich sagen, dass der Espressif ESP32 als WLAN-basiertes System eingesetzt werden kann. Dafür müssen die vorher festgelegten Limits eingehalten werden. Das minütliche Aufwachen und Versenden von Nachrichten kann hingegen nicht mit dieser Technologie stattfinden, ohne einen deutlichen Einbruch der Batterielaufzeit. Mit Ausblick auf die Zukunft müssen neue Protokolle den Ablauf deutlich beschleunigen. Das Assoziieren mit einem AP benötigt verhältnismäßig viel Zeit. Dieser Prozess ist bei anderen Technologien wie Zigbee oder Long Range Wide Area Network (LoRaWAN) mit einer einmaligen Assoziierung besser gelöst. In Hinblick auf die zunehmende Verschlüsselung von Nachrichten können TLS 1.3 und vor allem die Wiederaufnahme der Sitzung einer verschlüsselten Verbindung dazu beitragen, dass in Zukunft ein sicheres WLAN-basiertes System dieselbe Batterielaufzeit aufweist, wie die bereits genannten konkurrierenden Technologien.

Literatur

[1] Hong Linh Truong Andy Stanford-Clark. Mqtt for sensor networks (mqtt-sn) protocol specification. `http://www.mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf`. 14.11.2013.

[2] S. Bandyopadhyay and A. Bhattacharyya. Lightweight internet protocols for web enablement of sensors using constrained gateway devices. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 334–340, Jan 2013.

[3] Olaf Bergmann. libcoap: Cimplementation of coap. `https://libcoap.net`. 14.01.2019.

[4] N. De Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali. Comparison of two lightweight protocols for smartphone-based sensing. In *2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, pages 1–6, Nov 2013.

[5] Jasenka Dizdarevic, Francisco Carpio, Admela Jukan, and Xavier Masip-Bruin. Survey of communication protocols for internet-of-things and related challenges of fog and cloud computing integration. *CoRR*, abs/1804.01747, 2018.

[6] Espressif. Esp32 hardware design guidelines. `https://www.espressif.com/sites/default/files/documentation/esp32_hardware_design_guidelines_en.pdf`. 03.12.2018.

[7] International Organization for Standardization. Iso/iec 29192-1:2012. `https://www.iso.org/standard/56425.html`. 30.12.2018.

[8] Internet Engineering Task Force. The constrained application protocol (coap). `https://tools.ietf.org/html/rfc7252`. 01.06.2014.

[9] Internet Engineering Task Force. Datagram transport layer security version 1.2. `https://tools.ietf.org/html/rfc6347`. 15.01.2019.

[10] Internet Engineering Task Force. Terminology for constrained-node networks. `https://tools.ietf.org/html/rfc7228#section-4.2`. 11.10.2018.

[11] Internet Engineering Task Force. The transport layer security (tls) protocol version 1.2. `https://tools.ietf.org/html/rfc5246`. 01.08.2008.

[12] Eclipse Foundation. Mqtt 5 progress. `https://mosquitto.org/blog/2018/11/mqtt5-progress/`. 29.11.2018.

[13] Bundesamt für Sicherheit in der Informationstechnik. Wireless lan (isi-wlan). `https://www.bsi.bund.de/DE/Themen/StandardsKriterien/ISi-Reihe/ISi-WLAN/wlan_node.html`. 15.01.2019.

[14] Muhammad Harith Amaran, Nazmin Arif Mohd Noh, Mohd Saufy Rohmad, and Habibah Hashim. A comparison of lightweight communication protocols in robotic applications. 76:400–405, 12 2015.

[15] National Instruments. Datasheet ni 9205. `http://www.ni.com/pdf/manuals/374188a_02.pdf`. 14.01.2019.

[16] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan. Powering the internet of things. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 375–380, Aug 2014.

[17] David L. Jones. The ucurrent a professional precision current adapter for multimeters. `http://eevblog.com/files/uCurrentArticle.pdf`. 14.01.2019.

[18] Paul Kierstead. Mqttsn-gateway gets stuck at 100% cpu intermittently. `https://github.com/eclipse/paho.mqtt-sn.embedded-c/issues/41`. 03.01.2019.

[19] Hyeokjin Kwon, Jiye Park, and Namhi Kang. Challenges in deploying coap over dtls in resource constrained environments. In *Revised Selected Papers of the 16th International Workshop on Information Security Applications - Volume 9503*, WISA 2015, pages 269–280, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[20] Alexander Lagerqvist and Tejas Lakshminarayana. Iot latency and power consumption : Measuring the performance impact of mqtt and coap. Master's thesis, 2018.

[21] Arm Mbed. mbedtls. `https://github.com/ARMmbed/mbedtls`. 03.01.2019.

[22] S. R. Moosavi, T. N. Gia, E. Nigussie, A. Rahmani, S. Virtanen, H. Tenhunen, and J. Isoaho. Session resumption-based end-to-end security for healthcare internet-of-things. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 581–588, Oct 2015.

[23] OASIS. Mqtt version 3.1.1 oasis standard. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf`. 29.04.2014.

[24] OASIS. Mqtt version 5.0. http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. 31.10.2018.

[25] Panasonic. Panasonic ncr18650b. https://www.batteryspace.com/prod-specs/NCR18650B.pdf. 15.01.2019.

[26] Pratik Panda. Inside esp32-pico-d4 – should you use it? http://iot-bits.com/inside-esp32-pico-d4-design/. 15.01.2019.

[27] Saleae. Saleae logic pro 16 usb logic analyzer. http://downloads.saleae.com/specs/Logic+Pro+16+Data+Sheet.pdf. 14.01.2019.

[28] Espressif Systems. Esp-idf component libcoap. https://github.com/espressif/esp-idf/tree/master/components/coap. 14.01.2019.

[29] Espressif Systems. Esp-idf freertos smp changes. https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/freertos-smp.html. 14.01.2019.

[30] Espressif Systems. Esp32 mqtt library. https://github.com/espressif/esp-mqtt. 14.01.2019.

[31] Espressif Systems. Esp32-pico-d4 datasheet. https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4_datasheet_en.pdf. 14.01.2019.

[32] Espressif Systems. Espressif iot development framework. https://github.com/espressif/esp-idf. 14.01.2019.

[33] Espressif Systems. General notes about espidf programming. https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/general-notes.html. 14.01.2019.

[34] Espressif Systems. Nonvolatile storage library. https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/storage/nvs_flash.html. 14.01.2019.

[35] Espressif Systems. Wifi driver. https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/wifi.html. 14.01.2019.

[36] D. Thomas, R. McPherson, and J. Irvine. Power analysis of local transmission technologies. In *2016 12th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, pages 1–4, June 2016.

[37] Cheuk-Wang Yau, Tyrone Tai-On Kwok, Chi-Un Lei, and Yu-Kwong Kwok. *Energy Harvesting in Internet of Things*, pages 35–79. Springer Singapore, Singapore, 2018.