FH Vorarlberg
Vorarlberg University of Applied Sciences

Fog Computing-Framework für ressourcenbeschränkte Systeme

Master's in Mechatronics

Masterthesis zur Erlangung des akademischen Grades

Master of Science in Engineering, MSc

Dornbirn, den 06.09.2020

Erarbeitet von

Schwendinger Martin, BSc

Betreut von

Prof. (FH) Dipl.-Ing. Ritschel Patrick

Durchgeführt bei

System Industrie Electronic GmbH, Lustenau
Dipl.-Ing. (FH) Steglich Rainer, MEng

Kurzreferat

Fog Computing-Framework für ressourcenbeschränkte Systeme

Die cloud-basierte Verarbeitung von Datenströmen von IoT-Geräten ist aufgrund hoher Latenzzeiten für zeitkritische Anwendungen nur beschränkt möglich. Fog Computing soll durch Nutzung der Rechen- und Speicherkapazitäten von lokal vorhandenen Geräten eine zeitnahe Datenverarbeitung und somit eine Verringerung der Latenzzeit ermöglichen. In dieser Arbeit werden Anforderungen an ein Fog Computing-Framework erhoben, das die dynamische Zuweisung und Ausführung von Services auf ressourcenbeschränkten Geräten in einem lokalen Netzwerk zur dezentralen Datenverarbeitung ermöglicht. Zudem wird dieses Framework prototypisch für mehrere Transportkanäle, unterschiedliche Betriebssysteme und Plattformen realisiert. Dazu werden die Möglichkeiten der Skriptsprache Lua und des Kommunikationsmechanismus Remote Procedure Call genutzt. Das Resultat ist ein positiver Machbarkeitsnachweis für Fog Computing-Funktionalitäten auf ressourcenbeschränkten Systemen. Zudem werden eine geringere Latenz und eine Reduktion der Netzwerklast ermöglicht.

Abstract

Fog Computing Framework for Resource-Constrained Systems

Cloud-based processing of data streams from IoT devices is due to high latency limited for time-critical applications. Fog computing is intended to enable timely processing and thus optimize latency by utilizing the computing and storage capacities of locally available devices. In this thesis requirements for a fog computing framework are defined, which allows the dynamic allocation and execution of services on resource limited devices in a local network for decentralized data processing. In addition, this framework is prototypically realized for multiple transport channels, different operating systems and platforms. For this purpose, the possibilities of the scripting language Lua and remote procedure call as a communication mechanism are used. The result is a positive proof of concept for fog computing functionalities on resource-constrained systems. In addition, it enables a lower latency and a reduction of the network load.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Quellcodes

Abkürzungsverzeichnis

ELF Executable Linkable Format. 17
eLua embedded Lua. 18

HTTP Hypertext Transfer Protocol. 16

IoT Internet of Things. 4
IP Internet Protocol. 20

POSIX Portable Operating System Interface. 30

RAM Random-Access Memory. 29
REST Representational State Transfer. 16
ROM Read Only Memory. 18
RPC Remote Procedure Call. 16
RTOS Real-Time Operating System. 19

TCP Transmission Control Protocol. 24

1 Einleitung

Dieses Kapitel beschreibt in Abschnitt 1.1 den Grund und die Motivation für diese Arbeit und in Abschnitt 1.2 wird das Problem angesprochen, das in dieser Arbeit behandelt wird. In Abschnitt 1.3 wird auf die verwendete Methodik eingegangen und in Abschnitt 1.4 wird der Aufbau der Arbeit mittels einer Übersicht über die folgenden Kapitel präsentiert.

1.1 Motivation

Das klassische Cloud Computing-Modell, bei dem gesammelte Daten in die Cloud geschickt werden um dort verarbeitet zu werden, ist aufgrund hoher Latenzzeiten zwischen Datenquelle und Cloud für den Einsatz bei zeitkritischen Anwendungen nur bedingt geeignet. Fog Computing soll dieses Problem adressieren. (Vgl. Bonomi, Milito, Zhu u. a. 2012) Dabei werden lokal vorhandene Rechen- und Speicherkapazitäten benutzt um Daten zu verarbeiten, wodurch sich der Vorteil einer Latenzreduktion ergeben soll um zeitnah auf Ereignisse reagieren zu können. Dadurch ergeben sich spezielle Anwendungsfälle für Fog Computing, beispielsweise Smart Grid und Smart Vehicles, bei denen Komponenten von kleinen Latenzen abhängig sind und von lokaler Datenverarbeitung profitieren. (Vgl. Bonomi, Milito, Natarajan u. a. 2014)

Verarbeitete Daten werden bei Fog Computing-Anwendungen weiterhin in die Cloud gesendet, wo genügend Speicherplatz vorhanden ist um diese für Langzeitanalysen abzuspeichern. Da lediglich bereits verarbeitete Sensorwerte in die Cloud geschickt werden, soll dadurch zusätzlich eine Datenreduktion und somit eine Netzwerkentlastung begünstigt werden. (Vgl. Marquesone u. a. 2017)

1.2 Problembeschreibung

Nach Čolaković und Hadžialić (2018) ist es herausfordernd zu bestimmen, welche Funktionen der jeweiligen Ressource in einem Netzwerk zugewiesen werden sollten. Daher besteht die Notwendigkeit, das Verhalten von IoT-Anwendungen praktisch zu evaluieren, wenn sie auf verschiedenen Schichten der Systemarchitektur ausgeführt werden. Dafür bedarf es eines Softwaresystems, welches mit den im lokalen Netzwerk

vorhandenen Geräten kommunizieren kann und eine Schnittstelle für den Serviceaustausch bietet. (Vgl. Čolaković und Hadžialić 2018, S. 29) Ein solches Softwaresystem kann dann in weiterer Folge in verschiedenen „smarten“ Anwendungsgebieten eingesetzt werden um Services dynamisch bereitzustellen und eine automatische Verteilung von Software zu begünstigen.

In Bachmann (2017) wurde ein Fog Computing-Framework in einem Netzwerk auf Basis von Raspberry Pi's und Docker-Containern vorgestellt. Für ressourcenbeschränkte und unter Umständen heterogene IoT-Geräte ist dies allerdings möglicherweise nur begrenzt einsetzbar.

Für die Nutzung eines solchen Softwaresystems auf ressourcenbeschränkten Geräten lassen sich somit folgende Forschungsfragen formulieren:

- Welche Mechanismen eignen sich für die Verteilung von Services auf ressourcenbeschränkten Geräten in einem Fog-Netzwerk?
- Welche Funktionalitäten sollte ein Fog Computing-Framework bereitstellen, das die Verteilung von Services auf ressourcenbeschränkte Geräte in einem Fog-Netzwerk zulässt?

Zur Beantwortung dieser Forschungsfragen wird im Rahmen dieser Arbeit ein leichtgewichtiges Service-Deployment-Konzept mit Fokus auf Anwendbarkeit auf IoT-Geräten entwickelt. Durch dieses soll in weiterer Folge die dynamische Bereitstellung von Services auf ressourcenbeschränkten Geräten für verschiedene Anwendungsgebiete ermöglicht werden.

1.3 Methode

Zunächst werden relevante Informationen zu der vorliegenden Thematik aus dem aktuellen Forschungsfeld zu Fog Computing gesammelt und extrahiert. Weiters erfolgt eine Analyse von funktionalen und nicht-funktionalen Anforderungen. Darauf aufbauend werden Designentscheidungen für den Aufbau des Fog Computing-Frameworks getroffen und passende Technologien und Werkzeuge ausgewählt. Anschließend wird basierend auf den Designentscheidungen das Fog Computing-Framework prototypisch implementiert, bevor abschließend eine Evaluierung erfolgt.

1.4 Aufbau der Arbeit

Zunächst werden in Kapitel 2 Konzepte beschrieben, die Hintergrundwissen zum Verständnis von der in dieser Arbeit behandelten Thematik liefern. In Kapitel 3 werden relevante wissenschaftliche Arbeiten zum aktuellen Stand der Technik beschrieben und erfasst. Kapitel 4 behandelt Anforderungen an das Fog Computing-Framework.

Diese werden erhoben und darauf aufbauend werden Technologien ausgewählt und Designentscheidungen getroffen. In Kapitel 5 wird mittels der ausgewählten Technologien das Fog Computing-Framework implementiert und die einzelnen Komponenten und Funktionalitäten werden vorgestellt. Darauf folgend wird in Kapitel 6 die Implementierung mittels verschiedener Szenarien evaluiert, bevor abschließend in Kapitel 7 eine Zusammenfassung der Arbeit und ein Ausblick auf zukünftige Arbeiten gegeben wird.

2 Hintergrund

Dieses Kapitel beschreibt wichtige Konzepte, die in dieser Arbeit verwendet werden und gibt einen Überblick über erforderliches Hintergrundwissen. Zunächst wird der Begriff des „Internet of Things“ und dessen Einsatzgebiete vorgestellt. Als nächstes wird das Konzept des „Cloud Computing“ erläutert, um daraufhin Unterschiede zu „Fog Computing“ sichtbar zu machen. Schließlich wird „Edge Computing“ beschrieben, um eine Abgrenzung zu den anderen Begriffen aufzuzeigen.

2.1 Internet of Things

Der Begriff Internet of Things (IoT, deutsch: Internet der Dinge) wurde erstmalig 1999 bei einer Präsentation bei Procter & Gamble benutzt. Mittels dieses Begriffes wurde die Idee der Verbindung des Internets über Sensoren mit der physischen Welt beschrieben. (Vgl. Ashton 2009)

Die Idee, Sensoren in Dinge einzubauen und deren Informationen aus dem Internet abzurufen wurde allerdings bereits im Jahr 1982 erstmalig an der Carnegie Mellon University umgesetzt. In diesem Fall konnte über eine Internetverbindung festgestellt werden, ob ein mit Sensoren ausgestatteter und somit intelligenter Getränkeautomat befüllt ist und ob die enthaltenen Getränke bereits gekühlt sind. (vlg. Machine 2020)

Die Vision des Internets der Dinge erweitert dieses Konzept insofern, dass von Netzwerken intelligenter Geräte ausgegangen wird, die miteinander selbstständig kommunizieren können. Bei diesen Geräten handelt es sich um kommunikationsfähige Hardware, die mit Sensoren, Aktuatoren oder Prozessoren ausgerüstet ist. Die Spanne reicht dabei von kleinen zur Temperaturüberwachung installierten Geräten in der Industrie bis hin zu Wearables und Smartphones. Im industriellen Umfeld wird der spezifische Begriff „Industrial IoT“ verwendet. Dieser deutet auf spezielle industrielle Anforderungen der Geräte hin. Zudem werden diese Geräte in bestehende industrielle Anlagen verbaut und müssen mit den dort existierenden Schnittstellen kompatibel sein. (Vgl. Sisinni u. a. 2018)

Da eine hohe Diversität bei den IoT-Geräten vorherrscht, sind auch deren Einsatzgebiete vielfältig. Diese inkludieren beispielsweise industrielle Produktion, Logistik, Prozessmanagement, Warentransport, Medizintechnik und die Landwirtschaft. (Vgl. Shah und Yaqoob 2016)

2.2 Cloud Computing

Als Cloud Computing wird sowohl die Bereitstellung von Anwendungen als Service über das Internet als auch die Bereitstellung der darunterliegenden Hardware bezeichnet. Diese ist für gewöhnlich in zentralisierten Datenzentren untergebracht, auf welche der Kunde, der den Service in Anspruch nimmt, im Normalfall keinen physischen Zugriff hat. Diese Infrastruktur wird generell als Cloud (deutsch: Wolke) bezeichnet und es wird prinzipiell zwischen Public Cloud und Private Cloud unterschieden. Wird eine Cloud der Allgemeinheit gegen Entgelt zur Verfügung gestellt, so handelt es sich um eine Public Cloud. Als Private Cloud wiederum werden unternehmensinterne beziehungsweise nicht der allgemeinen Öffentlichkeit zur Verfügung gestellte Rechenzentren bezeichnet. (Vgl. Patidar, Rane und Jain 2012)

Somit ist es möglich, Internetservices auf gemieteten Servern zu betreiben, auf denen dynamisch stets so viel Computing-Ressourcen zur Verfügung gestellt werden, wie gerade benötigt werden. So lassen sich Lastspitzen abfangen und Anschaffungskosten für zusätzliche Hardware können eingespart werden, da 1000Server für eine Stunde nicht teurer sind als ein Server für 1000 Stunden. (Vgl. Armbrust u. a. 2010)

Somit ist Cloud Computing und das damit einhergehende „pay-as-you-go“-Bezahlmodell eine effiziente Alternative zur Verwaltung von eigenen Servern, da der Fokus dadurch auf den Spezifikationen der Anwendungen liegen kann und nicht auf der Hardware.

2.3 Fog Computing

Als Fog (deutsch: Nebel) wird in der Meteorologie eine Wolke, die in Bodenkontakt steht, bezeichnet. So wie Nebel sich näher als Wolken am Boden befindet, so findet Fog Computing näher an den Datenquellen als Cloud Computing statt. Obwohl Cloud Computing als etabliertes Servicemodell gilt, zeigt es dennoch Schwächen. So benötigt der Transfer von Daten in die Cloud und zurück Zeit, die in zeitkritischen Anwendungen nicht zur Verfügung steht.

Auch mit der steigenden Anzahl an IoT-Geräten werden ständig mehr Daten in der Cloud abgespeichert und verarbeitet. Dies benötigt mehr und mehr Bandbreite und gilt als nicht effizient. (Vgl. Yousefpour, Ishigaki und Jue 2017) Insbesondere Anwendungen, die Daten für die Berechnung in hoher zeitlicher Auflösung benötigen, tragen zur Auslastung der Bandbreite bei, da große Datenmengen in kurzer Zeit übertragen werden müssen.

Fog Computing soll derartige Probleme adressieren. Anstatt sämtliche Daten in die Cloud zu senden, wird versucht eine effizientere Lösung zu kreieren, indem Rechenkapazitäten nahe der Sensoren, die Daten generieren, benutzt werden. Fog lässt sich daher als Zwischenstück zwischen Cloud und datengenerierenden Geräten, beispielsweise IoT-Geräte, beschreiben. Ein mögliches Ziel der Verschiebung an Intelligenz

auf diese Zwischenschicht ist beispielsweise eine Verringerung der Latenzzeit zwischen Datengenerierung und -verarbeitung. Somit wird es für manche Anwendungen einfacher, Echtzeitfähigkeit zu garantieren. Durch die Eigenschaft der verteilten Knoten bietet sich eine derartige Plattform zudem für Anwendungen an, bei denen Standortbestimmung wichtig ist oder die prinzipiell geografisch verteilt sind. Diese kommen aus verschiedensten Bereichen, beispielsweise Verkehrsunterstützung, Analytik, Infotainment und Sicherheit. (Vgl. Bonomi, Milito, Zhu u. a. 2012)

Ein möglicher Anwendungsfall ist ein intelligentes Ampelsystem, das die Anwesenheit und Anzahl von Fußgängern und Fahrzeugen, inklusive deren Geschwindigkeit, detektiert. Basierend auf diesen Informationen soll dieses System den Verkehr regeln und zusätzlich Informationen von anderen Ampeln erhalten, um für einen möglichst reibungslosen Ablauf zu sorgen. Datenverarbeitung in Echtzeit ermöglicht dabei die ständige Anpassung an die aktuelle Verkehrslage und bereits verarbeitete Daten können für Langzeitanalysen in der Cloud abgespeichert werden. (Vgl. Bonomi, Milito, Zhu u. a. 2012)

2.4 Edge Computing

Der Begriff Edge Computing wird oft als Synonym für Fog Computing verwendet. In *IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing* (2018) wird allerdings darauf hingewiesen, dass dies fälschlicherweise geschieht. So arbeitet Edge Computing im Gegensatz zu Fog Computing nicht mit der Cloud zusammen, sondern versucht diese zu ersetzen. Außerdem ist Edge Computing auf eine geringe Anzahl von Schichten begrenzt, wohingegen Fog Computing mehrere Hierarchieebenen einnehmen kann. Abschließend befasst sich Edge Computing lediglich mit der Verarbeitung von Daten. Zusätzliche Aspekte, wie die Vernetzung von Geräten, die Speicherung von Daten und eine mögliche Steuerung des Verarbeitungsflusses fallen in die Kategorie Fog Computing.

3 Aktueller Stand der Technik

In diesem Kapitel werden verschiedene aktuelle wissenschaftliche Arbeiten präsentiert, die sich mit dem Thema Fog Computing auseinandersetzen. Zunächst werden Abhandlungen präsentiert, die Einsicht in mögliche Fog Computing-Architekturen bieten. Dadurch sollen verschiedene Realisierungsformen von Architekturen aufgezeigt werden. Zusätzlich werden auch Arbeiten, die Fog Computing-Frameworks realisieren, behandelt. Dadurch sollen unterschiedliche Implementierungsformen sichtbar werden, um zu erkennen, welche Möglichkeiten genutzt werden können, um Daten und Prozesse verfügbar zu machen.

3.1 Fog Computing-Architekturen

Im Vergleich zu Cloud Computing gilt Fog Computing als ein verteiltes Konzept, das besonders zeitkritische Anwendungen erlauben soll. Diese Anwendungen können sehr unterschiedlich ausfallen und benötigen daher entsprechend verschiedene Ansätze. In diesem Abschnitt werden verschiedene aktuelle Ansätze zu Fog Computing-Architekturen diskutiert.

In der allgemeinsten Form ist in der Literatur die Rede von einem IoT-Fog-Cloud-Modell. Dieses ist in Abbildung 3.1 abgebildet.

Der Name indiziert bereits die drei dabei vorkommenden Schichten. Der IoT-Layer beherbergt die Endgeräte und Sensoren, im Fog-Layer existieren die Fog-Knoten, also Geräte, die die Verbindung zur Cloud ermöglichen, in der Cloud-Ebene sind Server von Datenzentren platziert.

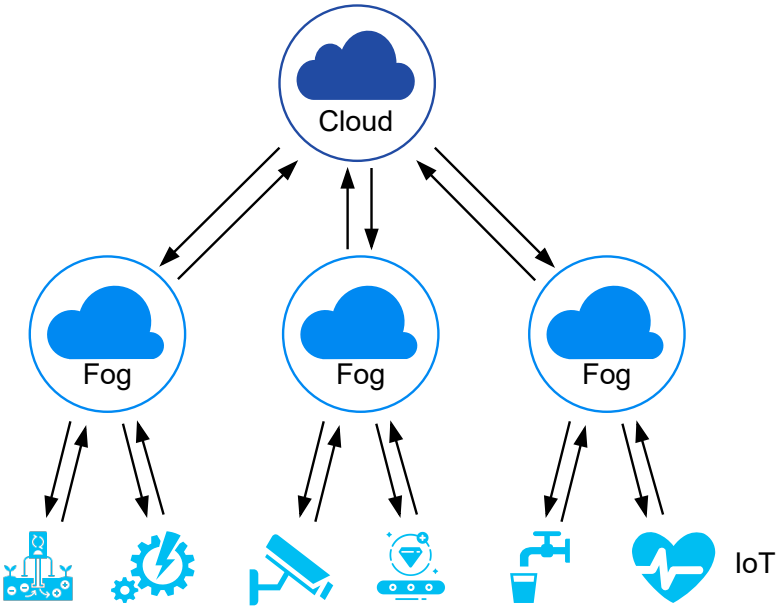


Abbildung 3.1: IoT-Fog-Cloud-Modell
Quelle: eigene Ausarbeitung

Dieses Modell ist bewusst allgemein gehalten, damit sich verschiedene anwendungsspezifische Szenarien beschreiben lassen. So könnten die Schichten beispielsweise in kleinere Domänen unterteilt werden, wobei jede Domäne für eine Anwendung stehen kann. Im IoT-Layer können dies in einer Fabrik verteilte Sensoren sein, die mit den Fog-Geräten kommunizieren. Im Bereich von Smart Home können es die Temperatur- und Luftfeuchtigkeitssensoren des jeweiligen Stockwerkes sein. Die dazugehörenden Fog-Geräte kommunizieren ebenfalls mit ihnen zugeteilten Cloud-Servern. Auch könnte die Fog-Schicht selbst über mehrere Ebenen verfügen. (Vgl. Zou u. a. 2019)

In der Arbeit von Munir, Kansakar und Khan (2017) wird ebenfalls dieser Architekturansatz behandelt. Allerdings wird dabei von einer integrierten Fog-Cloud-IoT-Architektur gesprochen, welche zudem genauer spezifiziert wird. Dabei besteht der Fog aus sogenannten Fog-Knoten, die kleinere Server oder intelligente Router sein können, aber auch Basisstationen des Mobilfunknetzes. Bei dieser Aufteilung ist ein Fog-Knoten für sämtliche IoT-Geräte in dessen drahtlosem Netzwerk zuständig. Als Technologien werden dabei WLAN und Mobilfunk als Beispiele aufgezählt. Die Fog-Geräte stehen dabei miteinander in Kontakt. Sollte sich die Position eines IoT-Gerätes von einer Zelle zu einer anderen verändern, so ändert sich die Zuständigkeit des jeweiligen Fog-Gerätes. In der Cloud treffen sich die Daten der einzelnen Fog-Geräte und bilden ein Ganzes für spätere Datenanalyse und Statistiken der gesamten geografischen Ausprägung des Netzwerkes.

Liu, Fieldsend und Min (2017) beschäftigen sich mit der Architektur der Fog-Ebene.

Speziell das Thema Latenz spielt in ihrer Arbeit eine wichtige Rolle. So wird darauf eingegangen, dass in einer komplexeren Architektur der Fog-Ebene durch zusätzliche Hops nicht notwendigerweise mehr Verarbeitungszeit benötigt wird, als in einer einfach gehaltenen Fog-Ebene. Die Zeit, die für das Passieren von mehreren Geräten benötigt wird, kann durch Geräte mit höherer Rechenleistung kompensiert werden.

Das bereits in Abschnitt 2.3 genannte Beispiel des intelligenten Ampelsystems mittels Fog Computing wird in Minh u. a. (2018) näher behandelt. In deren Architektur befinden sich auf der Fog-Ebene Master- und Slave-Knoten. Ein Slave ist dabei an jeder Kreuzung installiert und erhält sämtliche Daten, um optimierte Grünphasen für die jeweiligen Teilnehmer/innen zu berechnen. Eine Gruppe von Slaves ist dabei stets dem Masterknoten zugeteilt. Dieser erstellt statistische Analysen der ganzen Region und kann basierend auf den Ergebnissen als höhere Instanz die Entscheidungen der Slaves beeinflussen. Das simulierte System ergab, dass mittels eines Fog Computing-Ansatzes die Latenzzeit, im Vergleich zu einem Ansatz mittels Cloud Computing, von 200 ms auf 4 ms reduziert werden kann.

In Dastjerdi u. a. (2016) werden den Geräten auf dem IoT-Layer auch Anwendungen zugeschrieben, die darauf installiert werden können, um deren Funktionalitäten zu erweitern. Geräte in diesem Layer benutzen den darüberliegenden Layer, um miteinander zu kommunizieren. In dieser Arbeit wird diese Schicht nicht als Fog-Ebene, sondern lediglich als Netzwerk bezeichnet. Dennoch bildet diese Schicht die Zwischenebene zur Cloud und ist sowohl für die Kommunikation zwischen den Geräten dieser Ebene als auch für die Kommunikation mit der Cloud zuständig. Der besondere Unterschied dieser Architektur ist die Nennung einer vierten und fünften Ebene. Dies ist in Abbildung 3.2 ersichtlich. Diese weiteren Ebenen liegen über der Cloud-Ebene und stellen die Software für die Ressourcenverwaltung dar. Sie beinhalten als oberste und fünfte Schicht die einzelnen Anwendungen, die mittels Fog Computing im gesamten Netzwerk verteilt werden. Besonders interessant ist hierbei der vierte Layer, der das Ziel hat, den Datenverkehr in der Cloud zu senken und im gleichen Moment auch die Latenzzeit von Anwendungen zu verringern. Dies geschieht durch den Einsatz von verschiedenen Services, durch die die Ressourcen von Fog und Cloud möglichst optimal ausgenutzt werden.

Ein anderer Ansatz für die Auslagerung von Diensten aus der Cloud wird in Szydlo u. a. (2017) beschrieben. In dem in dieser Arbeit vorgestellten Beispiel wird Fog Computing anhand einer zweischichtigen Architektur dargestellt. Diese ist in Abbildung 3.3 abgebildet. Dabei werden Schritte der Datenverarbeitung aus der Cloud direkt auf die Endgeräte ausgelagert. Somit findet das Fog Computing direkt auf IoT-Geräten statt, die in diesem Fall über eine Netzwerkanbindung verfügen. Das Besondere an dieser Arbeit ist, dass das Prinzip des Fog Computings mittels einer Skriptsprache auf kostengünstigen 32-bit Mikrocontrollern umgesetzt wurde, die über stark begrenzten Speicherplatz verfügen.

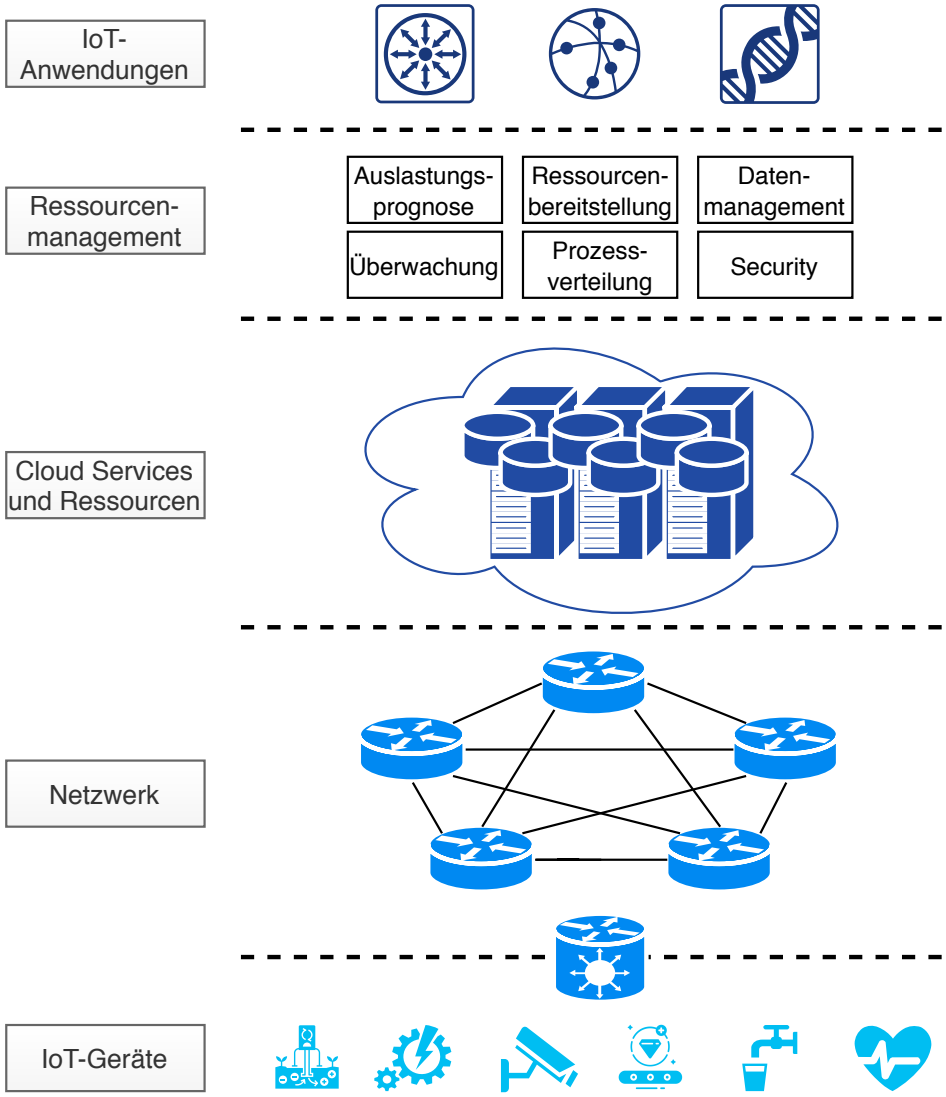


Abbildung 3.2: Erweiterte Fog Computing-Architektur
Quelle: in Anlehnung an Dastjerdi u. a. 2016

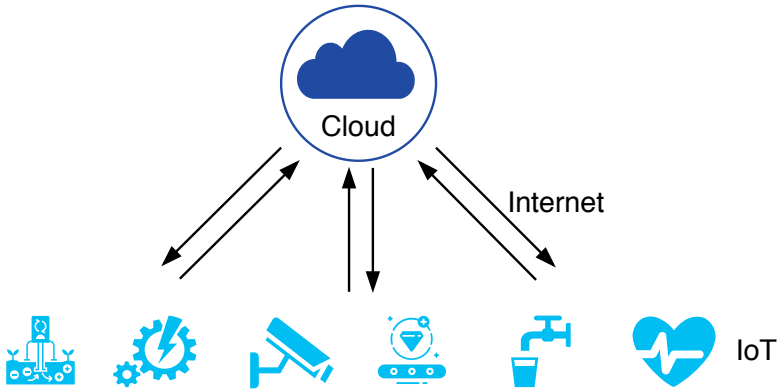


Abbildung 3.3: Zweischichtige Fog Computing-Architektur
Quelle: eigene Ausarbeitung

3.2 Fog Computing-Frameworks

Wie Abschnitt 3.1 Fog Computing-Architekturen zeigt, ist die Entscheidung über die zugrunde liegende Architektur eines Fog Computing-Systems abhängig von dessen Anwendungsfall und kann verschieden realisiert werden. Während beim vorangegangenen Abschnitt der Fokus eher auf dem allgemeinen Aufbau des Systems lag, so liegt der Schwerpunkt dieses Abschnittes bei den darunterliegenden Konzepten zur Realisierung.

Zur Illustrierung der vorgestellten Lösung eines Frameworks zur Verarbeitung von Datenstreams wird in Hießl, Hochreiner und Schulte (2019) ein Umfeld in Form einer Smart Factory benutzt. Das vorgestellte Framework bietet die Möglichkeit, Prozesse auf Fog-Devices zu laden, die Datenstreams von IoT-Sensoren verarbeiten. Diese Instanzen werden als Container, beispielsweise Docker-Container, bereitgestellt. Somit müssen Fog-Geräte, die diese Container benötigen, virtualisierte Ressourcen bereitstellen können. Zudem ist es auch möglich, diese Container in der Cloud einzusetzen, wodurch Prozesse, die mehr Rechenkapazität benötigen als die verfügbaren Fog-Geräte bereitstellen, bedient werden.

Ebenso werden in Al-Rakhami u. a. (2018) Docker-Container in der Zwischenschicht zur Cloud benutzt. Diese Schicht besteht in diesem Paper aus Raspberry Pi 3-Geräten, welche jeweils für mehrere hundert Sensoren zuständig sind. Sie werten deren Messwerte aus und leiten lediglich die Resultate an die Cloud weiter. Somit wird die Anzahl an Datenstreams reduziert, da nur die einzelnen Raspberry Pi´s Ergebnisse an die Cloud schicken und nicht jeder einzelne Sensor. Dies reduziert auch die Menge an zu übertragenden Daten, da lediglich bereits weiterverarbeitete Daten versendet werden.

Im Gegensatz dazu wird in Amjad u. a. (2017) zwischen Taskanfragen unterschieden, die entweder eine geringe Latenz benötigen oder ausschließlich verarbeitet werden

müssen und daher keine expliziten Anforderungen an die Latenz haben. Wird eine Anfrage ohne Latenzanforderungen empfangen, so wird versucht, diese beim Cloud-Provider zu verarbeiten. Sollten dort keine Ressourcen zur Verfügung stehen, so wird eine passende Fog-Ressource gesucht, die diese Anfrage verarbeiten kann. Sollte eine Anfrage mit Latenzanforderungen eintreffen, so wird zunächst überprüft, ob diese mit den verfügbaren Cloud-Ressourcen haltbar ist. Sollte dies nicht möglich sein, so werden Ressourcen im Fog allokiert, welche die Anforderungen erfüllen können. Bei diesem Modell wird zudem davon ausgegangen, dass die Fog-Geräte untereinander Ressourcen teilen, um möglichst viele Anfragen bearbeiten zu können.

In Al-khafajiy u. a. (2018) unterscheiden die Autoren für eine optimale Datenverarbeitung zwischen „Light Requests“ und „Heavy Requests“. Dem medizinischen Umfeld des Papers entsprechend werden Beispiele aus diesem Bereich benutzt. So könne ersteres den Daten eines Herzfrequenzsensors zugeordnet werden und letzteres dem Videostream einer Überwachungskamera entsprechen. Dieser Idee zugrundeliegend werden Anfragen zur Datenverarbeitung von Fog-Geräten je nach Prozessorauslastung entweder akzeptiert oder teilweise oder auch komplett an das nächste Fog-Gerät weitergeleitet. Durch die Unterscheidung der Art des Requests kann somit das passende, gegebenenfalls leistungsstärkere Fog-Gerät zur Verarbeitung ausgewählt werden und die Latenzzeit zwischen Datengenerierung und -verarbeitung noch stärker reduziert werden.

Bei der Betrachtung der verschiedenen Frameworks fällt grundsätzlich auf, dass zwischen zwei verschiedenen Ansätzen unterschieden werden kann. Der erste Ansatz stellt Anwendungen mittels Containervirtualisierungen bereit, wohingegen der zweite andere Möglichkeiten zur Bereitstellung von Anwendungen benutzt. Weitere Unterscheidungen zwischen den Kategorien von Anwendungen bzw. Tasks sind dabei unabhängig von den beschriebenen Ansätzen. Je nach Anwendungsszenario kann eine andere Methode gewählt werden. Dies sollte für das jeweilige Szenario evaluiert werden.

4 Design

In diesem Kapitel werden Funktionalitäten und Anforderungen des zu implementierenden Fog-Frameworks beschrieben. Zusätzlich werden weitere Anforderungen für Komponenten definiert, die mit dem Framework interagieren. Darauf basierend werden Designentscheidungen für die einzelnen Komponenten getroffen.

In Abschnitt 4.1 werden zunächst funktionale und nicht-funktionale Anforderungen an das Fog Computing-Framework definiert und Abgrenzungen der Arbeit genannt. Darauf folgend werden in Abschnitt 4.2 Möglichkeiten für Kommunikationsmechanismen beschrieben. Weiters werden in Abschnitt 4.3 Möglichkeiten zur Prozessverteilung zwischen Geräten erörtert. In Abschnitt 4.4 werden Kriterien definiert, die von Betriebssystemen für ressourcenbeschränkte Systeme erfüllt werden sollten. Daraus schlussfolgernd werden in Abschnitt 4.5 Designentscheidungen für die weitere Implementierung getroffen.

4.1 Fog Computing-Framework

Kapitel 3 ergibt, dass der verwendete Lösungsansatz stets vom System abhängig ist. Wie aus Abschnitt 3.2 ersichtlich, ist die Verschiebung von Prozessen als Container ein oft verwendeter Ansatz. Allerdings ist dies für ressourcenbeschränkte Systeme nicht möglich. Die von Szydlo u. a. (2017) erwähnte Methode der Auslagerung von Prozessen auf IoT-Geräte, die einen Skriptsprachen-Interpreter ausführen, siehe Abschnitt 3.1, scheint realistischer. Allerdings sind die dort erwähnten IoT-Geräte direkt an das Internet angebunden. Von dieser Voraussetzung kann bei vielen Geräten nicht ausgegangen werden.

In diesem Abschnitt werden Anforderungen an das zu implementierende Fog-Framework erhoben. Dabei wird zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden. Zudem werden auch Abgrenzungen festgelegt, die nicht zum Fokus des in dieser Arbeit vorgestellten Fog-Frameworks gehören.

4.1.1 Funktionale Anforderungen

Funktionale Anforderungen beschreiben, welche Funktionen das System erfüllen sollte. Diese werden in diesem Unterabschnitt definiert und sind jeweils mit einem Stichwort und einer dazugehörigen kurzen Beschreibung angeführt:

REQ01 Ausführbarkeit
Das Fog-Framework sollte auf ressourcenbeschränkten Systemen ausführbar sein.

REQ02 Kommunikation
Das Fog-Framework sollte eine Kommunikationsschnittstelle zwischen Fog- und IoT-Ebene bereitstellen.

REQ03 Softwareverteilung
Das Fog-Framework sollte einen Mechanismus bereitstellen, um Software - beispielsweise Funktionen - von Fog-Geräten auf IoT-Geräte zu verteilen.

REQ04 Softwareausführung
Das Fog-Framework sollte Möglichkeiten bieten, die verteilte Software auf den Zielgeräten auszuführen.

REQ05 Ressourcen auslesen
Das Fog-Framework sollte die Möglichkeit bieten, Ressourcen von Zielgeräten auszulesen.

REQ06 Fehlerbehandlung
Das Fog-Framework sollte eine Fehlerbehandlung für Funktionen des Frameworks bereitstellen. Diese Funktionen können beispielsweise für das Herstellen und das Schließen einer Verbindung zwischen zwei Geräten verantwortlich sein.

REQ07 Kompatibilitätsprüfung
Das Fog-Framework sollte vor der Verteilung von Software sicherstellen, dass das Zielgerät mit der zu verteilenden Software kompatibel ist.
Ein IoT-Gerät, das Temperatursensoren ausliest, soll beispielsweise keine Funktionen zur Verfügung gestellt bekommen, die andere spezifische Sensoren benötigen.

REQ08 Enumeration
Das Framework sollte Möglichkeiten zur Verfügung stellen, verbundene Geräte aufzulisten.

REQ09 Statusabfrage
Das Fog-Framework sollte die Möglichkeit bieten, Statusabfragen zu versenden.

4.1.2 Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen sind Beschreibungen von Charakteristiken des Systems und definieren somit nicht, im Gegensatz zu funktionalen Anforderungen, was das System können sollte sondern wie gut es etwas können sollte. Um die Anwendbarkeit des Systems zu gewährleisten werden in diesem Unterabschnitt folgende Aspekte, die vom Framework erfüllt werden sollten, als nicht-funktionale Anforderungen definiert:

REQ10 Plattformunabhängigkeit
Das Framework sollte auf IoT-Geräten, auf denen ein Echtzeitbetriebssystem ausgeführt wird, betrieben werden können und sollte möglichst nicht von speziellen Mechanismen des Betriebssystems abhängig sein. Die Auswahl von Technologien sollte - falls möglich - nicht die Wahl eines speziellen Betriebssystems fixieren.

REQ11 Transportlayerunabhängigkeit
Das Fog-Framework sollte unabhängig von einem speziellen Transportkanal sein.

REQ12 Erweiterbarkeit
Das Fog-Framework sollte mit neuen Funktionalitäten erweitert werden können.

4.1.3 Abgrenzungen

Dieser Unterabschnitt grenzt diese Arbeit ab. Die hier angeführten Punkte müssten in einem kommerziellen oder industriellen Umfeld beachtet werden. Um den Fokus auf anderen Punkten zu setzen, werden sie in dieser Arbeit allerdings bewusst nicht adressiert.

Security
Der Aspekt der Security wird in dieser Arbeit nicht behandelt. Somit liegt das Augenmerk beispielsweise nicht auf der Sicherstellung von Vertraulichkeit oder Integrität.

Cloudanbindung
Da Fog Computing die Datenverarbeitung näher zu Endgeräten, potenziell sogar auf die Endgeräte, bringt, wird die Anbindung an die Cloud in dieser Arbeit in weiterer Folge nicht betrachtet. Somit werden damit verbundene Themen wie Datenspeicherung und Datenvisualisierung nicht behandelt.

4.2 Kommunikationsmechanismen

Der Kommunikationsmechanismus, mit dessen Hilfe Geräte in einem Netzwerk miteinander kommunizieren ist je nach Anwendung unterschiedlich. Im Anwendungsfall dieser Arbeit handelt es sich um die Kommunikation zwischen Embedded-Geräten und um die Verteilung von Prozessen.

Um einen besseren Überblick zu erhalten, werden in diesem Abschnitt zunächst Kommunikationsmechanismen betrachtet und daraufhin Anforderungen definiert.

4.2.1 Übersicht

In diesem Unterabschnitt werden die Kommunikationsmechanismen REST und RPC vorgestellt. Im Anschluss daran werden Kriterien definiert, anhand derer eine Entscheidung für eine Kommunikationsmechanismus-Art getroffen wird.

REST

Representational State Transfer (REST) ist ein Kommunikationsmechanismus für verteilte Systeme. Generell gilt für diesen die Anforderung, dass eine klassische Client-Server-Architektur besteht. Bei REST müssen übertragene Nachrichten die Eigenschaft der Zustandslosigkeit erfüllen. Das bedeutet, dass keinerlei Zustandsinformationen zwischen Nachrichten abgespeichert werden. Nach Ausführung einer Operation vergisst eine Komponente alles über den Aufrufer. Somit muss die übertragene Nachricht selbstbeschreibend sein und alles für die Interpretation Notwendige enthalten. Als Transportprotokoll kommt beinahe ausschließlich das Hypertext Transfer Protocol (HTTP) zum Einsatz.

Vor allem für Webanwendungen wird REST oft benutzt und ist aufgrund seiner Einfachheit weit verbreitet. (Vgl. Steen und Tanenbaum 2017)

RPC

Remote Procedure Call (RPC) bezeichnet eine Methode, Funktionen auf anderen Computern auszuführen. Diese Herangehensweise trifft technisch gesehen auf eine größere Gruppe von Protokollen zu. Somit existieren verschiedene Implementierungen für verschiedene Formate. Client und Server müssen sich auf das zu übertragende Format und zusätzlich auch auf die Darstellung von Datentypen und auf das benutzte Transportprotokoll einigen.

Implementierungen, die diese angesprochenen Punkte behandeln, bieten den Vorteil, dass die Entwicklung von darauf aufbauenden Anwendungen sich als unkompliziert gestaltet. Somit ist RPC eine weit verbreitete Methode, die in vielen verteilten Systemen eingesetzt wird. (Vgl. Steen und Tanenbaum 2017)

4.2.2 Anforderungen

In diesem Unterabschnitt werden Kriterien für die Auswahl des Kommunikationsmechanismus definiert, welche angelehnt an die in Abschnitt 4.1 genannten sind. Diese werden als Anforderungen in folgender Liste genannt:

REQ13 Unabhängigkeit des Kommunikationskanals
Um die Anbindung an das Fog-Framework für ein breites Spektrum an IoT-Geräten zu ermöglichen, sollte der Kommunikationsmechanismus die Möglichkeit bieten, möglichst viele verschiedene Kommunikationskanäle abdecken zu können.

REQ14 Erweiterbarkeit
Die in Unterabschnitt 4.1.2 erwähnte Anforderung REQ12 Erweiterbarkeit soll bereits durch den Aufbau des Kommunikationsmechanismus möglich und unterstützt werden.

4.3 Prozessverteilung

In diesem Abschnitt werden verschiedene Konzepte zur Prozessverteilung präsentiert, die für den Einsatz auf eingebetteten Geräten vorstellbar sind und für das Fog-Framework genutzt werden könnten. Bei einer Implementierung werden diese Konzepte den Kommunikationsmechanismus benutzen, der aus den gelisteten Anforderungen in Abschnitt 4.2 resultiert.

4.3.1 Dynamisches Laden

Eine Möglichkeit zur Verteilung von Prozessen stellt das dynamische Laden dar. Dadurch können während der Laufzeit Module des Systems ersetzt werden. Dies geschieht, indem Dateien im Executable Linkable Format (ELF), einem Format für ausführbare Programme, geladen werden. Diese Dateien können ganze Programme enthalten oder lediglich Programmteile bzw. -module, die mit dem bestehenden Programm interagieren können.

Bei ELF handelt es sich um ein plattformunabhängiges Format. So kann eine ELF-Datei für verschiedene Plattformen erstellt werden, jedoch benötigt eine ELF-Datei Informationen über die Zielplattform und kann nicht Informationen für mehrere Zielplattformen gleichzeitig abspeichern. Zusätzlich sind auch Informationen über Größe und Lage des Zielspeicherbereichs enthalten, sodass lediglich die ELF-Datei übertragen werden muss und keine zusätzlichen Parameter beim Laden benötigt werden. (Vgl. O'Neill 2016)

4.3.2 Skriptsprache

Eine weitere denkbare Möglichkeit ist der Einsatz einer Skriptsprache. Dabei handelt es sich um eine Programmiersprache, deren Code erst bei der Laufzeit in Maschinencode übersetzt wird. Somit kann ein Programm, das in einer Skriptsprache geschrieben wurde, während der Laufzeit geändert werden und muss nicht bei einer Programmänderung wieder neu kompiliert und auf das Gerät geladen werden. In Abschnitt 3.1 Fog

Computing-Architekturen wurde mit Szydlo u. a. (2017) bereits eine Arbeit erwähnt, die einen solchen Ansatz verfolgt.

In weiterer Folge werden in diesem Unterabschnitt drei verschiedene Skriptsprachen vorgestellt, die in Projekten für Mikrocontroller bereits Einsatz fanden.

Lua

Die Skriptsprache Lua hat sich seit ihrer Veröffentlichung 1993 in Anwendungen verschiedenster Bereiche bewährt. Die C-Referenzimplementierung ist in verschiedene Module unterteilt und kompiliert auf allen Plattformen, die einen Standard-C-Compiler haben. (Vgl. *The Programming Language Lua* 2020) Zudem benötigt die Implementierung weniger als 100 kB Read Only Memory (ROM). (Vgl. Ierusalimschy 2007)

In Szydlo u. a. (2017) (erwähnt in Abschnitt 3.1) wird das Projekt embedded Lua (eLua) verwendet. Dieses Projekt bietet eine Lua-Implementierung für Mikrocontroller an, wobei hierbei kein Betriebssystem genutzt wird. (Vgl. *Overview - eluaproject* 2011)

Python

Auch die Programmiersprache Python, die in einem aktuellen Popularitätsranking Platz 3 der derzeit meistgenutzten Programmiersprachen belegt, kann als Skriptsprache bezeichnet werden. (Vgl. TIOBE Software BV 2020) Durch die Popularität von Python gibt es neben der Python-Standard-Bibliothek eine große Anzahl von zur Verfügung stehenden Bibliotheken.

Für den Embedded-Bereich existiert beispielsweise das Projekt „microPython“. Dabei handelt es sich um eine Python-Implementierung, die eine Teilmenge der Python-Standard-Bibliothek abbildet und kompakt genug ist, auf 256 kB ROM Platz zu finden. (Vgl. George 2018) Ähnlich wie bei „eLua“ wird bei diesem Projekt ebenfalls kein Betriebssystem genutzt.

JavaScript

Auch für die Skriptsprache JavaScript existieren Projekte, die diese Sprache auf Embedded-Geräten ausführen. Dafür wird meist die JavaScript-Laufzeitumgebung Node.js auf diesen Geräten installiert. Allerdings werden dafür bis zu 50 MB an Speicherplatz benötigt. Projekte, die diese Anforderungen reduzieren, erreichen eine Ausführbarkeit auf Geräten, die 16 MB an Speicherplatz bereitstellen. (Vgl. Mulder und Breseman 2016) Aufgrund dieser hohen Speicheranforderungen ist eine Umsetzung für viele Mikrocontroller nur sehr eingeschränkt möglich.

4.4 Betriebssystem

Um in weiterer Folge ein Betriebssystem für die Implementierung eines Fog-Frameworks auszuwählen, werden in diesem Abschnitt Kriterien für dieses gelistet:

- Bei dem Betriebssystem soll es sich um ein Echtzeitbetriebssystem (englisch real-time operating system, RTOS) handeln, um ein deterministisches Zeitverhalten zu erreichen.
- Bei dem Betriebssystem soll es sich um ein Open-Source-Projekt handeln.
- An dem Betriebssystem soll aktiv gearbeitet werden.
- Das Betriebssystem soll eine beträchtliche Anzahl an verschiedenen Baugruppen bzw. Boards unterstützen und dadurch industrieerprobt sein.
- Der Entwicklungsstatus des Betriebssystems soll bereits einen fortgeschrittenen Status erreicht haben, sodass auf mehrere verschiedene Vernetzungsmöglichkeiten zurückgegriffen werden kann.
- Ein Mechanismus zur Prozessverteilung soll für das Betriebssystem verfügbar sein.

4.5 Schlussfolgerungen

In diesem Abschnitt werden Schlussfolgerungen zur Wahl der Programmierschnittstelle, des Konzeptes zur Prozessverteilung und zu dem zu benutzenden Betriebssystem getroffen. Diese basieren auf den zuvor beschriebenen Anforderungen und existierenden Technologien.

4.5.1 Prozessverteilung

Da das zu wählende Konzept zur Prozessverteilung von dem noch zu wählenden Betriebssystem unterstützt werden muss, werden in diesem Unterabschnitt zunächst verschiedene Betriebssysteme betrachtet, die die Mehrzahl der Kriterien aus Abschnitt 4.4 erfüllen.
Im Anschluss daran wird eines der in Abschnitt 4.3 genannten Konzepte gewählt.

Contiki OS

Contiki OS ist ein open-source-Betriebssystem, welches einen Mechanismus für dynamisches Laden besitzt. Dieser wird hierbei „Contiki ELF loader“ genannt und besitzt die Möglichkeit, eine ELF-Datei während des laufenden Betriebs auf einem Board zu laden und somit die Funktionen des Programms während der Laufzeit zu verändern. (Vgl. *Contiki 2.6: The Contiki ELF loader* 2012)

Allerdings existiert seit 2017 ein Projekt namens Contiki-NG (Contiki Next Generation), welches als Nachfolger von Contiki OS bezeichnet werden kann. Dadurch wird lediglich an diesem Nachfolger aktiv gearbeitet, der die Funktionalität des „Contiki ELF loaders“ nicht besitzt. (Vgl. *Contiki NG Github Repository* 2020)

Mbed OS

Mbed OS ist ein frei verfügbares RTOS, das von Arm Limited entwickelt wird. (Vgl. Arm Limited 2020a) Eine Besonderheit dieses Betriebssystems ist die dazugehörige Software „Pelion Device Management“, mit der es möglich ist über Fernzugriff die Firmware von Mbed OS-Geräten upzudaten. Dafür wird ein direkter Internet Protocol (IP-)basierter Netzwerkzugriff benötigt oder ein sogenanntes „Edge production device“, das spezielle Hardware-Anforderungen erfüllen muss. (Vgl. Arm Limited 2020b)

Zusätzlich existieren Foreneinträge von mehreren Benutzern des Betriebssystems, die von einer erfolgreichen Implementierung der Skriptsprache Lua in ihren Applikationen berichten. (Vgl. East und Tilden 2014) Somit sind für Mbed OS mehrere Möglichkeiten zur Prozessverteilung möglich.

Zephyr

Bei Zephyr handelt es sich um ein von Unternehmen wie Intel Corporation und NXP Semiconductors unterstütztes RTOS-Projekt. (Vgl. Zephyr Project 2020b) Die Möglichkeit zum dynamischen Laden von ELF-Dateien wurde 2018 erstmals in den Release-Plan des Projektes aufgenommen. Im März 2020 wurde dieses Feature allerdings wieder aus dem Release-Plan entfernt. Somit bietet dieses Projekt aktuell keine derartige Möglichkeit. (Vgl. Zephyr Project 2020a)

Bezüglich der Implementierung einer Skriptsprache auf Zephyr findet sich eine Quelle vom Embedded Software Engineering Kongress 2019. In einer Präsentation listen die Referenten die Kombination von Lua mit Zephyr. (Vgl. Becker und Kästner 2019)

RIOT

Bei RIOT handelt es sich um ein RTOS, das von drei deutschen Universitäten stark unterstützt wird. (Vgl. FU Berlin 2020) Es stellt im Gegensatz zu den anderen genannten Betriebssystemen einen Lua-Interpreter bereit. Zusätzlich werden auch Funktionen bereitgestellt, die das Laden von selbstgeschriebenen Lua-Skripten und C-Bibliotheken, die mit dem Lua-Interpreter interagieren sollen, erleichtern. (Vgl. Carrano 2020)

Folgerungen

Das Konzept des dynamischen Ladens ist bei den betrachteten Betriebssystemen lediglich für Contiki OS und Mbed OS verfügbar. Da mit Contiki-NG bereits ein Nachfolgeprojekt von Contiki OS existiert und somit am ursprünglichen Betriebssystem nicht mehr aktiv gearbeitet wird, wird entschieden, dass Contiki OS nur begrenzt für Neuentwicklungen geeignet ist und somit in dieser Arbeit nicht weiter eingesetzt wird.

Für Mbed OS ist das Konzept des dynamischen Ladens mittels der „Pelion Device Management“-Software möglich. Allerdings ist dies von IP-basiertem Netzwerkzugriff oder von spezieller Hardware abhängig.

Da sowohl für Mbed OS als auch für Zephyr und RIOT Quellen zur Implementierung eines Skriptsprachen-Interpreters existieren, wird dieser Ansatz zur Prozessverteilung weiterverfolgt. Durch den dadurch gewählten Ansatz, die Problemlösung mittels einer Skriptsprache zu forcieren, ist keine Einschränkung auf ein Betriebssystem notwendig und eine Plattformunabhängigkeit sollte weiterhin gegeben sein.

Für die Wahl einer Skriptsprache gilt es festzuhalten, dass bei den genannten Betriebssystemen stets Lua als Skriptsprache erwähnt wird. Daher - und aufgrund des geringen Speicherverbrauchs - wird auf weitere Vergleiche zwischen Skriptsprachen verzichtet und Lua wird als zu benutzende Sprache festgelegt.

4.5.2 Kommunikationsmechanismen

Die in Abschnitt 4.2 präsentierten Kommunikationsmechanismen REST und RPC erfüllen beide durch deren charakteristische Simplizität die in Unterabschnitt 4.2.2 erwähnte Anforderung „REQ14 Erweiterbarkeit“.

Da REST sich auf HTTP als Transportprotokoll stützt, benötigen Geräte, die dadurch erreichbar sein sollen, einen direkten IP-basierten Netzwerkzugriff. Für RPC existiert eine derartige Beschränkung nicht. Aufgrund dieses Aspektes kann RPC die Anforderung „REQ13 Unabhängigkeit des Kommunikationskanals“ besser als REST erfüllen, da mehrere verschiedene Kommunikationskanäle abgedeckt werden können.

Für die Erfüllung von „REQ13 Unabhängigkeit des Kommunikationskanals“ und für mögliche Implementierungen mit verschiedenen Kommunikationskanälen wird daher RPC als Kommunikationsmechanismus ausgewählt.

4.5.3 Betriebssystem

Von denen in Unterabschnitt 4.5.1 präsentierten Betriebssystemen existieren zu Mbed OS, Zephyr und RIOT Quellen zur Implementierung eines Lua-Interpreters. Für RIOT existiert als einziges dieser Betriebssysteme ein bereits verwendbarer Lua-Interpreter. Daher wird entschieden, RIOT für weitere Entwicklungen zu verwenden.

Um die in Unterabschnitt 4.1.2 gelistete Forderung nach Plattformunabhängigkeit für das Fog-Framework zu erfüllen, wird zudem entschieden, dass das Fog-Framework auf einem weiteren RTOS ausführbar sein soll. Da Zephyr ein fortgeschrittenes Projekt ist, das eine beträchtliche Anzahl an Boards und Schnittstellen unterstützt, wird dieses Betriebssystem als weitere Plattform ausgewählt, um das Fog-Framework zu testen.

5 Implementierung

In diesem Kapitel wird aufbauend auf den in Kapitel 4 getroffenen Designentscheidungen die Implementierung des Fog Computing-Frameworks beschrieben. Dabei werden zunächst in Abschnitt 5.1 die einzelnen Komponenten beschrieben, um einen Überblick über das System zu erhalten. In Abschnitt 5.2 wird die für das Betriebssystem RIOT spezifische Implementierung erläutert. In dem darauffolgenden Abschnitt 5.3 werden die spezifischen Anpassungen für das Betriebssystem Zephyr erklärt. Dabei werden auch die notwendigen Schritte erläutert, um das Framework auf Embedded Geräten auszuführen. Abschließend werden in Abschnitt 5.4 die betriebssystemunabhängigen Features bzw. Services des Fog Computing-Frameworks erläutert.

5.1 Überblick

In diesem Abschnitt wird ein Überblick über das implementierte Fog Computing-Framework präsentiert. Die einzelnen Komponenten des Fog Computing-Frameworks sind in Abbildung 5.1 illustriert und werden in Unterabschnitt 5.1.1 beschrieben. Zusätzlich sind in dieser Übersicht auch Verbindungen zu Komponenten abgebildet, die mit den Framework-Komponenten interagieren. Diese werden in Unterabschnitt 5.1.2 erläutert. Abschließend werden der Aufbau und die Besonderheiten einer Beispielanwendung in Unterabschnitt 5.1.3 erklärt.

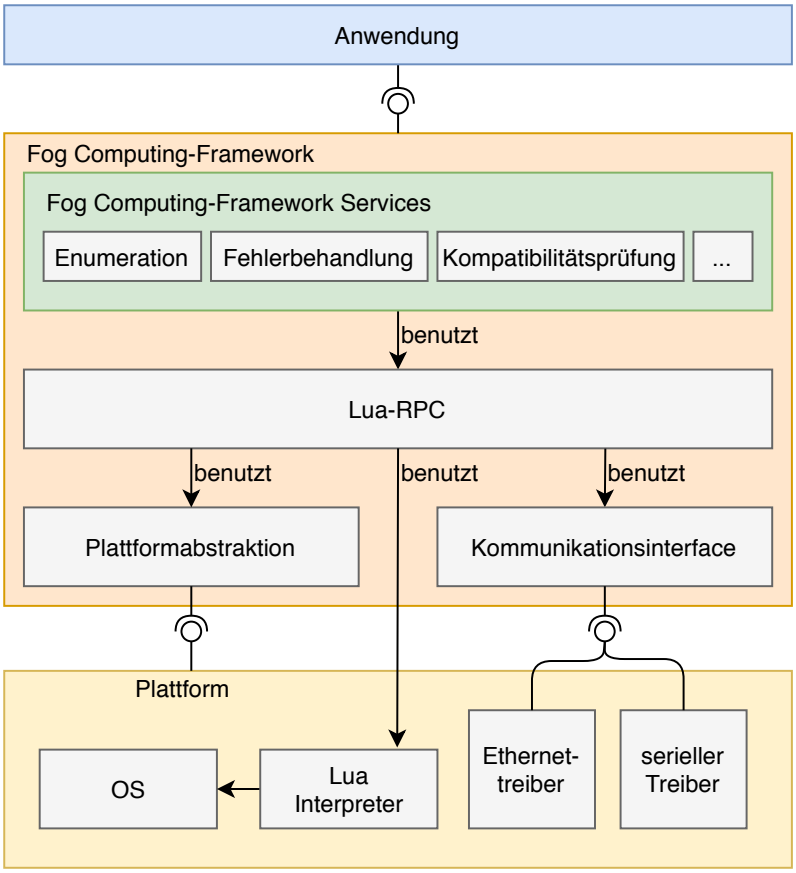


Abbildung 5.1: Übersicht Fog-Framework
Quelle: eigene Ausarbeitung

5.1.1 Komponenten des Fog Computing-Frameworks

In Abbildung 5.1 sind die Komponenten des Fog Computing-Frameworks und die Komponenten, die damit interagieren, abgebildet.

Basierend auf den in Abschnitt 4.5 getroffenen Entscheidungen wird für das Fog-Framework auf Lua als Skriptsprache und auf RPC als Methode gesetzt. Zusätzlich wird auf spezielle Features, beispielsweise eine Unterscheidung zwischen Light- und Heavy-Requests, verzichtet. Aufgrund von gewährleisteter Separierung zwischen dem RPC-Mechanismus und dem Transportmechanismus wird das bereits existierende Projekt „Lua-RPC“ benutzt.

Bei „Lua-RPC“ handelt es sich um eine Lua-Bibliothek, die Teil des eLua-Projektes ist und es ermöglicht ein RPC-Protokoll zur Kommunikation zwischen verschiedenen Geräten via Transmission Control Protocol (TCP) oder via serielle Verbindung zu

benutzen. (Vgl. Snyder 2014) Für die jeweiligen Transportmechanismen werden verschiedene Header-Dateien benutzt, die wiederum von der darunterliegenden Hardware und dem darauf betriebenen OS abhängig sind. Dies wird in Abbildung 5.1 durch das Kommunikationsinterface signalisiert, welches als Schnittstelle zu verschiedenen Treibern fungiert. Zudem ermöglicht das Design der Bibliothek die Erweiterbarkeit für weitere Transportmechanismen.

Da für das Betriebssystem RIOT bereits ein Lua-Interpreter existiert und dafür auch Funktionen existieren, die dessen Benutzung erleichtern sollen, weicht die Integration von „Lua-RPC“ für dieses Betriebssystem mit der von anderen ab. Somit unterscheidet sich das Framework von Betriebssystem zu Betriebssystem und es sind Anpassungen für die verschiedenen Betriebssysteme notwendig. Das wird durch die Plattformabstraktion symbolisiert.

Zusätzlich wurden in Unterabschnitt 4.1.1 Anforderungen an das Fog Computing-Framework definiert. Während einige dieser Punkte bereits von „Lua-RPC“ bereitgestellt werden, werden andere als zusätzliche Services des Frameworks implementiert. Zu letzterem gehören die Punkte Fehlerbehandlung, Kompatibilitätsprüfung und Enumeration. Sie beinhalten aber auch weitere wie beispielsweise die Statusabfrage, die in Abschnitt 5.4 detaillierter vorgestellt werden.

5.1.2 Weitere Komponenten

Komponenten, die nicht zum Framework gehören allerdings damit interagieren, werden in diesem Unterabschnitt vorgestellt.

Dazu gehört auf der einen Seite die Anwendung, die das Framework nutzt. Je nach Ausführungsplattform kann diese unterschiedliche Gestalt annehmen. So kann es sich dabei auf einer Desktop-Linux-Distribution um ein alleinstehendes Lua-Skript handeln, das die Funktionalitäten des Frameworks nutzt. Allerdings könnte dieses Lua-Skript auch auf Funktionen anderer Lua-Skripte zugreifen oder als C-Anwendung in Erscheinung treten, die die C-Programmierschnittstelle von Lua benutzt und auf Sensor-Bibliotheken und -Treiber zugreift. Dabei besteht die Möglichkeit, Lua-Code der C-Anwendung bereitzustellen. Das kann bei Vorhandensein eines Dateisystems mittels einer Lua-Datei geschehen oder andernfalls besteht die Möglichkeit, diesen Lua-Code direkt als String in eine C-Datei einzufügen.

Darüber hinaus können mittels der in Lua vorhandenen C-Programmierschnittstelle C-Funktionen in Lua zur Verfügung gestellt werden. So ist es möglich, dass in Lua Funktionen von C-Bibliotheken ausgeführt werden können. Diese Bibliotheken ermöglichen somit die Anbindung von Sensoren.

Auf der anderen Seite ist die Plattform, auf welcher die Anwendung ausgeführt wird, die Komponente, die mit dem Framework interagiert. Diese unterscheidet sich besonders durch das ausgeführte Betriebssystem.

Der Lua-Interpreter ist aufgrund seiner Abhängigkeiten als Komponente der Plattform anzuführen und ist als solche in Abbildung 5.1 abgebildet. Dieser ist nicht Bestandteil des Frameworks, wird aber für die Ausführung benötigt und ist abhängig von der Ausführungsplattform. So können möglicherweise aufgrund von fehlender Implementierung oder weil weniger Speicherplatz benötigt werden soll, auf manchen Betriebssystemen nicht alle Lua-Standardbibliotheken geladen werden. Auch der Lua-Interpreter unter RIOT hat Anpassungen erfahren und unterscheidet sich daher von der C-Referenzimplementierung.

Da das Kommunikationsinterface des Fog Computing-Frameworks von den jeweiligen Bibliotheken der Ausführungsplattform abhängig ist, sind auch diese in Abbildung 5.1 angeführt und werden dort als Treiber bezeichnet. So unterscheiden sich Bibliotheken, die für eine Kommunikation via TCP/IP-Sockets auf einer Desktop-Linux-Distribution benötigt werden, zu denen auf dem Echtzeitbetriebssystem Zephyr und es ist somit notwendig - je nach Plattform - Anpassungen zu tätigen.

5.1.3 Aufbau Beispielanwendung

In diesem Unterabschnitt wird der Aufbau einer Anwendung erläutert, die den RPC-Mechanismus für Funktionsaufrufe auf anderen Geräten nutzt. In Abbildung 5.2 sind die Unterschiede zwischen einer Anwendung, die diesen Mechanismus nutzt, und einer Anwendung, die darauf verzichtet, illustriert. Letztere wird in diesem Unterabschnitt als Ausgangsszenario bezeichnet.

Beim Ausgangsszenario erhält das System Daten von Sensoren via den entsprechenden Schnittstellen-Treibern. Diese werden anschließend an das Fog-Gerät weitergeleitet und dort (vor-)verarbeitet. Diese (vor-)verarbeiteten Daten können daraufhin zur weiteren Verarbeitung und Speicherung in die Cloud geschickt werden.

Im Gegensatz dazu ist in der rechten Spalte von Abbildung 5.2 ein System abgebildet, das via RPC Funktionen auf das IoT-Gerät verteilt, was auch dem Verhalten der in dieser Arbeit genutzten Lua-RPC-Bibliothek entspricht. Diese werden auf dem Fog-Gerät aufgerufen, aber auf dem IoT-Gerät ausgeführt. Dadurch werden lediglich (vor-)verarbeitete Daten vom IoT-Gerät versendet.

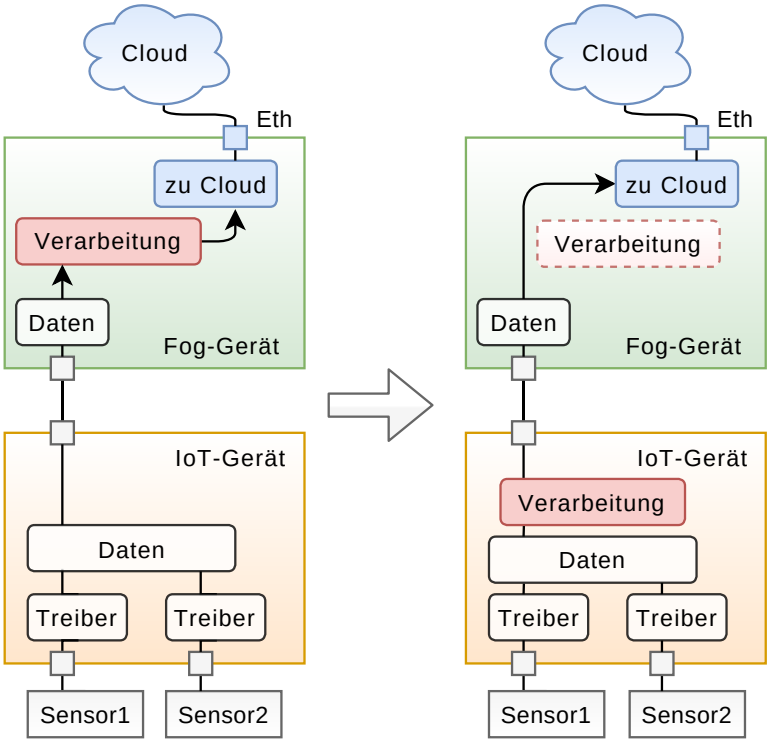


Abbildung 5.2: Prozessverlagerung mit RPC
Quelle: eigene Ausarbeitung

Im Vergleich dazu ist in Abbildung 5.3 ein konkreteres Beispiel abgebildet, das Lua-RPC als Mechanismus zur Verteilung von Funktionen nutzt. Zusätzlich ist auch der Übergang von einer C-Anwendung zu einer Lua-RPC-Anwendung dargestellt.

Der Hauptunterschied beider Anwendungen liegt im möglichen Zugriff auf C-Ressourcen aus einer Lua-Anwendung. Dies wird durch einen Lua-Interpreter sowie eine Funktion, einen sogenannten Lua-Runner, möglich. Letztere bettet die Funktionalitäten des Interpreters in ein C-Programm ein, wobei zusätzlich Komponenten benötigt werden, die die Funktionen der Schnittstellen bzw. Sensortreiber in Lua bereitstellen. Die Lua-RPC-Funktionalitäten können somit in einer Lua-Anwendung verwendet werden und bieten die eigenen Komponenten zur Kommunikation an.

Zur erfolgreichen Kommunikation via Lua-RPC sind diese Komponenten bei beiden Kommunikationspartnern notwendig. Im konkreten Beispiel, das in Abbildung 5.3 zu sehen ist, wird dies zwischen einem IoT- und von einem Fog-Gerät umgesetzt. Hierbei besitzt das Fog-Gerät die Möglichkeit Funktionen, die das IoT-Gerät nicht kennt, zu erstellen und diese auf das IoT-Gerät zu verschieben, um sie im Anschluss über den Lua-RPC-Kommunikationskanal aufzurufen. Dies wird durch Abbildung des Blocks „neuer Prozess“ symbolisiert.

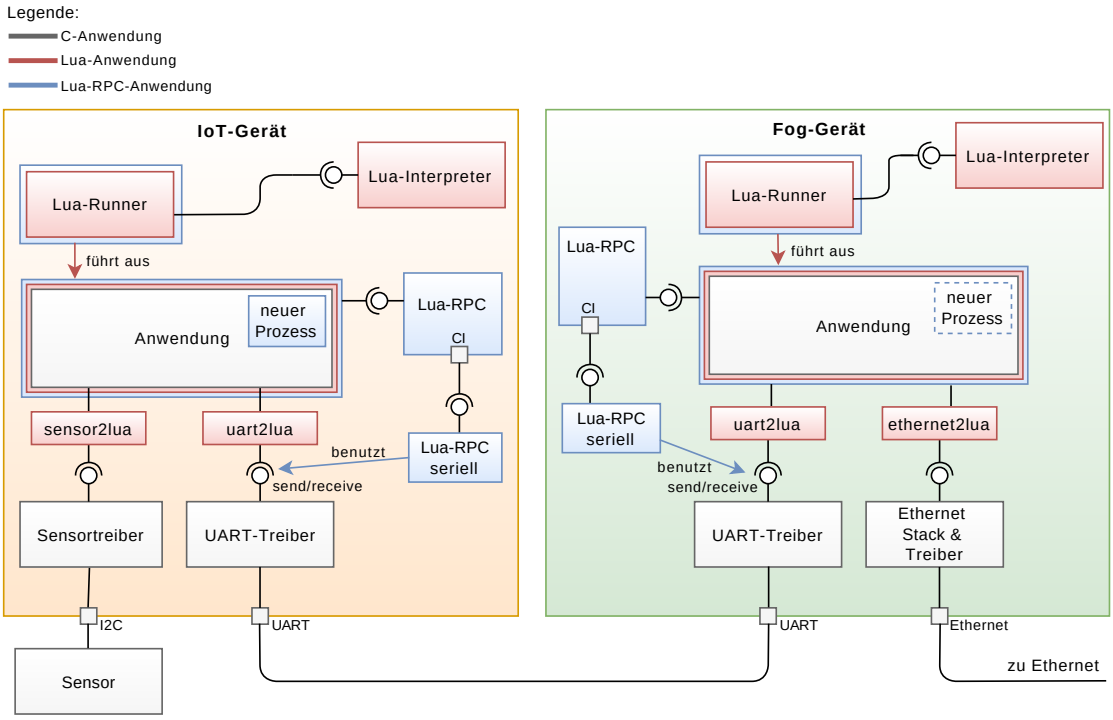


Abbildung 5.3: Beispiel für Transition zu Lua-RPC-Anwendung
Quelle: eigene Ausarbeitung

5.2 RIOT

In diesem Abschnitt werden Implementierungen auf dem in Unterabschnitt 4.5.1 beschriebenen Betriebssystem RIOT beschrieben. In Unterabschnitt 5.2.1 werden zunächst die Möglichkeiten des RIOT-spezifischen Lua-Interpreters vorgestellt. In Unterabschnitt 5.2.2 wird die Einbindung von Lua-RPC in RIOT mit TCP/IP-Sockets beschrieben, bevor in Unterabschnitt 5.2.3 eine Implementierung mit serieller Verbindung zur Kommunikation erklärt wird. Anschließend wird in Unterabschnitt 5.2.4 ein Beispiel zur Einbindung von weiteren C-Bibliotheken in eine Lua-Anwendung in RIOT präsentiert. Schließlich werden in Unterabschnitt 5.2.5 zusätzliche Erweiterungen zu Lua-RPC dargestellt.

5.2.1 Übersicht

Das Betriebssystem RIOT bietet für Linux eine native Portierung an. Somit können Anwendungen auf demselben Linux-PC ausgeführt werden, auf welchem diese auch geschrieben wurden. Dabei werden die architekturspezifischen Header-Dateien des Host-Systems benutzt und nicht die Kommunikationsschnittstellen, die durch RIOT

bereitgestellt werden. Zusammen mit dem von RIOT bereitgestellten Lua-Interpreter bietet dieses RTOS somit Werkzeuge an, um die Funktionalitäten von Lua-RPC zu benutzen. In diesem Unterabschnitt wird mittels einer adaptierten Version der RIOT-Anwendung „lua_demo“ von Carrano (2018) gezeigt, wie diese Komponenten miteinander verbunden werden. Ein Ausschnitt dieser adaptieren Version ist in Quellcode 5.1 sichtbar.

Der Lua-Interpreter von RIOT benutzt einen eigenen Speicherbereich, der im Vorhinein reserviert werden muss. Dies wird in Zeile 3 von Quellcode 5.1 gemacht. In diesem Beispiel werden 40kB an Random-Access Memory (RAM) zur Verfügung gestellt. Laut Carrano (2020) benötigt eine typische Anwendung etwa 12kB RAM. Der notwendige Heap-Speicher kann durch das Laden von lediglich ausgewählten Lua-Bibliotheken bei Bedarf reduziert werden, was in Zeile 4 von Quellcode 5.1 ersichtlich ist. Zudem beinhaltet der Interpreter Mechanismen zum Laden von sowohl Lua-Dateien als auch C-Erweiterungen. Dazu werden deren Namen in einem Array gespeichert, um sie in weiterer Folge der Anwendung bekanntzugeben. Die dafür benötigten Schritte sind in den Zeilen 8 bis 18 von Quellcode 5.1 dargestellt. In diesem Beispiel wird das Lua-Script „server.lua“ geladen sowie die C-Erweiterung „rpc“, bei der es sich um Lua-RPC handelt.

```
1  #include "server.lua.h"
2
3  #define MAIN_LUA_MEM_SIZE (40000)
4  #define BARE_MINIMUM_MODS (LUAR_LOAD_BASE | LUAR_LOAD_IO |
       LUAR_LOAD_PACKAGE | LUAR_LOAD_MATH)
5
6  static char lua_memory[MAIN_LUA_MEM_SIZE] __attribute__ ((aligned(
       __BIGGEST_ALIGNMENT__)));
7
8  extern int luaopen_rpc(lua_State *L);
9  const struct lua_riot_builtin_lua _lua_riot_builtin_lua_table[] = {
10     { "server", server_lua, sizeof(server_lua) }
11 };
12 const struct lua_riot_builtin_c _lua_riot_builtin_c_table[] = {
13     { "rpc", luaopen_rpc}
14 };
15 const struct lua_riot_builtin_lua *const lua_riot_builtin_lua_table
       = _lua_riot_builtin_lua_table;
16 const struct lua_riot_builtin_c *const lua_riot_builtin_c_table =
       _lua_riot_builtin_c_table;
17 const size_t lua_riot_builtin_lua_table_len = 1;
18 const size_t lua_riot_builtin_c_table_len = 1;
```

Quellcode 5.1: RIOT Lua-Beispiel Initialisierungen

Eine zuvor in einem Array abgespeicherte Lua-Datei wird in RIOT mittels der Funktion „lua_riot_do_module“ geladen. In Abbildung 5.3 wurde eine Funktion, die das ermöglicht, als Lua-Runner bezeichnet.

In dieser Lua-Datei kann anschließend auf die Funktionen der in diesem Beispiel

geladenen C-Erweiterung „rpc“ zugegriffen werden, wie in Quellcode 5.2 zusätzlich gezeigt.

```
int main(void)
{
    while (1) {
        int status, value;
        status = lua_riot_do_module("server", lua_memory,
            MAIN_LUA_MEM_SIZE, BARE_MINIMUM_MODS, &value);
        printf("Exited. status: %s, return code %d\n",
            lua_riot_strerror(status), value);
    }
    return 0;
}
```

Quellcode 5.2: RIOT Lua-Beispiel main-Funktion

5.2.2 TCP/IP-Verbindung

In diesem Unterabschnitt wird die Implementierung von Lua-RPC auf RIOT mittels Portable Operating System Interface (POSIX-)kompatiblen Sockets für TCP/IP als Kommunikationsschnittstelle erläutert.

Lua-RPC kann generell für verschiedene Kommunikationsschnittstellen gebaut werden. Für den Einsatz auf einem RTOS muss jedoch bekannt sein, welche Kommunikationsschnittstellen benutzt werden sollen, um die Software für die richtige Konfiguration zu bauen, was auf einer Linux-Desktopumgebung mittels Anpassung des Kommandos geschieht, das den Build-Prozess startet.

Für das folgende Beispiel wurde dies für TCP/IP als Kommunikationsschnittstelle gemacht. Quellcode 5.3 zeigt den Inhalt der in Quellcode 5.2 aufgerufenen Lua-Datei „server.lua“. Dieser Codeausschnitt zeigt die Funktionalitäten der Server-Komponente einer Client-Server-Verbindung von Lua-RPC. Dabei werden zunächst in Zeile 1 die Lua-RPC-Funktionalitäten eingebunden und im Anschluss eine TCP/IP-Socket-Verbindung an Port 12346 geöffnet und in einer Endlosschleife auf eingehende Pakete abgefragt.

```
1  local rpc = require"rpc"
2
3  handle = rpc.listen (12346);
4  while 1 do
5      if rpc.peek (handle) then
6          rpc.dispatch (handle)
7      else
8          doOtherThings();
9      end
10 end
```

Quellcode 5.3: RIOT Lua-RPC-Beispiel Sockets server.lua

Das Pendant dazu, die Client-Komponente, kann ebenfalls in diesem Stil erstellt werden. Quellcode 5.4 zeigt eine einfache Lua-Datei, die die Lua-RPC-Funktionalitäten benutzt. Damit wird eine Verbindung mit dem lokalen Server an Port 12346 hergestellt und es wird diesem die Funktion „squareval“ bekannt gegeben. Zusätzlich wird diese Funktion auf dem Server via RPC aufgerufen und das Ergebnis wird dort ausgegeben. In der letzten Zeile von Quellcode 5.4 wird die Verbindung zum Server wieder getrennt.

```
1  local rpc = require("rpc")
2
3  local  slave, err = rpc.connect ("localhost",12346);
4
5  function squareval(x) return x*x end
6
7  slave.squareval = squareval;
8  slave.print(slave.squareval(99));
9
10 rpc.close (slave)
```

Quellcode 5.4: RIOT Lua-RPC-Beispiel Sockets client.lua

5.2.3 Serielle Verbindung

Das in Unterabschnitt 5.2.2 benutzte Beispiel nutzt TCP/IP-Sockets zur Kommunikation. Dies ist allerdings auch über eine serielle Verbindung möglich. Dabei muss das Makro für eine TCP/IP-Verbindung mit dem für eine serielle Verbindung ersetzt werden. In weiterer Folge muss im Quellcode statt eines TCP-Ports ein serieller Port angegeben werden. Das wird für den Server in Quellcode 5.5 und für den Client in Quellcode 5.6 demonstriert.

```
1  local rpc = require "rpc"
2
3  handle = rpc.listen ("/dev/pts/3");
4  while 1 do
5      if rpc.peek (handle) then
6          rpc.dispatch (handle)
7      else
8          doOtherThings();
9      end
10 end
```

Quellcode 5.5: Lua-RPC-Beispiel serielle Verbindung Server

```
1  local rpc = require("rpc")
2
3  local slave, err = rpc.connect ("/dev/pts/4");
```

Quellcode 5.6: Lua-RPC-Beispiel serielle Verbindung Client

5.2.4 Einbindung von C-Bibliotheken in Lua

In den Unterabschnitten 5.2.1 bis 5.2.3 wurde lediglich gezeigt, wie der grundsätzliche Aufbau einer Anwendung mit Lua-RPC unter RIOT ist. In diesem Unterabschnitt wird zusätzlich anhand eines Beispiels gezeigt, wie C-Bibliotheken in Lua eingebunden werden, um in weiterer Folge dadurch auf in C initialisierte Komponenten zugreifen zu können. Die hier gezeigten Quellcodes orientieren sich an der Implementierung von Carrano (2018).

Abbildung 5.4 zeigt den Aufbau des in diesem Unterabschnitt vorgestellten Beispiels, bei dem der Server zusätzlich zur Lua-RPC-Komponente und einer Lua-Datei eine weitere C-Bibliothek als Komponente benutzt.

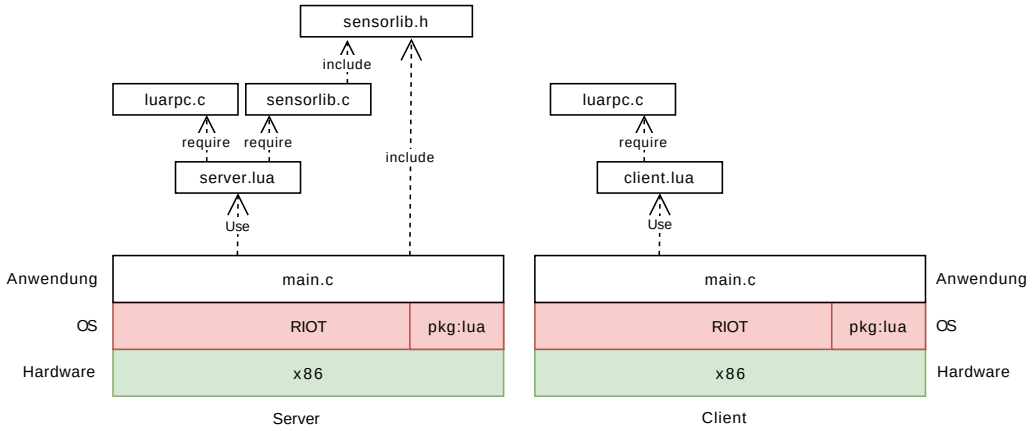


Abbildung 5.4: RIOT Lua-RPC Beispiel mit C-Bibliothek
Quelle: eigene Ausarbeitung

Mit dieser C-Bibliothek wird das Lesen und Schreiben von Sensorwerten simuliert. In diesem Beispiel werden sämtliche Sensoren in einer einfach verketteten Liste abgespeichert. Jeder Knoten dieser Liste entspricht einem Sensor und besitzt einen Eintrag für einen Treiber, der wiederum aus Funktionspointern besteht. Hierbei existieren Einträge für eine „read“- und eine „write“-Funktion. Die in diesem Absatz beschriebenen Verhältnisse sind in der in Quellcode 5.7 gezeigten Header-Datei definiert.

```
typedef int(*sensor_read_t)(const void *dev);
typedef int(*sensor_write_t)(const void *dev, int *data);

typedef struct {
    sensor_read_t read;
    sensor_write_t write;
} sensor_driver_t;

typedef struct sensor {
    struct sensor *next;
    int *value;
    const char *name;
    sensor_driver_t const *driver;
} sensor_t;

sensor_t *sensor_find_name(const char *name);
int sensor_read(sensor_t *dev);
int sensor_write(sensor_t *dev, int data);
int sensor_add(sensor_t *dev);
```

Quellcode 5.7: sensorlib.h

Damit in Lua auf diese Funktionen zugegriffen werden kann, müssen in der C-Datei der Bibliothek die Namen der Lua-Funktionen registriert werden, die diese aufrufen. Dies wird in Quellcode 5.8 durch ein Array dargestellt, wobei der erste Eintrag eines Array-Elements dem Namen einer, in diesem Fall gleichnamigen, Lua-Funktion entspricht und der zweite Eintrag dieser Array-Elemente der C-Funktion entspricht, die bei Aufruf der Lua-Funktion ausgeführt wird.

```
static const luaL_Reg sensor_methods[] = {
    {"read", _read},
    {"write", _write},
    {NULL, NULL}
};
```

Quellcode 5.8: sensorlib.c Registrierung von Funktionen

In einer solchen C-Funktion werden die Übergabeparameter der dazugehörigen Lua-Funktion eingelesen und die eigentliche, in diesem Fall mit der Lua-Funktion namensgleiche, C-Funktion, welche beispielsweise Funktionen eines Sensortreibers aufruft, wird ausgeführt.

Zusätzlich müssen Lua-Objekte aus den in C deklarierten Sensoren erstellt werden, damit auf diese zugegriffen werden kann. Dies geschieht durch die Zuweisung der Lua-Metamethode „_index“. Diese erlaubt einem Objekt Werte zuzuweisen, wenn dieses nicht gefunden wird, wie in Quellcode 5.9 zu sehen ist. Dabei wird zunächst in Zeile 5 der Sensor, auf den zugegriffen werden soll, in der verketteten Liste gesucht und in Zeile 6 wird eine Kopie davon in Lua abgebildet. Somit kann in Lua auf einen in der C-Anwendung erstellten Sensor, der zuvor der verketteten Liste hinzugefügt wurde, zugegriffen werden.

```c
static int _index(lua_State *L)
{
    const char * key = luaL_checkstring(L, 2);
    sensor_t *dev = sensor_find_name(key);
    sensor_to_lua(L, dev);
    return 1;
}
```

Quellcode 5.9: sensorlib.c Funktion _index

Der Schritt der Sensorinitialisierung in einer C-Anwendung wird in Quellcode 5.10 gezeigt, wohingegen in Quellcode 5.11 Möglichkeiten gelistet sind, in Lua darauf zuzugreifen. In Quellcode 5.12 ist zudem ersichtlich, wie auf den Sensor durch Fernzugriff mittels Lua-RPC zugegriffen wird. Bei Verbindungsaufbau mit dem Gerät, auf dessen Sensor zugegriffen werden soll, wird ein Handle zurückgegeben, welcher es ermöglicht auf die Ressourcen des verbundenen Gerätes zuzugreifen. So wird in Zeile 3 von Quellcode 5.12 auf die in Quellcode 5.11 definierte Funktion über die Lua-RPC-Bibliothek zugegriffen.

```c
int main(void)
{
    sensor_t myTempSensor = {.value=25, .name="tempSensor", .driver=
        &tempDriver};
    sensor_add(&myTempSensor);
...
```

Quellcode 5.10: main.c Server Sensorinitialisierung

```lua
local sensorLib = require "sensorLib"

print(sensorLib.tempSensor:read())
sensorLib.tempSensor:write(30)

function readTempSensor()
    return sensorLib.tempSensor:read()
end
print(readTempSensor())
```

Quellcode 5.11: Server Lua Sensorzugriff

```lua
local rpc = require "rpc"
local  slave, err = rpc.connect ("192.168.60.227",12346);
print(slave.readTempSensor())
```

Quellcode 5.12: Client Lua Sensorzugriff via Lua-RPC

5.2.5 Erweiterung von Lua-RPC

Lua-RPC bietet die Möglichkeit entweder TCP-Sockets oder eine serielle Verbindung als Kommunikationsschnittstelle zu benutzen und ist theoretisch für Implementie-

rungen mit anderen Schnittstellen erweiterbar. Allerdings können nicht gleichzeitig verschiedene Kommunikationsschnittstellen mit Lua-RPC benutzt werden. Somit können Szenarien, in denen mehrere Geräte in einer Kette miteinander verbunden sind, nicht realisiert werden, wenn sich der Transportkanal zwischen zwei Geräten ändert.

Diese Einschränkung wurde im Rahmen dieser Arbeit behoben. Die Kommandos der erweiterten Version, die solche Szenarien zulassen, benutzen einen weiteren Übergabeparameter, der die Kommunikationsschnittstelle festlegt. Beispielhaft wird dies in Quellcode 5.13 gezeigt.

```
1 local rpc = require "rpc"
2 local  slave_socket = rpc.connect ("192.168.60.227",12346,
    TRANSPORT_SOCKET);
3 local  slave_serial = rpc.connect ("/dev/pts/31", TRANSPORT_SERIAL);
```

Quellcode 5.13: Beispiel Lua-RPC Verbindungsaufbau mit Erweiterung

5.3 Zephyr

In diesem Abschnitt werden Implementierungen auf dem Betriebssystem Zephyr vorgestellt. In Unterabschnitt 5.3.1 wird zunächst ein Überblick über Zephyr und dessen Möglichkeiten einer Lua-Implementierung beschrieben. In Unterabschnitt 5.3.2 werden die notwendigen Schritte für eine native Ausführung eines Zephyr-Programmes, das einen Lua-Interpreter integriert, beschrieben. In Unterabschnitt 5.3.3 wird dies zusätzlich für Embedded-Geräte beschrieben.

5.3.1 Übersicht

Da Zephyr über keine Implementierung eines Lua-Interpreters verfügt, wird auf die Lua-C-Referenzimplementierung zur Integration zurückgegriffen. Diese ist in reinem ANSI C implementiert. (Vgl. *Lua: FAQ* 2020) Da in Zephyr allerdings lediglich eine Teilmenge der C-Standard-Bibliothek integriert ist, sind nicht sämtliche für Lua notwendigen Systemaufrufe in Zephyr implementiert. Zum Testen von Lua in Zephyr eignet sich als Zielplattform das Board „native_posix“. Dadurch wird die erstellte Zephyr-Anwendung, wie auch bei RIOT in Unterabschnitt 5.2.1 gezeigt, als Prozess auf dem Host-Computer ausgeführt. (Vgl. *Native POSIX execution (native_posix) — Zephyr Project Documentation* 2020) Die Stabilität der Implementierung wurde zusätzlich mit den Tests der offiziellen Lua-Testsuite getestet. (Vgl. *Lua: test suites* 2020)

5.3.2 Native Implementierung

Wird mit „native_posix“ als Ziel ein Lua-Programm auf Zephyr ausgeführt, so funktioniert dies ohne Quellcodeanpassungen. In Quellcode 5.14 ist eine simple Zephyr-Anwendung, die Lua benutzt, ersichtlich. Die daraus resultierende Ausgabe wird über Quellcode 5.15 dargelegt. Diese zeigt, dass die Lua-Befehle des Programmes abgearbeitet werden und keine Fehler resultieren.

In Zeile 14 von Quellcode 5.14 werden sämtliche Lua-Bibliotheken geladen. Damit auch Lua-RPC in Zephyr geladen wird, muss zunächst diese Bibliothek der Liste von Lua-Bibliotheken hinzugefügt werden, was in Quellcode 5.16 in Zeile 12 als Teil der Datei „linit.c“ geschieht, die wiederum Teil der Lua-C-Referenzimplementierung ist. Alle anderen Bibliotheken sind bereits standardmäßig ebenfalls Teil der Lua-C-Referenzimplementierung.

```
1  #include <zephyr.h>
2  #include <sys/printk.h>
3
4  #include "lauxlib.h" /* helper functions */
5  #include "lua.h"     /* core functions */
6  #include "lualib.h"  /* libraries */
7
8  void main()
9  {
10     int result;
11     lua_State* L = luaL_newstate();
12
13     printk("Opening libs ...\n");
14     luaL_openlibs(L);
15
16     printk("Compiling script ...\n");
17     result = luaL_loadstring(L, "print('Hello from Lua; ' .. 2/3)");
18     printk("Running script ...\n");
19     result = lua_pcall(L, 0, 0, 0);
20     if (result) {
21         printk("Error running script!\n");
22     }
23     printk("Shutting down lua ...\n");
24     lua_close(L);
25 }
```

Quellcode 5.14: Zephyr-Lua Hello World Programm

```
1  Opening libs ...
2  Compiling script ...
3  Running script ...
4  Hello from Lua; 0.66666666666667
5  Shutting down lua ...
```

Quellcode 5.15: Zephyr-Lua Hello World Programm Ausgabe

```
1  static const luaL_Reg loadedlibs[] = {
2      {"_G", luaopen_base},
3      {LUA_LOADLIBNAME, luaopen_package},
4      {LUA_COLIBNAME, luaopen_coroutine},
5      {LUA_TABLIBNAME, luaopen_table},
6      {LUA_IOLIBNAME, luaopen_io},
7      {LUA_OSLIBNAME, luaopen_os},
8      {LUA_STRLIBNAME, luaopen_string},
9      {LUA_MATHLIBNAME, luaopen_math},
10     {LUA_UTF8LIBNAME, luaopen_utf8},
11     {LUA_DBLIBNAME, luaopen_debug},
12     {LUA_RPCLIBNAME, luaopen_rpc},
13 #if defined(LUA_COMPAT_BITLIB)
14     {LUA_BITLIBNAME, luaopen_bit32},
15 #endif
16     {NULL, NULL}
17 };
```

Quellcode 5.16: Liste der Lua-Bibliotheken mit Lua-RPC

Wird die Lua-RPC-Bibliothek geladen, so bietet Lua zwei Mechanismen an, auf die dadurch verfügbaren Funktionen zuzugreifen:

1. Die Funktion „luaL_loadstring“ bietet die Möglichkeit Lua-Code als String zu laden.
2. Die Funktion „luaL_loadfile“ ermöglicht statt eines Strings eine Lua-Datei inklusive deren Pfad als Argument anzugeben und diese somit zu laden.

Dabei muss die Lua-RPC-Bibliothek, beispielsweise im Gegensatz zu Quellcode 5.13, nicht mehr mittels „require“ eingebunden werden, da sie mit Aufruf der Funktion „luaL_openlibs“ geladen wird, da sie in Zeile 12 in Quellcode 5.16 eingebunden wird.

Dies zeigt, dass mit Zephyr dieselben Lua-RPC-Funktionen genutzt werden können wie unter RIOT und auf einer Desktop-Linux-Distribution. Dies beinhaltet auch die in Unterabschnitt 5.2.5 beschriebene Erweiterung verschiedene Kommunikationsschnittstellen gleichzeitig zu benutzen.

5.3.3 Implementierung auf Embedded-Geräten

Zephyr bietet eine Teilimplementierung einer POSIX-Socket-kompatiblen Schnittstelle an. (Vgl. *BSD Sockets — Zephyr Project Documentation* 2020) Somit wird die Implementierung von Lua-RPC für diese Schnittstelle für Embedded Geräte erleichtert.

Als Zielplattform werden dabei von Zephyr unterstützte Boards mit Ethernet-Schnittstelle ausgewählt. In dieser Arbeit fiel die Wahl auf das SAM E70 Xplained Evaluation-Board, welches mit einem Atmel SAM E70 Mikrocontroller bestückt ist. (Vgl. *SAM E70(B) Xplained — Zephyr Project Documentation* 2020) Hierbei handelt es sich um einen 32-Bit ARM Cortex-M7-Mikrocontroller. Damit gewährleistet ist,

dass Applikationen auf mehreren verschiedenen Boards ausgeführt werden können, wurde zusätzlich ein Board mit Mikrocontroller des Herstellers STMicroelectronics ausgewählt. Dabei handelt es sich um das Nucleo-F746ZG, dessen 32-Bit-Mikrocontroller ebenfalls ein ARM Cortex-M7-Mikrocontroller ist. (Vgl. *ST Nucleo F746ZG — Zephyr Project Documentation* 2020)

Da Zephyr-Projekte für diese Boards lediglich für die TCP-Kommunikationsschnittstelle gebaut werden und nicht für die serielle, kann die Erweiterung, um mehrere Kommunikationsschnittstellen gleichzeitig zu benutzen, nicht genutzt werden. Zusätzlich müssen die Bibliotheken von Lua-RPC für die Kommunikation via TCP-Sockets mit denen von Zephyr ersetzt werden. Da die Socket-Implementierung von Zephyr deutlich neuer ist als die Lua-RPC-Bibliothek und nicht alle Funktionen der POSIX-Socket-Programmierschnittstelle implementiert sind, muss der Quellcode der Kommunikationsschnittstelle angepasst werden. So muss beispielsweise die in Lua-RPC benutzte Funktion „gethostbyname“ mit der in Zephyr implementierten Funktion „getaddrinfo“ der POSIX-Programmierschnittstelle ersetzt werden.

Zusätzlich gilt es zu beachten, dass bei einem Ziel mit 32-Bit Architektur aufgrund der ungleichen Architektur Unterschiede bei den Lua-Datentypengrößen im Vergleich zu einem 64-Bit System existieren. Das ist bei der nativen Implementierung bei RIOT und „native_posix“ als Ziel bei Zephyr nicht der Fall, da diese das Host-System nutzen. Zusätzlich ist es auch möglich, dass sich die Byte-Reihenfolge mancher Embedded-Geräte zueinander und zu Desktop-Systemen unterscheiden. Obwohl Lua-Code intepretiert wird und nicht zu Maschinencode kompiliert wird, so werden Lua-Funktionen dennoch zu Bytecode verarbeitet und in diesem Format abgespeichert. In Lua sind für das Vorkompilieren zu Bytecode und für das Laden von Bytecode die Dateien „ldump.c“ und „lundump.c“ der C-Referenzimplementierung verantwortlich. Diese können entsprechend angepasst werden, um Funktionen auf verschiedenen Plattformen auszuführen. Das eLua-Projekt, welches Lua 5.1.4 benutzt, beinhaltet bereits für diesen Zweck angepasste Versionen dieser Dateien. Daher werden diese übernommen und für weitere Entwicklungen mit Zephyr auf Embedded-Geräten wird aus diesem Grund ebenfalls Lua 5.1.4 benutzt.

Wird für diese Version eine Lua-RPC-Anwendung gebaut, so fällt auf, dass in Zephyr Implementierungen für die von Lua benötigten Systemaufrufe „_unlink“, „_link“ und „_times“ fehlen, was zu Linker-Fehlern führt. Um diese beheben zu können, müssen entweder Stubs für diese Systemaufrufe implementiert werden oder diejenigen Lua-Bibliotheken, die diese Systemaufrufe benutzen, dürfen nicht geladen werden. In dieser Arbeit wurde letzteres gewählt.

5.4 Framework-Services

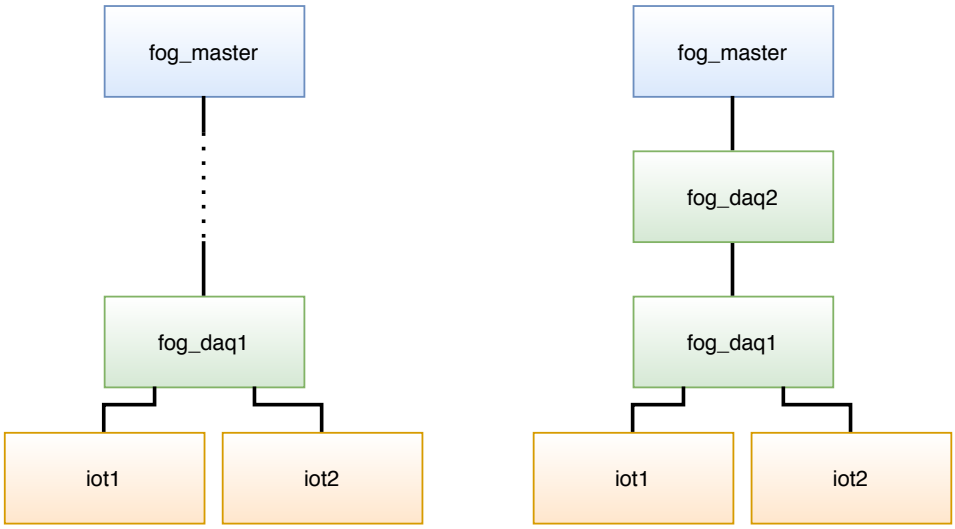
In diesem Abschnitt wird die Umsetzung von Anforderungen, die in Unterabschnitt 4.1.1 beschrieben wurden, erläutert.

5.4.1 Kommunikation

Bei der Anforderung „REQ02 Kommunikation“ wurde in Unterabschnitt 4.1.1 nach einer Kommunikationsschnittstelle zwischen Fog- und IoT-Ebene verlangt. Diese wird durch Lua-RPC entweder mittels TCP-Sockets oder als serielle Verbindung bereitgestellt und kann durch Aufruf der Funktion „connect“ gestartet werden. Dies ist beispielsweise in Quellcode 5.13 ersichtlich. Diese Funktionalität konnte somit unverändert übernommen werden.

5.4.2 Softwareverteilung

Die Anforderung „REQ03 Softwareverteilung“ wird durch Lua-RPC erfüllt. Allerdings ist die dort vorhandene Implementierung auf Punkt-zu-Punkt-Verbindungen beschränkt. Diese Einschränkung wurde im Rahmen dieser Arbeit behoben, was dazu führt, dass eine beliebige Anzahl von Geräten hintereinander zusammengeschalten werden kann, wie in Abbildung 5.5a abgebildet ist. Wird von dem konkreten Beispiel in Abbildung 5.5b ausgegangen, so kann auf dem Gerät „fog_master“ der Quellcode 5.17 ausgeführt werden. Dabei wird dem IoT-Gerät „iot1“ eine neue Funktion bekanntgegeben und einer Variable des zweiten IoT-Geräts „iot2“ wird ein Wert zugewiesen.



(a) System beliebige Layeranzahl (b) System 4 Layer

Abbildung 5.5: Systeme mit höherer Layeranzahl
Quelle: eigene Ausarbeitung

```
function add(x,y) return x+y end
fog_daq2.fog_daq1.iot1.add = add
fog_daq2.fog_daq1.iot2.myNumber = 5
```

Quellcode 5.17: Softwareverteilung über mehrere Layer

5.4.3 Softwareausführung

In Unterabschnitt 4.1.1 wurde in „REQ04 Softwareausführung“ verlangt, dass das Fog-Framework die Möglichkeit bieten sollte, verteilte Software auf den Zielgeräten zu starten und dort gegebenenfalls auszuführen. Diese Möglichkeit wird durch die Verwendung von Lua-RPC abgedeckt und wurde in Quellcode 5.4 in Zeile 8 gezeigt. In dieser Zeile werden beispielsweise zwei Funktionen auf dem Zielgerät ausgeführt: zum einen die in Lua standardmäßig vorhandene Funktion „print“ und zum anderen die auf das Zielgerät verteilte Funktion „squareval“.

Allerdings gilt für diesen Punkt eine ähnliche Punkt-zu-Punkt-Beschränkung, wie sie in Unterabschnitt 5.4.2 beschrieben wird. Standardmäßig bietet Lua-RPC keine Möglichkeit, Funktionen von nicht direkt verbundenen Geräten aufzurufen und dabei auszuführen. Diese Einschränkung wurde im Rahmen dieser Arbeit behoben. Somit ist es möglich, über eine beliebige Anzahl von Zwischengeräten Funktionen aufzurufen und diese auf dem Zielgerät ausführen zu lassen. Angenommen es gelten die gleichen Bedingungen wie in Unterabschnitt 5.4.2 für Quellcode 5.17, so sind Funktionsaufrufe wie in Quellcode 5.18 möglich.

```
x = 2;
y = 3;
z = fog_daq2.fog_daq1.iot1.add(x,y);
```

Quellcode 5.18: Funktionsaufruf über mehrere Layer

5.4.4 Ressourcen auslesen

Die Möglichkeit, Ressourcen auslesen zu können, wird in REQ05 in Unterabschnitt 4.1.1 gefordert. Lua-RPC bietet die Möglichkeit durch Aufruf der Funktion „get“ Ressourcen von direkt verbundenen Geräten auszulesen. Dies ist notwendig, wenn keine explizite Abfragefunktion für die jeweilige Ressource existiert. In dieser Arbeit wurde verzichtet diese Funktion zu erweitern, sodass auch Ressourcen von nicht direkt verbundenen Geräten ausgelesen werden können. Es wurde entschieden dies nur durch Abfragefunktion zuzulassen. Im Kontext von Abbildung 5.5 bedeutet das, dass beispielsweise Variablen von Gerät „iot2“ lediglich von „fog_daq1“ direkt ausgelesen werden können, während „fog_master“ dies nur durch Aufruf der in Quellcode 5.19 spezifizierten Abfragefunktion des Gerätes „fog_daq1“ erreicht.

```
function getIot2Value()
    return iot2.value:get()
end
```

Quellcode 5.19: Beispiel für Abfragefunktion einer Ressource eines direkt verbundenen IoT-Gerätes

5.4.5 Fehlerbehandlung

Die Anforderung „REQ06 Fehlerbehandlung“ aus Unterabschnitt 4.1.1 wird teilweise bereits durch Mechanismen von Lua-RPC erfüllt. So werden Kommunikationsfehler durch Try-Catch-Anweisungen abgefangen und es existiert ein zusätzlicher Error-Handler, der sinnvolle Fehlermeldungen mit Stacktrace ausgibt.

In Lua kann dies insofern zusätzlich erweitert werden, dass die von Lua-RPC bereitgestellten Funktionen mittels eines „protected calls“ aufgerufen werden. Dies geschieht durch die Lua-Funktion „pcall“. Damit können Laufzeitfehler behandelt werden und das Lua-Programm wird somit im Fehlerfall nicht sofort beendet, sondern ein Error-Handler kann aufgerufen werden.

Dies ist in Quellcode 5.20 beispielhaft für die Lua-RPC-Funktion „connect“ dargestellt. Die Fehlerbehandlung für Fehler bei Ausführung von nicht Lua-RPC-spezifischen Funktionen wird in Quellcode 5.21 zusätzlich angeführt.

```
function connect(...)
    local new_con;
    status, new_con, retval = pcall(rpc.connect, unpack(arg))
    if status == true then
        con[#con+1] = new_con;
    return con[#con];
    else
        error_handler("Executing rpc.connect() failed.");
    end
end
```

Quellcode 5.20: Beispiel für Fehlerbehandlung von Lua-RPC-Funktionen

```
function execFct(iotDeviceFunc,...)
    local status, retval = pcall(iotDeviceFunc,,unpack(arg))
    if status == true then
        return retval;
    else
        error_handler("Executing "..tostring(iotDeviceFunc).."
            failed. Return value: "..retval);
    end
end
```

Quellcode 5.21: Fehlerbehandlung bei Funktionsaufruf

5.4.6 Kompatibilitätsprüfung

Nach „REQ07 Kompatibilitätsprüfung“ sollte das Framework feststellen können, ob ein IoT-Gerät kompatibel mit zu verteilender Software ist. Um dies sicherzustellen, wird jedes Gerät mit einem globalen Table namens „info“ versehen. Dieser besitzt den Eintrag „type“, welcher das jeweilige Board einem Typ zuordnet. Damit anschließend lediglich passende Funktionen einem Gerät zugeordnet werden, können Funktionszuweisungen, wie beispielsweise in Zeile 2 von Quellcode 5.17 zu sehen, in if-Statements eingebettet werden. Dadurch werden einem Gerät lediglich Funktionalitäten zugeordnet, die es auch ausführen kann. Ein Beispiel hierfür ist in Quellcode 5.22 ersichtlich. Dabei kann je nach Implementierung statt der Verwendung eines Strings auch auf Datentypen zurückgegriffen werden, die weniger Speicher benötigen.

```
1 function getTemperature() return doSomething() end
2 if iotDevice.info.type:get() == 'temperature' then
3     iotDevice.getTemperature = getTemperature;
4 end
```

Quellcode 5.22: Kompatibilitätsprüfung bei Softwareverteilung

Wird die Ausführung einer dennoch fälschlicherweise zugewiesenen Funktion versucht, so resultiert ein Zugriff auf nicht-existierende Elemente in einer Lua-Fehlermeldung inklusive Stacktrace und das Programm wird beendet. Weitere Möglichkeiten gehören zur Kategorie Fehlerbehandlung, die in Unterabschnitt 5.4.5 beschrieben sind.

Wird die Fehlerbehandlung von Quellcode 5.21 benutzt, so endet der Versuch der Ausführung einer nicht ausführbaren Funktion nicht in einem Fehlerzustand des Zielgerätes. Auf dem Ausgangsgerät hingegen wird bei Implementierung des Code-Beispiels der dortige Error-Handler ausgeführt. Je nach Implementierung kann in diesem das Skript auch beendet werden.

5.4.7 Enumeration

Die Anforderung „REQ08 Enumeration“ verlangt nach der Möglichkeit verbundene Geräte aufzulisten. Um dies zu realisieren, wurde die in Quellcode 5.23 sichtbare Funktion „enumerate“ erstellt.

```
function enumerate()
    print("Amount of connected devices: " .. #con)
    local i = 1
    local deviceInfo = {}
    while i <= #con do
        deviceInfo[i] = getValue(con[i].info);
        print("Name: "..deviceInfo[i].name,"Type: "..deviceInfo[i]
            .type,"Status: "..deviceInfo[i].status);
        i = i + 1;
    end
    return deviceInfo;
end
```

Quellcode 5.23: Funktion zur Enumeration verbundener Geräte

Grundvoraussetzung für diese Umsetzung ist ein globaler Table namens „con“, der Einträge für sämtliche verbundene Geräte listet. Zusätzlich wird auch der bereits in Unterabschnitt 5.4.6 erwähnte Table „info“ auf sämtlichen verbundenen Geräten benötigt. Dieser muss die in diesem Beispiel benutzten Einträge „name“, „type“, und „status“ beinhalten. Die Funktion, die in Quellcode 5.23 sichtbar ist, kann über die Mechanismen der Softwareverteilung auf verbundene Geräte verteilt werden, wodurch die mit diesen Geräten verbundenen Geräte gelistet werden können. Dafür eignet sich allerdings eine Version ohne „print“-Statements besser, da ansonsten die Statements auf den Remote-Geräten ausgegeben werden würden und dadurch nicht sichtbar sind. Über den Rückgabewert können die verbundenen Geräte ausgelesen werden.

5.4.8 Statusabfrage

Die Anforderung „REQ09 Statusabfrage“ verlangt, dass das Fog-Framework die Möglichkeit bieten sollte den Status eines verbundenen Gerätes auszulesen. Dies erfolgt ebenfalls über den in Unterabschnitt 5.4.6 und Unterabschnitt 5.4.7 erwähnten Table „info“. Hierbei wird der Eintrag „status“ ausgelesen, der bereits in Quellcode 5.23 sichtbar ist. In der dort erkennbaren Funktion wird der Status in Zeile 7 ausgegeben.

5.4.9 Fazit

Durch die in diesem Abschnitt beschriebenen Services und Features werden von den in Unterabschnitt 4.1.1 angeführten Kriterien die Anforderungen REQ02 bis REQ09 erfüllt. Alle weiteren Anforderungen sind durch Entscheidungen aus Abschnitt 4.5 bereits erfüllt.

6 Evaluierung

In diesem Kapitel werden die Implementierungen von Kapitel 5 anhand von konkreten Anwendungsszenarien evaluiert. Szenario 1 - RIOT, Szenario 2 - Zephyr und Szenario 3 - Kombination RIOT und Zephyr dienen der Evaluierung der Implementierung auf den verschiedenen RTOS. In Abschnitt 6.4 und Abschnitt 6.5 werden durch Fog Computing einhergehende Vorteile mittels Komponenten und Features des implementierten Fog Computing-Frameworks untersucht.

6.1 Szenario 1 - RIOT

Das in diesem Unterabschnitt beschriebene Szenario dient der Evaluation der Implementierung des Fog-Frameworks unter RIOT. In Abbildung 6.1 ist der Testaufbau dieses Szenarios abgebildet. Dabei existieren drei RIOT-Instanzen, die allesamt auf der Zielplattform „native“ und in diesem Fall auf einer Desktop-Linux-Distribution als Prozess ausgeführt werden. Zusätzlich wird auf einer anderen Maschine mit einer Linux-Distribution als Betriebssystem ein Lua-Skript ausgeführt. Dieses stellt eine RPC-Verbindung via TCP zu der in der Abbildung beschriebenen Instanz „fog1“ bereit. Letztere besitzt zudem eine TCP-Verbindung sowohl zu der RIOT-Instanz „iot1“ wie auch eine serielle Verbindung zu der RIOT-Instanz „iot2“.

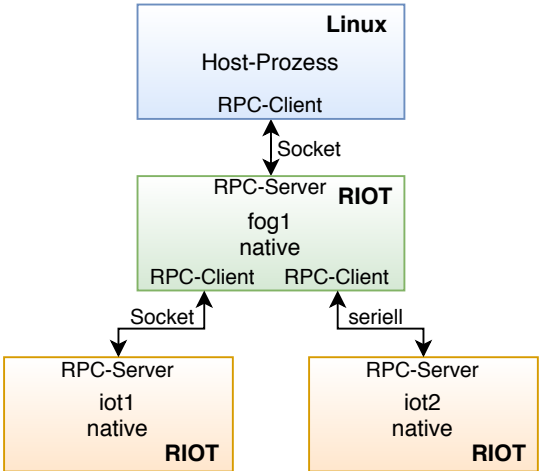


Abbildung 6.1: Evaluierungsszenario RIOT
Quelle: eigene Ausarbeitung

Alle drei RIOT-Instanzen sind mit den Fog-Framework-Komponenten ausgestattet. Die beiden letzteren gelten in diesem Szenario als IoT-Geräte, die jeweils zwei simulierte Temperatursensoren besitzen. Die Werte dieser Sensoren können - wie in Unterabschnitt 5.2.4 beschrieben - ausgelesen und mittels einer Funktion mit neuen Werten überschrieben werden.

In diesem Szenario werden die verbundenen Geräte gelistet und die Sensoren der IoT-Geräte zunächst von „fog1“ ausgelesen, überschrieben und nochmals ausgelesen. Im Anschluss daran verbindet sich der Host-Prozess mit „fog1“ und führt dieselben Operationen wiederum aus, wobei die in diesen zwei Fällen ausgeführten Befehle im Anhang dieser Arbeit in Quellcode A.1 und Quellcode A.2 sichtbar sind. Beim Auslesen durch den Host-Prozess sollten die ausgelesenen Sensorwerte zunächst mit den bereits überschriebenen Werten übereinstimmen. Die Framework-Features Kommunikation, Softwareausführung, Enumeration und Statusabfrage werden durch die benutzten Befehle auf ihre Funktionalität hin überprüft.

Die Ergebnisse sind in Quellcode 6.1 und Quellcode 6.2 sichtbar. Die verbundenen Geräte werden dabei inklusive deren Typ und Status angezeigt. Zusätzlich ist zu erkennen, dass die Sensorwerte beim zweiten Auslesen eine Wertveränderung erfahren haben und somit die Funktionen für das Auslesen und Überschreiben dieser Werte von verschiedenen Instanzen aus korrekt ausgeführt wurde. Daraus ergibt sich, dass die in diesem Szenario überprüften Framework-Features in einem Szenario mit RIOT-Instanzen einsetzbar sind.

```
Amount of connected devices: 2
Name: iot1  Type: Temperature   Status:okay
Name: iot2  Type: Temperature   Status:okay
iot1 Sensor1: 20.0
iot1 Sensor2: 30.0
iot2 Sensor1: 25.0
iot2 Sensor2: 35.0
iot1 Sensor1: 50.0
iot1 Sensor2: 60.0
iot2 Sensor1: 55.0
iot2 Sensor2: 65.0
```

Quellcode 6.1: Szenario 1 Ausgabe von „fog1“

```
Amount of connected devices: 1
Name: fog1  Type: Fog   Status:okay
Amount of connected devices: 2
Name: iot1  Type: Temperature   Status:okay
Name: iot2  Type: Temperature   Status:okay
iot1 Sensor1: 50.0
iot1 Sensor2: 60.0
iot2 Sensor1: 55.0
iot2 Sensor2: 65.0
iot1 Sensor1: 80.0
iot1 Sensor2: 90.0
iot2 Sensor1: 85.0
iot2 Sensor2: 95.0
```

Quellcode 6.2: Szenario 1 Ausgabe von Host-Prozess

Die durch dieses Szenario verifizierten Anforderungen sind zusammengefasst in Tabelle 6.1 gelistet. Die Anforderung „REQ11 Transportlayerunabhängigkeit“ wird durch den Systemaufbau des Evaluierungsszenarios erfüllt.

ID	Name	Umsetzung	Verifikation
REQ01	Ausführbarkeit	durch Entscheidungen aus Abschnitt 4.5	-
REQ02	Kommunikation	siehe Unterabschnitt 5.4.1	✓
REQ03	Softwareverteilung	siehe Unterabschnitt 5.4.2	-
REQ04	Softwareausführung	siehe Unterabschnitt 5.4.3	✓
REQ05	Ressourcen auslesen	siehe Unterabschnitt 5.4.4	-
REQ06	Fehlerbehandlung	siehe Unterabschnitt 5.4.5	-
REQ07	Kompatibilitätsprüfung	siehe Unterabschnitt 5.4.6	-
REQ08	Enumeration	siehe Unterabschnitt 5.4.7	✓
REQ09	Statusabfrage	siehe Unterabschnitt 5.4.8	✓
REQ10	Plattformunabhängigkeit	durch Entscheidungen aus Abschnitt 4.5	-
REQ11	Transportlayerunabhängigkeit	durch Entscheidungen aus Abschnitt 4.5	✓
REQ12	Erweiterbarkeit	durch Entscheidungen aus Abschnitt 4.5	-

Tabelle 6.1: Szenario 1 Verifikation von Anforderungen
Quelle: eigene Ausarbeitung

6.2 Szenario 2 - Zephyr

Das in diesem Unterabschnitt beschriebene Szenario dient der Evaluation der Implementierung des Fog-Frameworks unter Zephyr. Der Testaufbau ist in Abbildung 6.2 abgebildet. Dabei existieren vier Zephyr-Instanzen. Die abgebildeten Instanzen „fog1“ und „iot2“ werden aufgrund ihrer seriellen Verbindung auf der Zielplattform „native_posix“ ausgeführt. Eine dritte Zephyr-Instanz bildet ein SAM E70 Xplained Evaluation-Board, das über eine TCP-Verbindung mit „fog1“ verbunden ist, während die vierte Zephyr-Instanz durch ein Nucleo F746ZG Board abgebildet wird,

das via TCP mit „iot2“ verbunden ist. Zusätzlich wird auf einer weiteren Maschine mit einer Linux-Distribution als Betriebssystem ein Lua-Skript ausgeführt. Dieses stellt eine RPC-Verbindung via TCP zu „fog1“ bereit. Somit werden verschiedene Hardwarearchitekturen benutzt, da die Ausführung des Host-Prozesses auf einem Gerät mit 64-Bit-Architektur erfolgt, wohingegen die beiden Embedded-Geräte eine 32-Bit-ARM-Architektur besitzen.

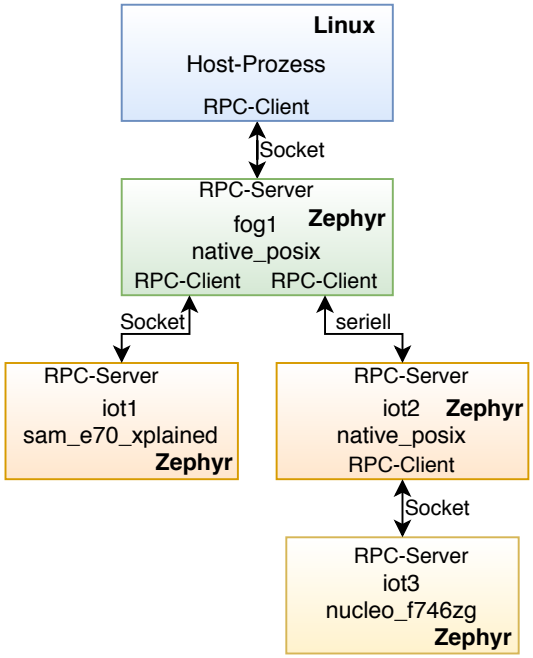


Abbildung 6.2: Evaluierungsszenario Zephyr
Quelle: eigene Ausarbeitung

In diesem Szenario besitzen die beiden „native_posix“-Instanzen die Möglichkeit, eine serielle Verbindung und eine Verbindung via TCP-Sockets gleichzeitig geöffnet zu haben. Somit wird die in Unterabschnitt 5.2.5 beschriebene Erweiterung bei diesen Instanzen angewendet. Die zwei Instanzen „iot1“ und „iot3“ besitzen jeweils in C implementierte, simulierte Sensoren, deren Werte über Lua-RPC ausgelesen werden können.

Bei diesem Szenario wird das Framework-Feature Fehlerbehandlung evaluiert, indem der Host-Prozess versucht, sich mit einem Gerät zu verbinden, welches nicht existiert. Der in diesem Szenario jeweils aufgerufene Error-Handler gibt eine Fehlermeldung aus, wodurch das Programm nicht beendet wird und somit weitere Features während des Programmablaufs getestet werden können.

Zusätzlich werden die von „iot1“ und „iot3“ simulierten und in C implementierten Sensoren ausgelesen, um die Funktionalität der C-API von Lua unter Zephyr zu zeigen.

Auch werden die in Unterabschnitt 4.1.2 definierten, nicht-funktionalen Anforderungen „REQ10 Plattformunabhängigkeit“, „REQ11 Transportlayerunabhängigkeit“ und „REQ12 Erweiterbarkeit“ getestet, indem eine Funktion im Host-Prozess, siehe Abbildung 6.2, definiert wird und über verschiedene Transportkanäle und Hardwarearchitekturen kopiert wird. Dies geschieht mittels des Fog Computing-Frameworks auf das Gerät „iot3“. Anschließend wird diese Funktion dort ausgeführt und das Ergebnis wird wieder in diesem Host-Prozess ausgegeben. Durch diesen Vorgang wird gleichzeitig das Framework-Feature Softwareverteilung getestet.

Anschließend wird in diesem Szenario das Framework-Feature der Kompatibilitätsprüfung aus Unterabschnitt 5.4.6 evaluiert, indem versucht wird eine Funktion für Geräte des Typs „Temperature“ auf das Gerät „iot3“ zu kopieren. Dieses Gerät ist in diesem Szenario allerdings vom Typ „Generic“, wodurch der Versuch fehlschlagen sollte. Die Befehlsabfolge zur Evaluierung der Möglichkeiten ist in Anhang A.2 in Quellcode A.3 ersichtlich.

Diese Befehle werden im Host-Prozess ausgeführt. In Quellcode 6.3 ist dessen resultierende Ausgabe gelistet. So ist in Zeile 1 und 2 von Quellcode 6.3 die resultierende Fehlermeldung bei Versuch einer Verbindung zu einer nicht existierenden IP-Adresse ersichtlich, während die Zeilen 3 bis 9 der Reihe nach die Ausgabe einer Enumeration der drei Instanzen „Host-Prozess“, „fog1“ und „iot2“ zeigen. In den Zeilen 10 bis 13 werden die in C implementierten und simulierten Sensorwerte durch den Host-Prozess ausgelesen. Dabei werden zunächst die von „iot3“ simulierten Sensorwerte mittels Aufrufes der Funktion „get“ von Lua-RPC innerhalb einer auf „iot2“ existierenden Funktion ausgelesen. Für die Sensorwerte von „iot1“ werden hingegen auf dieser Instanz existierende Abfragefunktionen verwendet. Dieser Unterschied ist in Anhang A.2 in Quellcode A.3 ersichtlich.

In Zeile 14 von Quellcode 6.3 ist die Ausgabe eines Strings und einer Gleitkommazahl sichtbar, welche durch eine auf „iot3“ kopierte und dort ausgeführte Funktion hervorgerufen wird. Letztendlich ist in Zeile 15 eine Fehlermeldung - bedingt durch Einsatz der Kompatibilitätsprüfung - erkennbar.

```
Err: Executing rpc.connect() failed.
Parameter:  192.168.1.404   12345
Amount of connected devices: 1
Name: fog1  Type: Fog   Status:okay
Amount of connected devices: 2
Name: iot1  Type: Generic   Status:okay
Name: iot2  Type: Generic   Status:okay
Amount of connected devices: 1
Name: iot3  Type: Generic   Status:okay
Value of 'iot3' sensor 1: 5
Value of 'iot3' sensor 2: 6
Value of 'iot1' sensor 1: 3
Value of 'iot1' sensor 2: 2
Text and Number:1.8571428571429
Err: Device type of iot3 != 'Temperature'
```

Quellcode 6.3: Szenario 2 Ausgabe von Host-Prozess

In diesem Szenario wurden die nicht-funktionalen Anforderungen REQ10 bis REQ12 aus Unterabschnitt 4.1.2 behandelt. Dabei handelt es sich um sämtliche in diesem Unterabschnitt gelisteten Anforderungen. Ebenfalls wurden sämtliche in Abschnitt 5.4 genannten Framework-Services in diesem Szenario getestet. Zusätzlich ist erkennbar, dass der gewählte Kommunikationsmechanismus RPC die an ihn gestellte Anforderung „REQ13 Unabhängigkeit des Kommunikationskanals“ durchaus erfüllt und somit Verbindungen zwischen Geräten nicht nur via TCP, sondern auch über eine serielle Verbindung möglich sind.

Die Anforderung „REQ01 Ausführbarkeit“ verlangt, dass das Fog-Framework ebenfalls auf ressourcenbeschränkten Systemen ausführbar sein sollte. Dies wird in diesem Szenario durch die Ausführung auf dem SAM E70 Xplained Evaluation-Board und dem Nucleo F746ZG Board erreicht. In Quellcode 6.4 ist für eine genauere Betrachtung des Speicherverbrauchs derjenige des SAM E70 Xplained Evaluation-Boards ersichtlich. Erkennbar ist, dass 230 KB an Flash benutzt werden und etwa 37 KB an SRAM. Zuvor haben keinerlei Speicheroptimierungen stattgefunden. So kann der benötigte Flash-Speicher durch Exklusion von nicht benötigten Lua-Bibliotheken und den Verzicht auf Textausgabe zusätzlich reduziert werden.

```
Memory region    Used Size    Region Size    %age Used
FLASH:           230312 B     2 MB           10.98%
SRAM:            36816 B      384 KB         9.36%
```

Quellcode 6.4: Szenario 2 - Speicherverbrauch von SAM E70 Xplained Evaluation-Board

Die durch dieses Szenario verifizierten Anforderungen sind zusammengefasst in Tabelle 6.2 gelistet. Sichtbar ist, dass sämtliche Anforderungen, die in Abschnitt 4.1 an das Fog Computing-Framework gestellt wurden, in diesem Szenario verifiziert worden sind.

ID	Name	Umsetzung	Verifikation
REQ01	Ausführbarkeit	durch Entscheidungen aus Abschnitt 4.5	✓
REQ02	Kommunikation	siehe Unterabschnitt 5.4.1	✓
REQ03	Softwareverteilung	siehe Unterabschnitt 5.4.2	✓
REQ04	Softwareausführung	siehe Unterabschnitt 5.4.3	✓
REQ05	Ressourcen auslesen	siehe Unterabschnitt 5.4.4	✓
REQ06	Fehlerbehandlung	siehe Unterabschnitt 5.4.5	✓
REQ07	Kompatibilitätsprüfung	siehe Unterabschnitt 5.4.6	✓
REQ08	Enumeration	siehe Unterabschnitt 5.4.7	✓
REQ09	Statusabfrage	siehe Unterabschnitt 5.4.8	✓
REQ10	Plattformunabhängigkeit	durch Entscheidungen aus Abschnitt 4.5	✓
REQ11	Transportlayerunabhängigkeit	durch Entscheidungen aus Abschnitt 4.5	✓
REQ12	Erweiterbarkeit	durch Entscheidungen aus Abschnitt 4.5	✓

Tabelle 6.2: Szenario 2 Verifikation von Anforderungen
Quelle: eigene Ausarbeitung

6.3 Szenario 3 - Kombination RIOT und Zephyr

In den vorangegangenen Szenarien wurden bereits sämtliche Framework-Features aus Abschnitt 5.4 behandelt. Allerdings wurde dabei entweder auf RIOT oder auf Zephyr als RTOS gesetzt. In diesem Szenario kommen nun beide zum Einsatz.

Der Testaufbau dieses Szenarios ist in Abbildung 6.3 abgebildet. Dabei existieren zwei Zephyr-Instanzen, eine RIOT-Instanz und ein Lua-RPC-Prozess, der als Host-Prozess in einer Linux-Desktopumgebung ausgeführt wird. Da der Lua-Interpreter von RIOT für Lua 5.3 spezifiziert ist, jener für die Implementiung auf Zephyr allerdings Lua 5.1.4 benutzt und diese beiden Versionen inkompatibel miteinander sind, wird für sämtliche Instanzen dieses Szenarios Lua 5.3 verwendet. Somit werden in diesem Szenario keine Embedded-Geräte, sondern native Instanzen zur Evaluierung genutzt. Ebenfalls ist der Host-Prozess im Gegensatz zu den vorherigen Szenarien über eine serielle Verbindung mit dem Fog-Gerät verbunden.

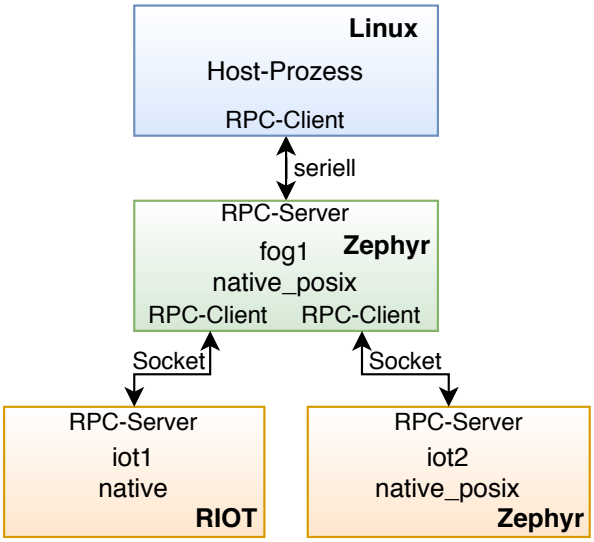


Abbildung 6.3: Evaluierungsszenario Kombination RIOT und Zephyr
Quelle: eigene Ausarbeitung

Im Vergleich zu Szenario 1 - RIOT werden in diesem Szenario Funktionen auf andere Geräte verschoben. In Szenario 1 wurden lediglich auf den Geräten bereits existierende Funktionen via RPC auf anderen Geräten aufgerufen. Somit wird in diesem Szenario die Anforderung „REQ03 Softwareverteilung“ überprüft. Da dafür eine funktionierende Kommunikationsschnittstelle notwendig ist, wird damit „REQ02 Kommunikation“ und bei Ausführung der verteilten Software „REQ04 Softwareausführung“ überprüft.

Dieses Szenario dient zudem als Stresstest für das System. So wird nach Herstellung der Verbindung zwischen den Instanzen eine Funktion vom Host-Prozess auf die restlichen drei Instanzen kopiert und dann via Lua-RPC 100.000-mal auf diesen Instanzen aufgerufen. Dadurch soll die Stabilität des Systems und zudem der Speicherbedarf eines solchen Programms überprüft werden. Letzteres ist mittels der Lua-Funktion „collectgarbage“ möglich.

Die Befehlsabfolge zur Evaluierung ist in Anhang A.3 in Quellcode A.4 ersichtlich. Diese Befehle werden im Host-Prozess ausgeführt. In Quellcode 6.5 ist die sich daraus resultierende Ausgabe dargestellt.

Dabei ist in den Zeilen 1 bis 5 ersichtlich, dass das Framework-Feature Enumeration auch für dieses Szenario verfügbar ist und somit „REQ08 Enumeration“ und „REQ09 Statusabfrage“ befriedigt werden. In den Zeilen 6 bis 9 wird der von Lua benötigte Speicher der einzelnen Instanzen nach erfolgreichem Verbindungsaufbau gelistet.

In den Zeilen 12 bis 15 wird dasselbe nochmals gelistet. Allerdings wurde in der Zwischenzeit eine zuvor via Lua-RPC kopierte Funktion mittels des RPC-Mechanismus 100.000-mal auf der Instanz „fog1“ ausgeführt. Dabei zeigt sich, dass sich lediglich der Speicherverbrauch des Host-Prozesses ändert. Der Speicherverbrauch von „fog1“ wird

aufgrund der im Vergleich zum Host-Prozess geringen Änderung als gleichbleibend betrachtet.

In den Zeilen 18 bis 21 wird der Speicherverbrauch nach 100.000-facher Ausführung derselben Funktion auf den Instanzen „iot1“ und „iot2“ angeführt. Dabei wird deutlich, dass der Speicherverbrauch von „iot1“ und „iot2“ sich kaum verändert. Der Verbrauch des Host-Prozesses und von „fog1“ betragen allerdings jeweils mehrere Megabyte.

Die Zeilen 24 bis 27 zeigen abschließend den Speicherverbrauch nach mehrmaligem, manuell gestarteten Durchlauf des Garbage Collectors. Dabei unterschreitet die Größe des benötigen Speichers von „iot1“ und „iot2“ jene zu Beginn, die sich aus den Zeilen 8 und 9 ergeben. Der Speicherverbrauch des Host-Prozesses und von „fog1“ liegt allerdings dennoch bei über 6 bzw. 3 Megabyte.

```
1  Anzahl verbundener Geräte: 1
2  Name: fog1  Type: Fog   Status: okay
3  Amount of connected devices: 2
4  Name: iot1  Type: Generic   Status:okay
5  Name: iot2  Type: Generic   Status:okay
6  Memory usage Host: 32.654296875
7  Memory usage fog1: 27.673828125
8  Memory usage iot1: 15.177734375
9  Memory usage iot2: 25.0966796875
10
11 Done running function on fog1.
12 Memory usage Host: 4837.6748046875
13 Memory usage fog1: 28.072265625
14 Memory usage iot1: 15.177734375
15 Memory usage iot2: 25.0966796875
16
17 Done running function on IoT devices.
18 Memory usage Host: 28600.452148438
19 Memory usage fog1: 18996.819335938
20 Memory usage iot1: 16.43359375
21 Memory usage iot2: 25.4775390625
22
23 Running garbage collector...
24 Memory usage Host: 6170.3544921875
25 Memory usage fog1: 3094.5224609375
26 Memory usage iot1: 14.9931640625
27 Memory usage iot2: 21.2958984375
```

Quellcode 6.5: Szenario 3 - Ausgabe von Host-Prozess

Somit ist erkennbar, dass der Speicherverbrauch von Geräten, die zu keinen weiteren Geräten verbunden sind und lediglich die RPC-Server-Funktionalitäten bereitstellen, nahezu unverändert bleibt. Auf solchen Geräten wird somit die Anforderung „REQ01 Ausführbarkeit“ erfüllt.

Die durch dieses Szenario verifizierten Anforderungen sind zusammengefasst in Tabelle 6.3 gelistet. Die Anforderungen REQ10 bis REQ12 werden durch den Systemaufbau

des Evaluierungsszenarios und die Verschiebung von Funktionen über mehrere Geräte hinweg erfüllt.

ID	Name	Umsetzung	Verifikation
REQ01	Ausführbarkeit	durch Entscheidungen aus Abschnitt 4.5	✓
REQ02	Kommunikation	siehe Unterabschnitt 5.4.1	✓
REQ03	Softwareverteilung	siehe Unterabschnitt 5.4.2	✓
REQ04	Softwareausführung	siehe Unterabschnitt 5.4.3	✓
REQ05	Ressourcen auslesen	siehe Unterabschnitt 5.4.4	-
REQ06	Fehlerbehandlung	siehe Unterabschnitt 5.4.5	-
REQ07	Kompatibilitätsprüfung	siehe Unterabschnitt 5.4.6	-
REQ08	Enumeration	siehe Unterabschnitt 5.4.7	✓
REQ09	Statusabfrage	siehe Unterabschnitt 5.4.8	✓
REQ10	Plattformunabhängigkeit	durch Entscheidungen aus Abschnitt 4.5	✓
REQ11	Transportlayerunabhängigkeit	durch Entscheidungen aus Abschnitt 4.5	✓
REQ12	Erweiterbarkeit	durch Entscheidungen aus Abschnitt 4.5	✓

Tabelle 6.3: Szenario 3 Verifikation von Anforderungen
Quelle: eigene Ausarbeitung

6.4 Latenz

Als Motivation für Fog Computing wurde zu Beginn dieser Arbeit in Abschnitt 1.1 der Vorteil einer Latenzreduktion genannt. Dabei handelt es sich somit um ein Ziel des Fog Computing-Framework, auf das nicht bereits in den vorhergehenden Evaluierungsszenarien eingegangen wurde.

Somit wird in diesem Szenario evaluiert, wie sich der Einsatz von Fog Computing auf die Latenz auswirkt. Dabei wird Latenz in diesem Abschnitt als jene Zeit zwischen Einlesen der Daten am Sensor und Reaktion des Systems durch Ansteuerung eines Aktors definiert.

Zunächst wird die Signalumlaufzeit zwischen einem IoT-Gerät und einem Fog-Gerät sowie jene zwischen einem Fog-Gerät und einem Server in der Cloud gemessen. Dies geschieht mittels Ausführung des „ping“-Befehl auf einer Linux-Desktopumgebung, welche in diesem Szenario als Synonym für ein Fog-Gerät verstanden werden kann. Mittels der Implementierung dieses Kommandos können Ziele dauerhaft gepingt werden und bei Beendigung des Kommandos werden automatisch Statistiken erfasst.

Als IoT-Gerät wird in diesem Szenario ein SAM E70 Xplained Board im selben Netzwerk genutzt und somit gepingt. In Quellcode 6.6 sind die Statistiken des Pingvorganges zu sehen. Dabei wurde das Embedded-Gerät 1.000-mal gepingt, der arithmetische Mittelwert der Signalumlaufzeit beträgt dabei 0,321 ms.

Um die Laufzeit zu einer Cloud zu messen, wird auf eine IP-Adresse des öffentlichen DNS-Servers von Google zurückgegriffen. Dabei wird diese Adresse wiederum 1.000-mal gepingt. Das Ergebnis ist in Quellcode 6.7 erkennbar. Der sich ergebende arithmetische Mittelwert der Signalumlaufzeit entspricht dabei 5,033 ms.

```
--- 192.168.60.219 ping statistics ---
1000 packets transmitted, 982 received, 1% packet loss, time 999772ms
rtt min/avg/max/mdev = 0.192/0.321/23.300/0.956 ms
```

Quellcode 6.6: Ping Statistiken Fog

```
--- 8.8.8.8 ping statistics ---
1000 packets transmitted, 1000 received, 0% packet loss, time 1000369ms
rtt min/avg/max/mdev = 4.390/5.033/35.811/1.258 ms
```

Quellcode 6.7: Ping Statistiken Cloud

In Abbildung 6.4 ist ein Sequenzdiagramm abgebildet, das einen Anwendungsfall für IoT-Geräte darstellt. Dabei werden von einem IoT-Gerät periodisch Sensordaten des damit verbundenen Sensors angefordert und über ein Fog-Gerät, welches lediglich als Gateway fungiert, für die Verarbeitung in die Cloud gesendet, worauf daraufhin ein Aktor eingestellt wird. Dabei wird auf das Paradigma des Fog Computing verzichtet und die gesamte Umlaufzeit bzw. Latenz wird als T_{CC} bezeichnet.

Unter der Annahme, dass die Laufzeit von Daten zwischen zwei Teilnehmern unabhängig von deren Richtung ist, so lässt sich T_{CC} als Gleichung (6.1) anschreiben. Dabei gilt, dass $2 \cdot T_{fogIot}$ der Signalumlaufzeit zwischen Fog- und IoT-Gerät und $2 \cdot T_{fogCloud}$ der Signalumlaufzeit zwischen Fog-Gerät und Cloud entspricht. T_{iot} entspricht dem Zeitraum den das IoT-Gerät zwischen Start der „readSensorData()“-Anfrage und Versenden des Rückgabewertes an das Fog-Gerät benötigt. Zusätzlich entspricht T_{cloud} dem Zeitraum den die Cloud für die Datenverarbeitung und das Versenden der Reaktion benötigt und $T_{triggerAktor}$ ist jene Zeit, die das IoT-Gerät für das Einstellen des Aktors benötigt.

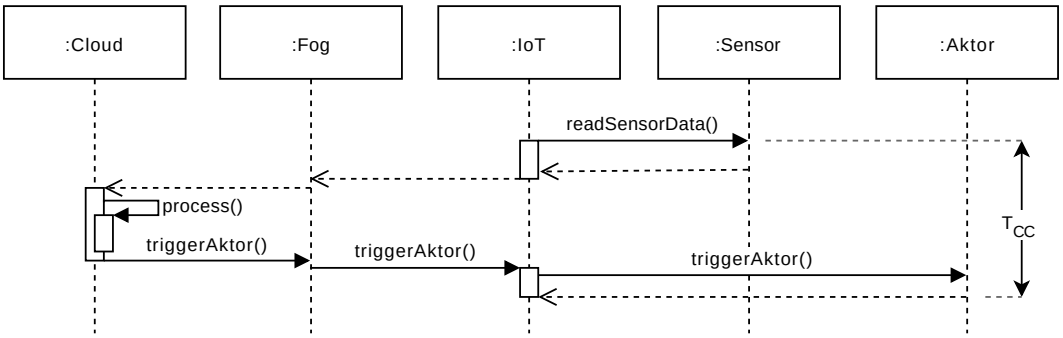


Abbildung 6.4: Sequenzdiagramm Latenz Ausgangslage
Quelle: eigene Ausarbeitung

$$T_{CC} = T_{iot} + 2 \cdot T_{fogIot} + 2 \cdot T_{fogCloud} + T_{cloud} + T_{triggerAktor} \tag{6.1}$$

Im Gegensatz dazu ist in Abbildung 6.5 ein Sequenzdiagramm dargestellt, dass Fog Computing-Funktionalitäten auf dem Fog-Gerät implementiert, wobei die dafür benötigte Gesamtzeit T_{FC} genannt wird. Somit findet statt einer Datenverarbeitung in der Cloud eine Datenverarbeitung auf dem Fog-Gerät statt, welche den Zeitraum T_{fog} benötigt und die Umlaufzeit zwischen Fog-Gerät und Cloud entfällt. Lediglich für die Speicherung werden Daten in die Cloud geschickt. T_{FC} lässt sich daher entsprechend Gleichung (6.2) berechnen.

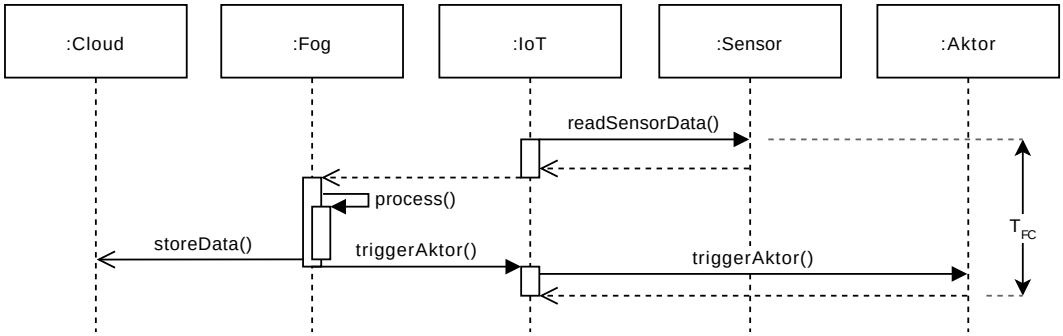


Abbildung 6.5: Sequenzdiagramm Latenz Fog Computing
Quelle: eigene Ausarbeitung

$$T_{FC} = T_{iot} + 2 \cdot T_{fogIot} + T_{fog} + T_{triggerAktor} \tag{6.2}$$

Werden die Gleichungen (6.1) und (6.2) gleichgesetzt, so ergibt sich Gleichung (6.3). Diese zeigt, dass, wenn die Latenz T_{CC} der Latenz T_{FC} entspricht, T_{fog} die Summe von Signalumlaufzeit zwischen Fog-Gerät und Cloud und Verarbeitungszeit in der Cloud ist. Damit durch Einsatz von Fog Computing die Latenz reduziert wird, muss T_{fog} allerdings kleiner sein. Somit wird $T_{\mathrm{fog}_{max}}$ als Grenzwert in Gleichung (6.4) definiert, den T_{fog} für die Gewährleistung einer Latenzreduktion nicht erreichen darf. Unter der Annahme, dass in der Cloud eine deutlich höhere Rechenleistung zur Verfügung steht, als auf einem Fog-Gerät, so lässt sich zudem Ungleichung (6.5) anführen. Somit ist die Zeit T_{cloud} sehr klein gegenüber der Verarbeitungszeit auf dem Fog-Gerät und kann aufgrund des geringen Einflusses vernachlässigt werden, wodurch sich Gleichung (6.6) ergibt, welche besagt, dass $T_{\mathrm{fog}_{max}}$ etwa der gemessenen Umlaufzeit zwischen Fog-Gerät und Cloud entspricht.

$$T_{fog} = 2 \cdot T_{fogCloud} \ + \ T_{cloud} \tag{6.3}$$

$$T_{fog_{max}} = \underbrace{2 \cdot T_{fogCloud}}_{5{,}033\,\mathrm{ms}} + T_{cloud} \tag{6.4}$$

$$T_{cloud} << T_{fog} < T_{fog_{max}} \tag{6.5}$$

$$T_{fog_{max}} = 5,033\,\mathrm{ms} + \cancelto{0}{T_{cloud}} \approx\ 5,033\,\mathrm{ms} \tag{6.6}$$

Somit muss die Zeit für die Datenverarbeitung auf dem Fog-Gerät T_{fog} kleiner als etwa 5,033 ms sein, damit eine Latenzreduktion durch Einsatz von Fog Computing stattfindet.

6.5 Datenreduktion

Neben der Latenzreduktion wurde in Abschnitt 1.1 auch eine mögliche Datenreduktion und eine damit einhergehende mögliche Netzwerkentlastung als Vorteil für Fog Computing genannt.

In diesem Abschnitt wird dieser Vorteil anhand eines Beispiels durch Berechnungen betrachtet und anschließend mittels einer Implementierung auch empirisch untersucht.

In diesem Beispiel wird zur Glättung eines gemessenen Sensorsignals ein gleitender Mittelwert via Fog Computing-Framework auf den IoT-Geräten benutzt. Dadurch können hohe Frequenzen, die möglichen Störungen entsprechen können, herausgefiltert werden. Der gleitende Mittelwert besitzt in diesem Beispiel eine Fensterbreite von 16 und alle 4 Messwerte wird der resultierende Wert des gleitenden Mittelwerts zu einem Fog-Gerät geschickt. In diesem Beispiel werden die Werte des nun geglätteten Signals direkt weiter in die Cloud geleitet. Das Sequenzdiagramm dieses Beispiels ist in Abbildung 6.6 abgebildet. Dabei wird davon ausgegangen, dass im Takt von 1 ms die Sensordaten ausgelesen werden. Im Gegensatz dazu ist in Abbildung 6.7 das Sequenzdiagramm der Ausgangslage eines solchen Beispiels abgebildet, bei dem auf Fog Computing-Funktionalität verzichtet wird und jeder Sensorwert einzeln zum Fog-Gerät gesendet wird.

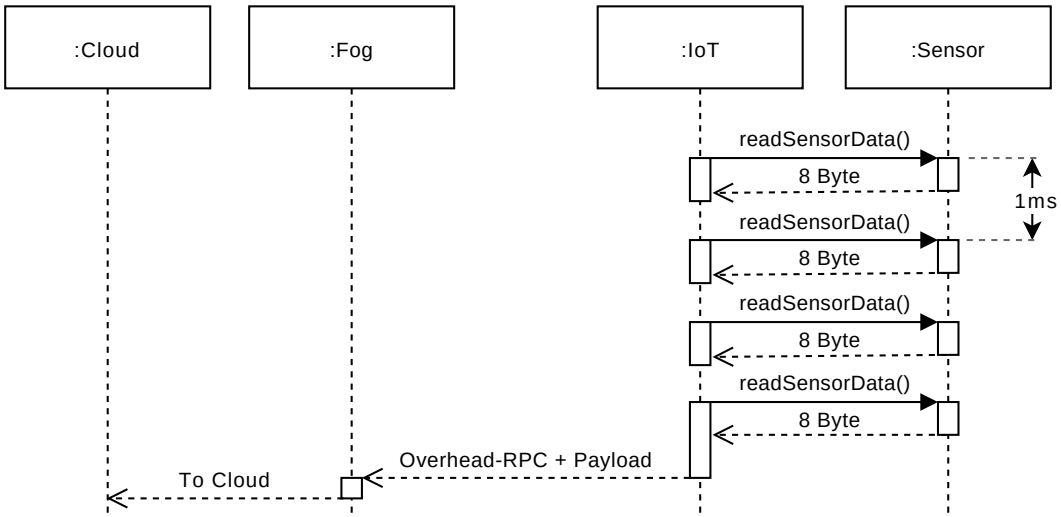


Abbildung 6.6: Sequenzdiagramm Datenreduktion Fog Computing
Quelle: eigene Ausarbeitung

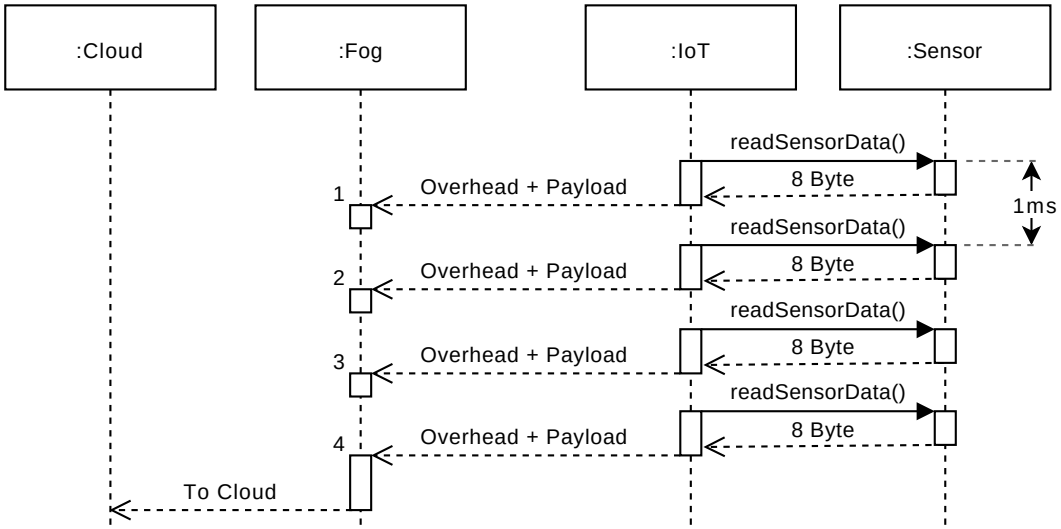


Abbildung 6.7: Sequenzdiagramm Datenreduktion Ausgangslage
Quelle: eigene Ausarbeitung

Da lediglich nach je 4 Messwerten der Mittelwert an das Fog-Gerät gesendet wird, verringert sich die Datenmenge, die in die Cloud geschickt wird, um den Faktor 4.

Da durch das Fog Computing-Framework ein zusätzlicher Overhead erzeugt wird, enthält die in einem TCP-Segment transportierte Nutzlast auch Lua-RPC-spezifische Informationen. Das wird erkennbar, wenn die Kommunikation mittels Wireshark analysiert wird. In diesem Beispiel wird 400.000-mal ein 64-Bit-Zahlenwert ausgelesen.

Dabei werden 8.001.127 Byte an Nutzlast transportiert. Dies entspricht abgerundet 20 Byte pro Auslesevorgang eines 64-Bit- bzw. 8-Byte-Zahlenwertes, was in Gleichung (6.7) beschrieben ist. Somit sind in diesem Beispiel lediglich 40 % der in einem TCP-Segment vorhandenen Nutzdaten tatsächlich Sensordaten und die restlichen 60 % davon sind Overhead, der durch die Benutzung des Fog Computing-Frameworks zusätzlich verursacht wird. Dies ist in Gleichung (6.8) sichtbar.

$$\frac{8.001.127\,\text{Byte}}{400.000} = 20,0028175\,\text{Byte} \approx 20\,\frac{\text{Byte}}{\text{Auslesevorgang}} \tag{6.7}$$

$$20\,\text{Byte} = \underbrace{8\,\text{Datenbyte}}_{40\,\%} + \underbrace{12\,\text{Overhead-Byte}}_{60\,\%} \tag{6.8}$$

Durch den in diesem Beispiel benutzten Algorithmus des gleitenden Mittelwerts würde sich somit keine tatsächliche Datenmengenreduktion um den Faktor 4 ergeben. Allerdings würden, falls der Overhead der Ausgangslage in Abbildung 6.7 außer Acht gelassen wird, dennoch lediglich 62,5 % der ursprünglichen Datenmenge transportiert werden, was in Gleichung (6.9) sichtbar ist. In Gleichung (6.10) ist zusätzlich ersichtlich, dass bei einem Overhead von 2 Byte pro Auslesevorgang die zu transportierende Datenmenge bei diesem Beispiel halbiert wird.

$$\frac{20\,\text{Byte}}{4 \cdot 8\,\text{Datenbyte}} = \frac{20\,\text{Byte}}{32\,\text{Datenbyte}} = 62,5\,\% \tag{6.9}$$

$$\frac{20\,\text{Byte}}{4 \cdot (8\,\text{Datenbyte} + 2\,\text{Overhead-Byte})} = \frac{20\,\text{Byte}}{40\,\text{Byte}} = 50\,\% \tag{6.10}$$

Somit ergibt sich, dass der in Abschnitt 1.1 angesprochene Punkt der Datenreduktion durchaus ein möglicher Effekt von Fog Computing sein kann.

7 Zusammenfassung und Ausblick

In diesem Kapitel wird in Abschnitt 7.1 eine Zusammenfassung der vorliegenden Arbeit gegeben. Daraufhin wird abschließend in Abschnitt 7.2 noch ein Ausblick gegeben.

7.1 Zusammenfassung

In dieser Arbeit wurde ein Framework erstellt, welches Fog Computing-Funktionalitäten durch Kombination der Skriptsprache Lua und RPC als Kommunikationsmechanismus auf unterschiedlichen Plattformen ermöglicht. Dabei wurden verschiedene Services bzw. Features für das Framework implementiert und in mehreren Szenarien evaluiert. Durch ein solches Softwaresystem lassen sich Services für „smarte“ Anwendungsgebiete, beispielsweise Smart Grid, Smart Vehicle oder Smart Factory, auf ressourcenbeschränkten Systemen entwickeln und verteilen.

Grundlage für Designentscheidungen bildeten dabei festgelegte funktionale und nichtfunktionale Anforderungen. Mittels dieser Anforderungen wurde festgelegt, welche Funktionalitäten ein Fog Computing-Framework bereitstellen muss, welches die Verteilung von Services auf ressourcenbeschränkten Systemen zulässt. Zusätzlich wurde dadurch die zweite Forschungsfrage aus Abschnitt 1.2 beantwortet, welche Funktionalitäten dafür in Frage kommen.

Aufgrund des Aspektes der Ausführbarkeit auf Systemen, die über beschränkte Rechen- und Speicherkapazitäten verfügen, wurden Implementierungen für die Echtzeitbetriebssysteme Zephyr und RIOT erstellt.

Da die Anbindung von Embedded-Geräten über verschiedene Transportkanäle erfolgen kann, wurde bei der Implementierung zudem auf eine Transportkanalunabhängigkeit geachtet. Dies wurde durch die Wahl von RPC als Kommunikationsmechanismus erreicht und mittels der Benutzung von TCP- und seriellen Verbindungen in den Evaluationsszenarien überprüft.

Diese Szenarien zeigten, dass die implementierten Services und Features der Softwareverteilung, Softwareausführung, Fehlerbehandlung, Kompatibilitätsprüfung, Enumeration und Statusabfrage, sowie das Auslesen von Ressourcen, auf unterschiedlichen Plattformen genutzt werden können. Weiters wurde der benötigte Speicher für eine Embedded-Gerät-Art ermittelt und durch einen Stresstest des Frameworks wurde

der zur Laufzeit benötigte Speicher von verschiedenen Geräten in einem Netzwerk ermittelt.

Zusätzlich wurde die Veränderung der Latenz durch Einführung von Fog Computing betrachtet und es wurde aufgezeigt, dass Anwendungsszenarien für Fog Computing existieren, welche die Datenmenge, die in die Cloud geschickt wird, reduzieren können und somit gleichzeitig eine Netzwerkentlastung ermöglichen.

Somit kann als Antwort auf die erste Forschungsfrage aus Abschnitt 1.2 festgehalten werden, dass sich der Mechanismus RPC in Kombination mit Lua eignet, um Services auf ressourcenbeschränkten Geräten in einem Fog-Netzwerk zu verteilen.

Zusammenfassend zeigt diese Arbeit somit einen Ansatz für Fog Computing auf ressourcenbeschränkten Geräten und kann als Grundlage für weitere Forschung im Bereich des Fog Computings genutzt werden.

7.2 Ausblick

In dieser Arbeit wurde gezeigt, dass das Fog Computing-Framework mit verschiedenen Transportkanälen und Geräten genutzt werden kann. In weiterer Folge können daraus Anwendungen entstehen, die auf Eigenschaften des Fog Computing-Paradigmas angewiesen sind und beispielsweise dynamisch und automatisiert Services auf Geräte verteilen.

Da für Embedded-Geräte lediglich eine Anbindung mittels TCP-Verbindungen gezeigt wurde, könnte dies als nächster Schritt für eine serielle Verbindung ermöglicht werden. Das sollte die Kapazitäten des Frameworks auf Geräten mit noch größerer Ressourcenbeschränktheit testbar machen und die Grenzen des Frameworks ausloten.

Zusätzlich sind Optimierungen des vorgestellten Frameworks denkbar. So könnte in weiterer Folge eine Qualifizierung eines Lua-Just-In-Time-Kompilers stattfinden, um die Ausführungsgeschwindigkeit von Lua-Code zu erhöhen.

Weiters sind Weiterentwicklungen der in dieser Arbeit genutzten Bibliothek Lua-RPC vorstellbar, um derzeit bestehende Beschränkungen aufzuheben. Damit ist die Einführung eines asynchronen Kommunikationsmechanismus vorstellbar, aber auch die Möglichkeit, mehrere Clients zu einem Server zu verbinden.

Literaturverzeichnis

Amjad, Anas u. a. (Mai 2017): „Cognitive Edge Computing based resource allocation framework for Internet of Things“. en. In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. Valencia, Spain: IEEE, S. 194–200. ISBN: 978-1-5386-2859-1. DOI: 10.1109/FMEC.2017.7946430. URL: http://ieeexplore.ieee.org/document/7946430/ (Zugriff am: 19. 04. 2020) (siehe S. 11).

Arm Limited (2020a): *Mbed OS | Mbed*. URL: https://os.mbed.com/mbed-os/ (Zugriff am: 22. 06. 2020) (siehe S. 20).

Arm Limited (2020b): *Requirements for embedded Client production devices - Device and tool requirements | Pelion Device Management Documentation*. URL: https://www.pelion.com/docs/device-management/current/cloud-requirements/requirements-for-production-devices.html (Zugriff am: 22. 06. 2020) (siehe S. 20).

Armbrust, Michael u. a. (Apr. 2010): „A view of cloud computing“. en. In: *Communications of the ACM* 53.4, S. 50–58. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1721654.1721672. URL: https://dl.acm.org/doi/10.1145/1721654.1721672 (Zugriff am: 19. 04. 2020) (siehe S. 5).

Ashton, Kevin (Juni 2009): „That 'Internet of Things' Thing“. en. In: Library Catalog: www.rfidjournal.com. URL: https://www.rfidjournal.com/that-internet-of-things-thing (siehe S. 4).

Bachmann, Kevin (Feb. 2017): *Design and Implementation of a Fog Computing Framework*. de (siehe S. 2).

Becker, Ulrich und Kästner, Tobias (Dez. 2019): *Fit for Digitalization: So entsteht eine moderne System-Architektur*. deutsch. Sindelfingen (siehe S. 20).

Bonomi, Flavio; Milito, Rodolfo; Natarajan, Preethi u. a. (2014): „Fog Computing: A Platform for Internet of Things and Analytics“. en. In: Bessis, Nik und Dobre, Ciprian (Hrsg.): *Big Data and Internet of Things: A Roadmap for Smart Environments*. Bd. 546. Series Title: Studies in Computational Intelligence. Cham: Springer International Publishing, S. 169–186. ISBN: 978-3-319-05028-7 978-3-319-05029-4. DOI: 10.1007/978-3-319-05029-4_7. URL: http://link.springer.com/10.1007/978-3-319-05029-4_7 (Zugriff am: 26. 08. 2020) (siehe S. 1).

Bonomi, Flavio; Milito, Rodolfo; Zhu, Jiang u. a. (Aug. 2012): „Fog Computing and Its Role in the Internet of Things“. en. In: S. 3 (siehe S. 1, 6).

BSD Sockets — Zephyr Project Documentation (Aug. 2020). URL: https://docs.zephyrproject.org/latest/reference/networking/sockets.html (Zugriff am: 04. 08. 2020) (siehe S. 37).

Carrano, Juan (2018): *Basic networking with RIOT and Lua*. URL: https://github.com/riot-appstore/lua_demo (Zugriff am: 13. 07. 2020) (siehe S. 29, 32).

Carrano, Juan (2020): *Lua ported to RIOT*. URL: http://doc.riot-os.org/group__pkg__lua.html (Zugriff am: 22. 06. 2020) (siehe S. 21, 29).

Čolaković, Alem und Hadžialić, Mesud (Okt. 2018): „Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues“. en. In: *Computer Networks* 144, S. 17–39. ISSN: 13891286. DOI: 10.1016/j.comnet.2018.07.017. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128618305243 (Zugriff am: 04. 08. 2020) (siehe S. 1, 2).

Contiki 2.6: The Contiki ELF loader (Juli 2012). URL: http://contiki.sourceforge.net/docs/2.6/a01749.html (Zugriff am: 22. 06. 2020) (siehe S. 20).

Contiki NG Github Repository (Juni 2020). original-date: 2017-05-13T17:37:59Z. URL: https://github.com/contiki-ng/contiki-ng (Zugriff am: 22. 06. 2020) (siehe S. 20).

Dastjerdi, A.V. u. a. (2016): „Fog Computing: principles, architectures, and applications“. en. In: *Internet of Things*. Elsevier, S. 61–75. ISBN: 978-0-12-805395-9. DOI: 10.1016/B978-0-12-805395-9.00004-6. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780128053959000046 (Zugriff am: 19. 04. 2020) (siehe S. 9, 10).

East, Dan und Tilden, Scott (Nov. 2014): *Lua Mbed*. URL: https://os.mbed.com/forum/mbed/topic/5310/?page=1#comment-26507 (Zugriff am: 22. 06. 2020) (siehe S. 20).

FU Berlin (2020): *RIOT - The friendly Operating System for the Internet of Things*. URL: https://www.riot-os.org/#home (Zugriff am: 22. 06. 2020) (siehe S. 21).

George, Damien (2018): *MicroPython - Python for microcontrollers*. en. Library Catalog: micropython.org. URL: http://micropython.org/ (Zugriff am: 22. 06. 2020) (siehe S. 18).

Hießl, Thomas; Hochreiner, Christoph und Schulte, Stefan (Aug. 2019): „Towards a Framework for Data Stream Processing in the Fog“. en. In: *Informatik Spektrum* 42.4, S. 256–265. ISSN: 0170-6012, 1432-122X. DOI: 10.1007/s00287-019-01192-z. URL: http://link.springer.com/10.1007/s00287-019-01192-z (Zugriff am: 19. 04. 2020) (siehe S. 11).

IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing (2018). en. Techn. Ber. IEEE. DOI: 10.1109/IEEESTD.2018.8423800. URL: https://ieeexplore.ieee.org/document/8423800/ (Zugriff am: 13. 05. 2020) (siehe S. 6).

Ierusalimschy, Roberto (Nov. 2007): *lual mailing list*. URL: http://lua-users.org/lists/lua-l/2007-11/msg00248.html (Zugriff am: 22. 06. 2020) (siehe S. 18).

Al-khafajiy, Mohammed u. a. (Sep. 2018): „Fog Computing Framework for Internet of Things Applications“. en. In: *2018 11th International Conference on Developments in eSystems Engineering (DeSE)*. Cambridge, United Kingdom: IEEE, S. 71–77. ISBN: 978-1-5386-6712-5. DOI: 10.1109/DeSE.2018.00017. URL: https://ieeexplore.ieee.org/document/8648580/ (Zugriff am: 19. 04. 2020) (siehe S. 12).

Liu, Yang; Fieldsend, Jonathan E. und Min, Geyong (2017): „A Framework of Fog Computing: Architecture, Challenges, and Optimization“. en. In: *IEEE Access* 5, S. 25445–25454. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2766923. URL: https://ieeexplore.ieee.org/document/8085127/ (Zugriff am: 19. 04. 2020) (siehe S. 8).

Lua: FAQ (Juni 2020). URL: https://www.lua.org/faq.html#1.1 (Zugriff am: 04. 08. 2020) (siehe S. 35).

Lua: test suites (Juni 2020). URL: https://www.lua.org/tests/ (Zugriff am: 04. 08. 2020) (siehe S. 35).

Machine, The Carnegie Mellon University Computer Science Department Coke (2020): *The "Only"Coke Machine on the Internet*. URL: https://www.cs.cmu.edu/~coke/history_long.txt (Zugriff am: 19. 04. 2020) (siehe S. 4).

Marquesone, Rosangela de Fátima Pereira u. a. (2017): „Towards Bandwidth Optimization in Fog Computing using FACE Framework:“ en. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. Porto, Portugal: SCITEPRESS - Science und Technology Publications, S. 491–498. ISBN: 978-989-758-243-1. DOI: 10.5220/0006303804910498. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006303804910498 (Zugriff am: 27. 08. 2020) (siehe S. 1).

Minh, Quang Tran u. a. (2018): „FogFly: A Traffic Light Optimization Solution based on Fog Computing“. en. In: *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers - UbiComp '18*. Singapore, Singapore: ACM Press, S. 1130–1139. ISBN: 978-1-4503-5966-5. DOI: 10.1145/3267305.3274169. URL: http://dl.acm.org/citation.cfm?doid=3267305.3274169 (Zugriff am: 19. 04. 2020) (siehe S. 9).

Mulder, Patrick und Breseman, Kelsey (Okt. 2016): *Node.js for Embedded Systems*. en. O'Reilly Media, Inc. ISBN: 978-1-4919-2899-8 (siehe S. 18).

Munir, Arslan; Kansakar, Prasanna und Khan, Samee U. (Juli 2017): „IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things.“ en. In: *IEEE Consumer Electronics Magazine* 6.3, S. 74–82. ISSN: 2162-2248, 2162-2256. DOI: 10.1109/MCE.2017.2684981. URL: http://ieeexplore.ieee.org/document/7948854/ (Zugriff am: 19. 04. 2020) (siehe S. 8).

Native POSIX execution (native_posix) — Zephyr Project Documentation (2020). URL: https://docs.zephyrproject.org/latest/boards/posix/native_posix/doc/index.html (Zugriff am: 04. 08. 2020) (siehe S. 35).

O'Neill, Ryan (2016): *Learning Linux Binary Analysis*. en. Birmingham: Packt Publishing. ISBN: 978-1-78216-710-5 (siehe S. 17).

Overview - eluaproject (2011). URL: http://www.eluaproject.net/overview (Zugriff am: 22. 06. 2020) (siehe S. 18).

Patidar, Shyam; Rane, Dheeraj und Jain, Pritesh (Jan. 2012): „A Survey Paper on Cloud Computing“. en. In: *2012 Second International Conference on Advanced Computing & Communication Technologies*. Rohtak,Haryana, India: IEEE, S. 394–398. ISBN: 978-1-4673-0471-9 978-0-7695-4640-7. DOI: 10.1109/ACCT.2012.15. URL: http://ieeexplore.ieee.org/document/6168399/ (Zugriff am: 19. 04. 2020) (siehe S. 5).

Al-Rakhami, Mabrook u. a. (Aug. 2018): „Cost Efficient Edge Intelligence Framework Using Docker Containers“. en. In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. Athens: IEEE, S. 800–807. ISBN: 978-1-5386-7518-2. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00138. URL: https://ieeexplore.ieee.org/document/8511980/ (Zugriff am: 19. 04. 2020) (siehe S. 11).

SAM E70(B) Xplained — Zephyr Project Documentation (Aug. 2020). URL: https://docs.zephyrproject.org/latest/boards/arm/sam_e70_xplained/doc/index.html (Zugriff am: 04. 08. 2020) (siehe S. 37).

Shah, Sajjad Hussain und Yaqoob, Ilyas (Aug. 2016): „A survey: Internet of Things (IOT) technologies, applications and challenges“. en. In: *2016 IEEE Smart Energy Grid Engineering (SEGE)*. Oshawa, ON, Canada: IEEE, S. 381–385. ISBN: 978-1-5090-5111-3. DOI: 10.1109/SEGE.2016.7589556. URL: http://ieeexplore.ieee.org/document/7589556/ (Zugriff am: 30. 04. 2020) (siehe S. 4).

Sisinni, Emiliano u. a. (Nov. 2018): „Industrial Internet of Things: Challenges, Opportunities, and Directions“. en. In: *IEEE Transactions on Industrial Informatics* 14.11, S. 4724–4734. ISSN: 1551-3203, 1941-0050. DOI: 10.1109/TII.2018.2852491. URL: https://ieeexplore.ieee.org/document/8401919/ (Zugriff am: 13. 05. 2020) (siehe S. 4).

Snyder, James (2014): *Lua-RPC*. URL: https://github.com/jsnyder/luarpc (Zugriff am: 13. 07. 2020) (siehe S. 25).

ST Nucleo F746ZG — Zephyr Project Documentation (Aug. 2020). URL: https://docs.zephyrproject.org/latest/boards/arm/nucleo_f746zg/doc/index.html (Zugriff am: 04. 08. 2020) (siehe S. 38).

Steen, Maarten van und Tanenbaum, Andrew S. (2017): *Distributed systems*. en. Third edition (Version 3.01 (2017)). OCLC: 1006750554. London: Pearson Education. ISBN: 978-1-5430-5738-6 978-90-815406-2-9 (siehe S. 16).

Szydlo, Tomasz u. a. (Juni 2017): „Flow-Based Programming for IoT Leveraging Fog Computing“. en. In: *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. Poznan, Poland: IEEE, S. 74–79. ISBN: 978-1-5386-1759-5. DOI: 10.1109/WETICE.2017.17. URL: http://ieeexplore.ieee.org/document/8003792/ (Zugriff am: 19. 04. 2020) (siehe S. 9, 13, 18).

The Programming Language Lua (Juni 2020). URL: https://www.lua.org/about.html (Zugriff am: 22. 06. 2020) (siehe S. 18).

TIOBE Software BV (Juni 2020): *index | TIOBE - The Software Quality Company*. URL: https://www.tiobe.com/tiobe-index/ (Zugriff am: 22. 06. 2020) (siehe S. 18).

Yousefpour, Ashkan; Ishigaki, Genya und Jue, Jason P. (Juni 2017): „Fog Computing: Towards Minimizing Delay in the Internet of Things". en. In: *2017 IEEE International Conference on Edge Computing (EDGE)*. Honolulu, HI, USA: IEEE, S. 17–24. ISBN: 978-1-5386-2017-5. DOI: 10.1109/IEEE.EDGE.2017.12. URL: http://ieeexplore.ieee.org/document/8029252/ (Zugriff am: 19. 04. 2020) (siehe S. 5).

Zephyr Project (2020a): *Dynamic Module Loading · Issue #2746 · zephyrproject-rtos/zephyr*. en. Library Catalog: github.com. URL: https://github.com/zephyrproject-rtos/zephyr/issues/2746 (Zugriff am: 22. 06. 2020) (siehe S. 20).

Zephyr Project (2020b): *Zephyr Project | Home*. en-US. Library Catalog: www.zephyrproject.org. URL: https://www.zephyrproject.org/ (Zugriff am: 22. 06. 2020) (siehe S. 20).

Zou, Zhuo u. a. (März 2019): „Edge and Fog Computing Enabled AI for IoT -An Overview". en. In: *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. Hsinchu, Taiwan: IEEE. ISBN: 978-1-5386-7884-8 (siehe S. 8).

Anhang

A Codeausschnitte

In diesem Teil des Anhangs sind Codeausschnitte gelistet, die zum Erhalt des Leseflusses nicht im jeweiligen Kapitel der Arbeit gelistet wurden.

A.1 Szenario 1 - RIOT

```
-- BEGIN TESTS

enumerate();
print("iot1 Sensor1: "..execFct(iot1.readSensor1));
print("iot1 Sensor2: "..execFct(iot1.readSensor2));
print("iot2 Sensor1: "..execFct(iot2.readSensor1));
print("iot2 Sensor2: "..execFct(iot2.readSensor2));
execFct(iot1.writeSensor1, 50);
execFct(iot1.writeSensor2, 60);
execFct(iot2.writeSensor1, 55);
execFct(iot2.writeSensor2, 65);
print("iot1 Sensor1: "..execFct(iot1.readSensor1));
print("iot1 Sensor2: "..execFct(iot1.readSensor2));
print("iot2 Sensor1: "..execFct(iot2.readSensor1));
print("iot2 Sensor2: "..execFct(iot2.readSensor2));
```

Quellcode A.1: Szenario 1 Lua-Befehlsabfolge von „fog1“

```
-- BEGIN TESTS

enumerate();
local fogConnections = fog1.enumerate();
printConnections(fogConnections);
print("iot1 Sensor1: " .. fog1.iot1.readSensor1());
print("iot1 Sensor2: " .. fog1.iot1.readSensor2());
print("iot2 Sensor1: " .. fog1.iot2.readSensor1());
print("iot2 Sensor2: " .. fog1.iot2.readSensor2());
fog1.iot1.writeSensor1(80);
fog1.iot1.writeSensor2(90);
fog1.iot2.writeSensor1(85);
fog1.iot2.writeSensor2(95);
print("iot1 Sensor1: " .. fog1.iot1.readSensor1());
print("iot1 Sensor2: " .. fog1.iot1.readSensor2());
print("iot2 Sensor1: " .. fog1.iot2.readSensor1());
print("iot2 Sensor2: " .. fog1.iot2.readSensor2());
```

Quellcode A.2: Szenario 1 Lua-Befehlsabfolge von Host-Prozess

A.2 Szenario 2 - Zephyr

```
--connect to devices
fog1 = connect ("localhost",12349);
fogNotExisting = connect ("192.168.1.404",12345);

--run functions to evaluate
enumerate();
local fog1_con = fog1.enumerate();
printConnections(fog1_con);
local iot2_con = fog1.iot2.enumerate();
printConnections(iot2_con);
local iot3_info = iot2_con[1];
print("Value of 'iot3' sensor 1: " .. fog1.iot2.getIotSensorValue1()
    );
print("Value of 'iot3' sensor 2: " .. fog1.iot2.getIotSensorValue2()
    );
print("Value of 'iot1' sensor 1: " .. fog1.iot1.getSensorValue1());
print("Value of 'iot1' sensor 2: " .. fog1.iot1.getSensorValue2());
fog1.iot2.iot3.mirror = mirror;
print(fog1.iot2.iot3.mirror("Text and Number:" .. 13/7));

if iot3_info.type == 'temperature' then
    fog1.iot2.iot3.getTemperature = getTemperature;
else
    error_handler("Device type of " .. iot3_info.name .. " != '
        Temperature'");
end

close(fog1)
```

Quellcode A.3: Szenario 2 Lua-Befehlsabfolge von Host-Prozess

A.3 Szenario 3 - Kombination RIOT und Zephyr

```
fog1 = connect("/dev/pts/5",TRANSPORT_SERIAL);

enumerate();

local fog1_con = fog1.enumerate();
printConnections(fog1_con);

fog1.add = add;

print("Memory usage Host: "..collectgarbage("count"));
print("Memory usage fog1: "..fog1.collectgarbage("count"));
print("Memory usage iot1: "..fog1.iot1.collectgarbage("count"));
print("Memory usage iot2: "..fog1.iot2.collectgarbage("count"));
local res_fog = 0;
for i = 1, 100000 do
    res_fog =fog1.add(i,i);
```

```
17 end
18 print("Done running function on fog1");
19
20 print("Memory usage Host: "..collectgarbage("count"))
21 print("Memory usage fog1: "..fog1.collectgarbage("count"));
22 print("Memory usage iot1: "..fog1.iot1.collectgarbage("count"));
23 print("Memory usage iot2: "..fog1.iot2.collectgarbage("count"));
24 fog1.iot1.add = add;
25 fog1.iot2.add = add;
26 local res1 = 0;
27 local res2 = 0;
28 for i = 1, 100000 do
29     res1 = fog1.iot1.add(1,i);
30     res2 = fog1.iot2.add(i,i);
31 end
32 print("Done running function on IoT devices.");
33
34 print("Memory usage Host: "..collectgarbage("count"));
35 print("Memory usage fog1: "..fog1.collectgarbage("count"));
36 print("Memory usage iot1: "..fog1.iot1.collectgarbage("count"));
37 print("Memory usage iot2: "..fog1.iot2.collectgarbage("count"));
38
39 print("Running garbage collector...");
40 local collect_host = collectgarbage("collect");
41 local collect_fog1 = fog1.collectgarbage("collect");
42 fog1.iot1.collectgarbage("collect");
43 fog1.iot2.collectgarbage("collect");
44 while( collect_host ~= collectgarbage("collect")) do
45     collect_host = collectgarbage("collect");
46 end
47 while( collect_fog1 ~= fog1.collectgarbage("collect")) do
48     collect_fog1 = fog1.collectgarbage("collect");
49 end
50
51 --make sure there really is not anything left to garbage collect.
52 for i = 1, 10 do
53     collectgarbage("collect");
54     fog1.collectgarbage("collect");
55 end
56 print("Memory usage Host: "..collectgarbage("count"));
57 print("Memory usage fog1: "..fog1.collectgarbage("count"));
58 print("Memory usage iot1: "..fog1.iot1.collectgarbage("count"));
59 print("Memory usage iot2: "..fog1.iot2.collectgarbage("count"));
60
61 close(fog1,TRANSPORT_SERIAL);
```

Quellcode A.4: Szenario 3 Lua-Befehlsabfolge von Host-Prozess

Eidesstattliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Masterthesis selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 06.09.2020 Schwendinger Martin, BSc