Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Anton Dementyev

Adaptive Laufzeitumgebung für IoT Plattformen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Anton Dementyev

Adaptive Laufzeitumgebung für IoT Plattformen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Bettina Buth

Eingereicht am: 14. Juni 2018

Anton Dementyev

Thema der Arbeit

Adaptive Laufzeitumgebung für IoT Plattformen

Stichworte

Internet der Dinge (IoT), IoT Plattform, Adaptive Laufzeitumgebung

Kurzzusammenfassung

Das Internet der Dinge (engl. IoT) ist aktuell ein großes Forschungsgebiet. Analysten prognostizieren 50 Milliarden IoT-Geräte im Jahr 2020 [1], die ihre Anwendung in vielen verschiedenen Bereichen, zum Beispiel dem Smart Home, der Smart City, dem Gesundheitswesen und vielem mehr finden [2].

Gegenwärtige Open-Source IoT-Plattformen bieten viele Dienste wie Auswertung, Darstellung und Austausch der Daten. Dabei decken sie aber nicht die Interoperabilität der installierten Software zwischen den einzelnen IoT-Geräten ab. Einer der Gründe dafür ist die Individualität der Software, die oft mit verschiedenen Programmiersprachen geschrieben ist. Obwohl Hersteller versuchen, die Geräte maximal benutzerfreundlich zu machen, steht der Anwender vor mehreren Problemen. Der Nutzer verliert leicht die Übersicht, denn das Update oder die Anpassung der bestehenden Software jedes einzelnen IoT-Geräts muss individuell durchgeführt werden. Eine passende Software hängt nicht nur von der benutzten Plattform oder vom konkreten Typ des IoT-Geräts ab, sondern auch von seiner Version und Faktoren, wie z.B. dem Ort oder der Sprache. In diesem Kontext stellt sich die Frage, wie diese Probleme gelöst werden können und wie ein Software-Update-Mechanismus unabhängig von der individuellen Hardware aussehen kann.

Diese Bachelorthesis befasst sich mit der Implementierung eines Prototyps einer adaptiven Laufzeitumgebung für IoT-Plattformen. Der Prototyp unterstützt durch sein Design die Interoperabilität der Software für möglichst viele unterschiedliche Hardware. Die Laufzeitumgebung ermöglicht eine dynamische Installation des auszuführenden Codes auf den IoT-Geräten, seine Validierung und die anschließende Ausführung mit einem Rollback-Mechanismus. Für die angebotenen Lösungskonzepte wird eine Beschreibungssprache vorgestellt, die zur Erstellung von Software-Schnittstellen verwendet wird. Zusätzlich wird die Software aus Sicherheitsgründen mittels Zertifikaten signiert, damit eine sichere Authentifizierung der Software-Quelle angeboten wird.

Anton Dementyev

Title of the paper

Adaptive runtime environment for IoT platforms

Keywords

Internet of Things (IoT), IoT platform, Adaptive runtime environment

Abstract

The Internet of Things (IoT) is currently a large research area. Analysts forecast 50 billion IoT devices in 2020 [1] in many different areas, such as the smart home, the smart city, the healthcare and many more [2].

Current open-source IoT platforms offer many services for evaluation, presentation and exchange of data. However, they don't cover the interoperability of the installed software between the individual IoT devices within a platform. One of the reasons is the individuality of the installed software that is often written with different programming languages. Although manufacturers try to make devices as user-friendly as possible, users face several problems. The users easily lose the overview, because the update or adjustment of the existing software of IoT devices must be performed individually. Suitable software depends not only on the IoT platform or the specific type of the IoT device, but also on its version, the location or the language. So it has an issue how these problems can be solved and how a software update mechanism can be independent of the individual hardware.

This bachelor thesis deals with the implementation of a prototype of an adaptive runtime environment for IoT platforms. Its design supports the interoperability of the software for as many different hardware as possible. The proposed environment allows the dynamic installation of the code on the IoT devices, its validation and execution with a rollback mechanism. The offered concept presents a generic description language, which is used for the creation of software interfaces. For security reasons, the code is also signed using certificates, thus providing a secure authentication of the software source within the network.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1 Einleitung

1.1 Motivation

Momentan haben IoT-Geräte - miteinander vernetzte Gegenstände, die durch ihre Informations- und Kommunikationstechniken zusammenarbeiten [3] - sich sehr verbreitet. Die Fähigkeit, auch Geräte mit einer relativ schwachen CPU- oder Speicherleistung mit einem Netz verbinden zu können, ermöglicht dem Internet der Dinge, seine Anwendung in vielen verschiedenen Bereichen zu finden, wie zum Beispiel dem Smart Home, der Smart City, dem Gesundheitswesen [2].

Die Wissenschaftsgemeinde beschäftigt sich aktiv mit der Forschung des Internets der Dinge. Dessen rasante Entwicklung wirft immer neue Fragen auf. Unter vielen verschiedenen Entwicklungsrichtungen finden zurzeit die folgenden Fachgebiete besondere Beachtung [4][5]:

- Es fehlt an großen, potenziellen Kunden. Zwar werden über 10 Millionen Sensoren in Geschäften, Häusern, Autos, Stromnetzen etc. eingesetzt, doch davon sind viele nicht mit dem Internet verbunden. Solange das Umsatzpotenzial nicht klar ist, kommen die für diese Verbindungen abzuschließenden Investitionen nicht zustande.
- Sicherheitsfragen müssen geklärt werden. Angeschlossene Gerätesysteme können angegriffen werden und damit den Zugriff auf hochsensible Daten ermöglichen.
- Software muss definiert werden. Es gibt noch zahlreiche Software-, Systemintegrations- und Bearbeitungsprobleme zu lösen. Beispielsweise die Fragen, was genau die Middleware tun sollte, oder ob die Performanceanalyse besser im Gerät selbst oder im Netzwerk durchgeführt wird. Eine standardisierte Middleware wird dringend benötigt, doch von einem Konsens ist man noch weit entfernt.

- Es mangelt an Standards. Über 400 gängige Standards gibt es – was eigentlich bedeutet, dass es keine wirklichen Standards gibt. Entwickler müssen also davon überzeugt werden, ein einheitliches Kommunikationsprotokoll für Daten zu nutzen, die dann in die Geräte eingebettet werden, ansonsten entstehen nur Daten-Silos mit eigenen, nicht ersichtlichen Protokollen. Unternehmen müssen somit anfangen zu verstehen, dass Daten zu teilen mehr bringt, als Daten zu horten.

Das fehlende einheitliche Kommunikationsprotokoll sowie die Individualität der Software jedes IoT-Geräts verursachen weitere Probleme. Die gegenwärtigen IoT-Plattformen decken nicht die Interoperabilität der installierten Software zwischen den einzelnen IoT-Geräten ab. Die IoT-Plattformen bieten zwar viele komplizierte Dienste für die Auswertung und Austausch der Daten an, verfügen jedoch nicht über eine hinreichende und maximal einfache Funktionalität für ein Update oder eine Anpassung der Software jedes einzigen IoT-Geräts. Der Nutzer verliert dabei Ordnung und Übersicht, denn eine passende Software hängt nicht nur von der benutzten Plattform und dem konkreten Typ des IoT-Geräts, sondern auch von weiteren Faktoren wie z.B. seiner Version oder dem Ort ab.

Für die Lösung dieses Problems wird eine adaptive Laufzeitumgebung für IoT-Plattformen vorgeschlagen, die einen einfachen und von einer individuellen Hardware unabhängigen Software-Update-Mechanismus ermöglicht.

1.2 Zielsetzung und Anforderungen an das System

Das Ziel dieser Bachelorthesis ist zu zeigen, wie das Design eines Systems, das die Interoperabilität der Software für möglichst viele unterschiedliche IoT-Geräte unterstützt, aussehen könnte. Um dieses Problem zu diskutieren, soll ein Prototyp entwickelt werden, der die folgenden Anforderungen erfüllt:

- **Interoperabilität**. IoT-Geräte werden von verschiedenen Herstellern entwickelt und hergestellt. Sie benutzten verschiedene Standards und Programmiersprachen und bieten ihren Nutzern verschiedene Dienste an. Diese Gründe führen zur Individualität der instal-

lierten Software und sind deswegen inkompatibel mit IoT-Geräten anderer Art. Die zu entwickelnde Laufzeitumgebung muss die Interoperabilität der Software unterstützen. Es muss möglich sein, dieselbe Software an unterschiedlichen IoT-Geräten auszuführen.

- **Dynamik**. Viele Anwendungsszenarien der IoT-Geräte erfordern die Ununterbrochenheit. Fahrende Autos, medizinische Geräte müssen während der Updates ihre Dienste ununterbrochen weiter erfüllen, sonst kann es zu tragischen Konsequenzen führen. Die Updates für IoT-Geräte müssen zur Laufzeit dynamisch durchgeführt werden. Die Installation einer neuen Software darf die aktuell laufende Software nicht unterbrechen.
- **Sicherheit.** Die Sicherheit stellt heutzutage eine der größten Anforderungen an die angeschlossenen IoT-Geräte. Die IoT-Geräte können durch Software-Aktualisierungen angegriffen werden und als Folge ihre Daten verändert oder Operationen manipuliert werden. Das System muss in der Lage sein, die zu installierende Software von autorisierten Herstellern zu verifizieren. Wenn die Software verschlüsselt werden soll, muss es auf einer Weise erfolgen, dass das Zielgerät diese entschlüsseln kann.
- **Robustheit.** Ein Fehler während der Installation oder Ausführung eines Updates darf zu keinem Zeitpunkt zu einem Ausfall des Gerätes führen. Um dies zu erreichen, muss das Gerät mindestens zwei Speicherplätze bereitstellen - einen, um die Software zu speichern, und einen, um die Software auszuführen.
- **Fail-Safe.** Die Umgebung muss immer einen arbeitsfähigen Zustand bieten. Schlägt die installierte Software während der Ausführung fehl, muss ein Rollback-Mechanismus ausgeführt werden, in dem eine der vorherigen und arbeitsfähigen Versionen der Software ausgeführt wird. Damit kann der Benutzer mit dem System immer interagieren.
- **Atomarität.** Um die unerwarteten Fehler während der Laufzeit zu vermeiden, muss ein Software Update komplett oder gar nicht installiert werden.

- **Administration**. Der IoT-Bereich findet seine Anwendung in vielen verschiedenen Bereichen. IoT-Geräte werden von Leuten mit unterschiedlichen Fachkenntnissen benutzt. Die zu entwickelnde Laufzeitumgebung muss einfach einzurichten und zu administrieren sein.

1.3 Aufbau der Arbeit

Die Bachelorthesis ist wie folgt aufgebaut:

- Kapitel 2 – Verwandte Arbeiten – stellt verwandte Arbeiten vor und präsentiert existierende IoT-Plattformen und Lösungen für Software-Update-Architekturen.
- Kapitel 3 – Wahl des Technologie-Stacks – beschäftigt sich mit der Analyse vorhandener Technologien, um anschließend diejenigen auszuwählen, die für die Erfüllung der definierten Anforderungen am besten geeignet sind.
- Kapitel 4 – Design und Architektur – beschreibt die Festlegung der Systemarchitektur, Beziehungen zwischen ihren Komponenten und die wichtigsten Systemsinteraktionen.
- Kapitel 5 – Implementierung eines Prototyps – dokumentiert die konkrete Umsetzung der Laufzeitumgebung in der Form eines Prototyps.
- Kapitel 6 – Experimente – beschreibt die durchgeführten Experimente und präsentiert ihre Ergebnisse.
- Kapitel 7 – Schlussfolgerungen – fasst Forschungsergebnisse und mögliche zukünftige Entwicklungswege der vorgestellten Laufzeitumgebung zusammen.

2 Verwandte Arbeiten

Um zu analysieren, wie die Interoperabilität der installierten Software zwischen den einzelnen Geräten in IoT-Plattformen organisiert werden kann, werden in diesem Kapitel aktuelle IoT-Plattformen und bereits existierende Arbeiten und Lösungen für Software-Updates für IoT-Geräte vorgestellt und verglichen. Aus der großen Anzahl der bestehenden Systeme werden einige ausgewählt, die unterschiedliche Ansätze präsentieren.

2.1 IoT Plattformen

Die Erforschung der Integrationsplattformen hat mit der Entwicklung des Internets der Dinge gleichzeitig begonnen. Der Grund dafür ist, dass die IoT-Konzepte eine Dienstkomposition der Geräte und deren Kontrolle erfordern. Eines der wichtigsten Probleme bei der Entwicklung einer IoT-Plattform ist die Skalierbarkeit im Netz [6]. Ein privates Netz kann aus dutzenden Geräten bestehen, ein verteiltes Netz im Maßstab einer Stadt aus mehreren tausenden. Um die Steuerung der Ressourcen zentralisiert zu organisieren, wird oft ein Cloud Computing Ansatz verwendet – die Bereitstellung von IT-Infrastrukturen, die netzbasiert genutzt werden können [7]. Im Rahmen dieser Arbeit werden zwei Cloud-Computing-Plattformen verglichen: IoT Hub von Microsoft Azure und AWS IoT von Amazon.

2.1.1 Microsoft Azure IoT Hub

IoT Hub ist ein vollständig verwalteter Dienst von Microsoft Azure, der eine zuverlässige und sichere bidirektionale Kommunikation zwischen IoT-Geräten und einem Lösungs-Back-End ermöglicht [8].

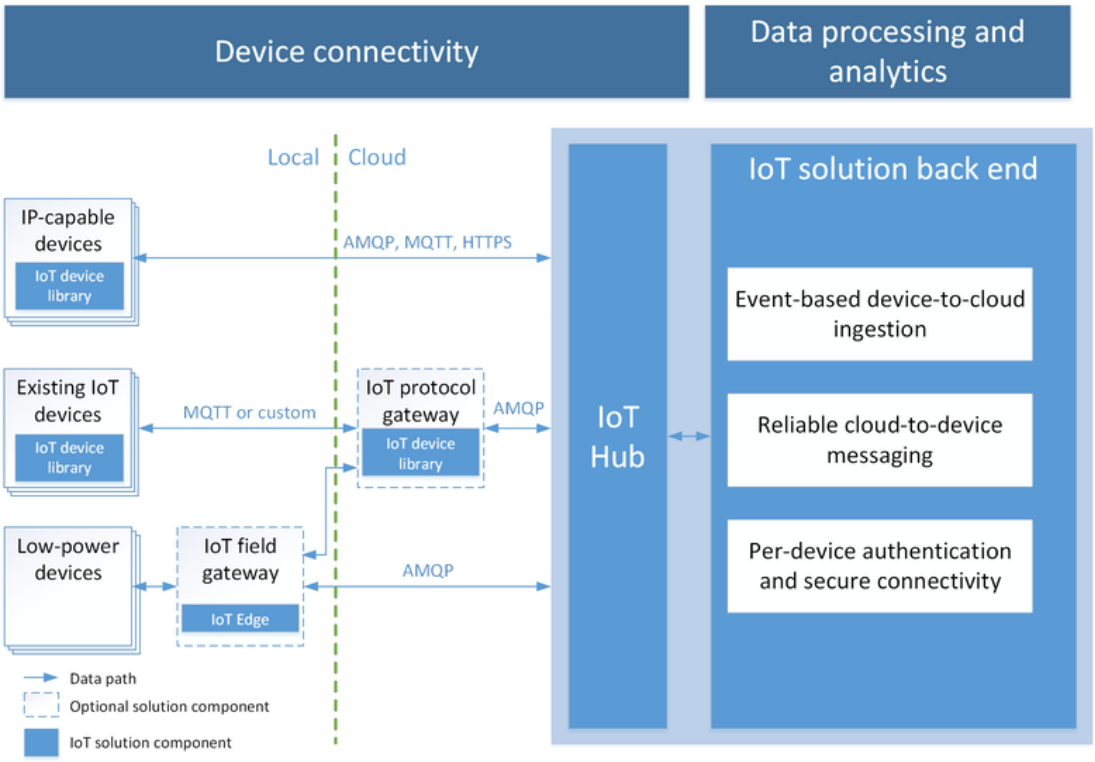

Abbildung 2.1: IoT Hub Architektur [8]

Aufgrund der bidirektionalen Natur findet das Nachrichtenrouting zwischen Geräten und der Cloud in beiden Richtungen entlang des etablierten Kanals statt. Jedes Gerät verfügt über zwei Endpunkte für die Interaktion mit IoT Hub [9]:

- Device-to-Cloud – Geräte verwenden diesen Endpunkt, um Nachrichten und Telemetriedaten zur Speicherung und Verarbeitung in die Cloud zu senden.
- Cloud-to-Device – dieser Endpunkt wird von Geräten verwendet, um Cloud-zu-Gerät-Nachrichten zu empfangen und darauf zu reagieren ggf. die angeforderten Aufgaben auszuführen.

Zum Herstellen von sicheren und zuverlässigen Verbindungen zwischen Geräten erlaubt IoT Hub für jedes Gerät, einen eigenen Sicherheitsschlüssel bereitzustellen. Die IoT Hub Identitätsregistrierung kann „einzelne Geräte ei-

ner Positiv- oder Negativliste hinzufügen und so die vollständige Kontrolle über den Gerätezugriff ermöglichen“ [8].

Azure IoT Hub bietet einen abfragbaren Speicher für Gerätemetadaten und synchronisierte Zustandsinformationen und eine umfassende Überwachung der Ereignisse zur Verwaltung der Gerätekonnektivität und –identität [8].

Microsoft Azure stellt einen umfassenden Satz von Gerätebibliotheken zur Verfügung: „Azure IoT-Geräte-SDKs sind für die unterschiedlichsten Sprachen und Plattformen verfügbar und werden entsprechend unterstützt (C für viele Linux-Distributionen, Windows und Echtzeitbetriebssysteme). Azure IoT-Geräte-SDKs unterstützen auch verwaltete Sprachen wie C#, Java und JavaScript“ [8].

Azure IoT Hub unterstützt nativ die Kommunikation über die Protokolle MQTT, AMQP und HTTPS. Außerdem ermöglicht IoT Hub eine Protokollanpassung für IoT Hub-Endpunkte und unterstützt damit auch benutzerdefinierte Gateways [10].

Für die Geräteverwaltung sowie die Änderung der Konfiguration und Software- und Firmwareupdates bietet Microsoft Azure das IoT Hub Device Management. Ein Firmwareupdate besteht dabei aus mehreren Schritten: „das Firmwareimage herunterzuladen, das Firmwareimage anzuwenden und schließlich wieder eine Verbindung mit dem IoT Hub-Dienst herzustellen“ [11].

2.1.2 Amazon AWS IoT Core

Amazon bietet eine weitere verwaltete IoT Cloud-Plattform - AWS IoT Core. Wie Azure IoT Hub verfolgt AWS IoT Core den gleichen Zweck: eine einfache Verbindung von Geräten mit der Cloud und mit anderen Geräten. Allerdings verwendet Amazon einen anderen Ansatz, um dies zu erreichen.

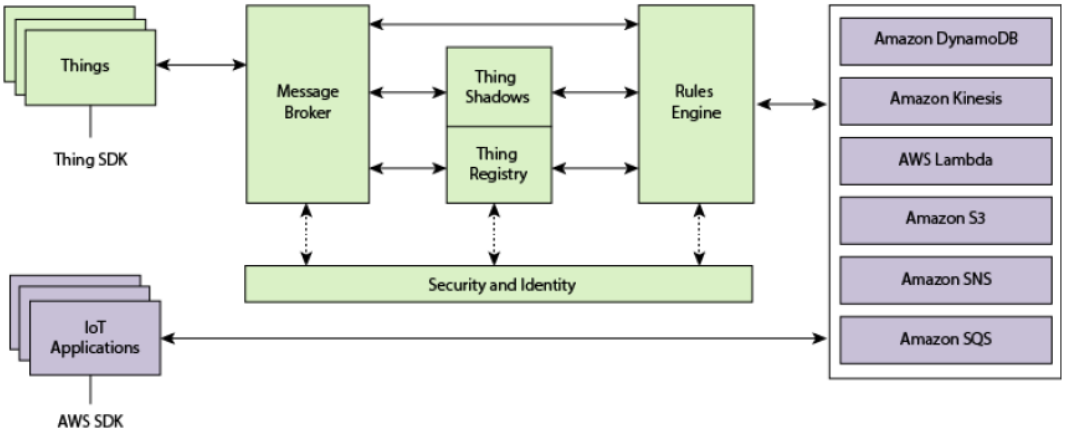

Abbildung 2.2: AWS IoT Core Architektur [12]

Eins der wichtigsten Elemente in der AWS IoT Core Architektur ist ein Pub/Sub-Message Broker. Der Message Broker ermöglicht mit geringer Latenz sicher Nachrichten an beliebig viele Geräte zu senden und diese von beliebig vielen Geräten zu empfangen. Der Message Broker ist automatisch skalierbar gemäß Nachrichtenvolumen und unterstützt Nachrichtenmuster von Eins-zu-Eins-Befehls- und Kontrollmeldungen bis hin zu eins zu einer Million [13].

AWS IoT Core unterstützt HTTP, WebSockets und MQTT (Message Queue Telemetry Transport Protokoll) [14]. MQTT ist ein leichtes und einfaches Client-Server publish-subscribe Nachrichtenprotokoll, das trotz hoher Verzögerungen oder beschränkter Netzwerke die Nachrichtenübertragung für Machine-to-Machine-Kommunikation ermöglicht [15]. Dadurch werden im AWS IoT Core Plattform die Anforderungen an die Netzwerkbandbreite reduziert und der Code auf Geräten minimisiert. Außerdem werden in der Plattform benutzerdefinierte Protokolle unterstützt [14].

Das Zentrale Konzept der Plattform ist der Status der Geräte. Geräte sind in der Lage, ihren Status in Nachrichten an den Message Broker über die Topic-Kommunikationskanäle zu senden. Der Status jedes Gerätes ist mit einem so genannten Schattengerät verbunden. Das Schattengerät enthält den zuletzt angemeldeten Status eines Gerätes und ermöglicht den Anwendungen oder anderen Geräten, Nachrichten zu lesen und mit dem Gerät zu interagieren, auch wenn das Gerät offline ist [13].

Bei Authentifizierungen und Verschlüsslungen verwendet AWS IoT Core die SigV4 AWS-Methode, die Authentifizierung auf Basis des X.509-Zertifikats oder auf Basis eines vom Kunden erstellten Tokens. Dadurch werden Daten nie ohne geprüfte Identität zwischen Geräten und AWS IoT Core ausgetauscht [13].

Wie in Azure IoT Hub verfügt AWS IoT Core über eine Registry, die eine Identität für Geräte erstellt und Metadaten wie die Attribute und Fähigkeiten der Geräte verfolgt [13].

Um eine einfache und schnelle Verbindung eines Hardwaregeräts oder einer mobilen Anwendung mit AWS IoT Core zu ermöglichen, unterstützt das AWS IoT Device SDK JavaScript, C und Arduino sowie bietet zudem Open Source-Alternativen [13].

Für eine unkomplizierte Organisation von Geräten bietet Amazon ein Device Management. AWS IoT Device Management ermöglicht über Fernzugriff die Software auf Geräten zu aktualisieren, Geräte auf die Werkanstellungen zurückzusetzten und Sicherheitspatches auszuführen [16]. Dadurch kann sichergestellt werden, dass auf den Geräten stets die neueste Software ausgeführt wird und die Geräte sicher sind.

2.1.3 Diskussion

Die Verwaltung von Millionen Geräten und Nachrichten sowie deren Überwachung und Fernsteuerung sind die Ziele der dargestellten IoT-Plattformen. Beide Plattformen verfügen über Device Management Systeme, mit denen Softwareupdates an den IoT-Geräten durchgeführt werden können. Obwohl die Device Management Systeme der Plattformen den Softwareupdate-Mechanismus erleichtern, bieten sie keine SoftwareInteroperabilität zwischen den einzelnen Geräten.

Microsoft und Amazon entwickeln ihre Plattformen mit unterschiedlichen Ansätzen, aber gehen die gleichen wichtigen Punkte an. Diese Merkmale, die auch im Kapitel 1.2 der Arbeit als Anforderungen an die zu entwickelnde Laufzeitumgebung definiert sind, sind in der Vergleichstabelle 2.1 zu finden:

Plattform	Microsoft Azure IoT Hub	Amazon AWS IoT Core
Interoperabilität der Software	Nein	Nein
Kommunikations-protokolle	MQTT, AMQP, HTTPS, Benutzerdefinierten Protokolle	HTTP, MQTT
SDK / Sprachen	C, C#, Java, JavaScript	JavaScript, C
Sicherheit / Authentifizierung	Sicherheitsschlüssel pro Gerät	SigV4 AWS-Methode, X.509-Zertifikat, Benutzerdefinierten Tokens

Tabelle 2.1: Vergleich Microsoft Azure IoT Hub und Amazon AWS IoT Core Plattformen

2.2 Software-Update

Das Aktualisieren der Software spielt eine wichtige Rolle im Lebenszyklus eines IoT-Geräts. Für die Notwendigkeit, eine bestehende Software zu aktualisieren, gibt es unterschiedliche Gründe. Beispielswieser kann es die Behebung eines Fehlers, das Hinzufügen einer neuen Funktionalität oder eine neue Konfiguration des Geräts sein. Nicht sicher geführte Aktualisierungsmechanismen können für Sicherheitsverletzungen anfällig sein. Aktualisierungen der Software können für Angriffe ausgenutzt werden, um die Gerätedaten oder -operationen zu manipulieren. Aus diesem Grund müssen bei den Aktualisierungsprozessen von Geräten Sicherheitsmaßnahmen verwendet werden [17].

Die IETF (Internet Engineering Task Force) Organisation befasst sich mit der technischen Weiterentwicklung des Internets [18]. Zur großen Datenbank der veröffentlichten technischen Dokumente gehören auch Vorschläge mit Lösungen für Software-Update-Architekturen.

2.2.1 Software-Update-Architektur. Beispiellösung 1

Eine Architekturbeschreibung für eine sichere Firmware-Aktualisierung wurde von Brendan Moran im Januar 2018 veröffentlicht [19]. Moran schlägt die folgende End-to-End Architektur vor:

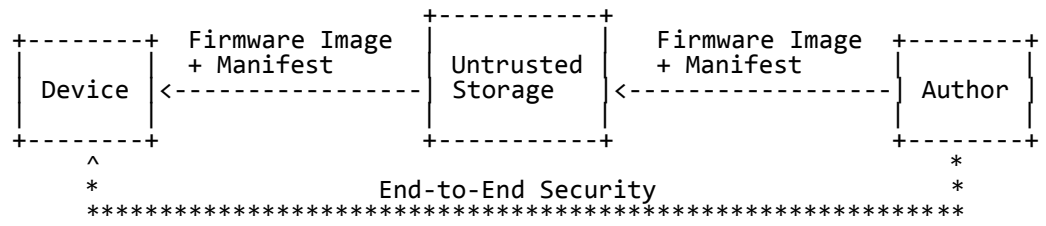

Abbildung 2.3: End-to-End Sicherheitsarchitektur [19]

Der Hersteller der Firmware (Autor) erstellt eine neue Software für ein IoT-Gerät (Firmware-Image) und die entsprechende Meta-Data Information (Manifest). Das Firmware-Image kann verschlüsselt sein und ist integritätsgeschützt. Die Metadaten sind auch integritätsgeschützt. Wenn der Autor bereit ist, das Firmware-Image zu verteilen, übermittelt er es über einen bevorzugten Kommunikationskanal an das Gerät. Dies erfordert normalerweise die Verwendung eines nicht vertrauenswürdigen Dateiservers (Untrusted Storage).

Für das Akzeptieren des Updates muss ein Gerät entscheiden, ob der Autor, der das Firmware-Image und das Manifest signiert hat, berechtigt ist, die Aktualisierungen vorzunehmen. Dafür wird eine Kryptographie mit dem öffentlichen Schlüssel verwendet. Das Gerät muss das Zertifikat bzw. den öffentlichen Schlüssel des Autors enthalten. Nachdem das Gerät das Manifest validiert und die Firmware verifiziert hat, wird die Firmware am Gerät gespeichert und das Gerät rebootet. Beim Neustart validiert der Bootloader des Gerätes die neue installierte Firmware, aktiviert sie und übergibt ihr die Kontrolle.

2.2.2 Software-Update-Architektur. Beispiellösung 2

Eine weitere Architekturlösung für sichere Software-Updates [20] lässt sich mit der folgenden Sequenz beschreiben:

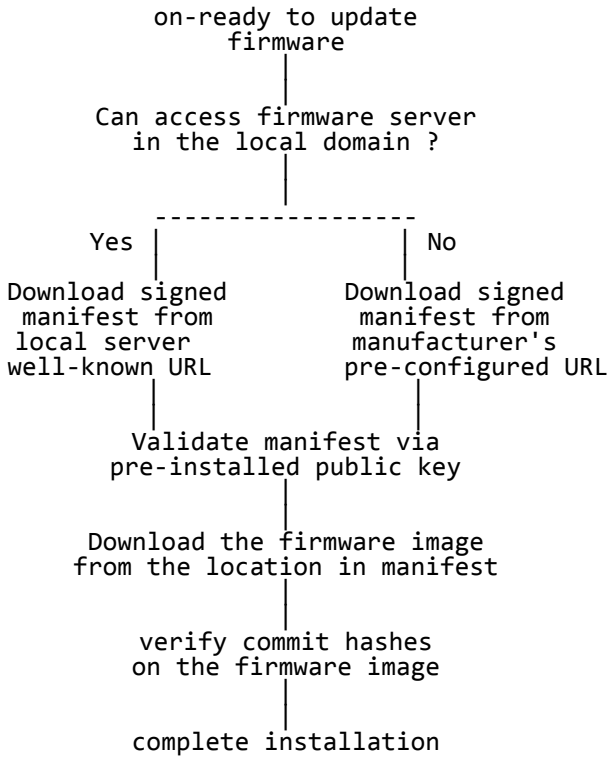

Abbildung 2.4: Erfolgreiches Firmware-Update [20]

Wie in der Lösung 2.2.1 dargestellt, wird hier für die Akzeptierung eines Firmware-Images auch ein Manifest verwendet, das die Metadaten über die Firmware repräsentiert. Das Manifest wird vom Firmware-Hersteller mit seinem privaten Schlüssel kryptografisch signiert. Nach der erfolgreichen Validierung des Manifestes überprüft das Gerät die Commit-Hashes des Firmware-Images und führt die weitere Installation fort.

Für die Durchführung eines Firmware-Updates muss das IoT-Gerät den Standort des Firmware-Images bestimmen. Dafür versucht das Gerät, den Ser-

ver erst im lokalen Domain zu entdecken. Die Server-URL wird automatisch mit Hilfe von DHCP (Dynamic Host Configuration Protocol) generiert. Wenn der lokale Server nicht erreichbar ist, lädt das Gerät das Manifest und die Firmware von der anderen URL runter, die vom Hersteller des Geräts vorkonfiguriert ist. Wenn dies auch nicht möglich ist (z.B. es gibt keinen Internetzugang), wird das Gerät entweder unbenutzt oder wird eine ältere Version von der Firmware ausgeführt.

2.2.3 Diskussion

Sicherheitslücken in IoT-Geräten haben den Bedarf an einem sicheren Firmware-Update-Mechanismus. Sicherheitsexperten und Forscher empfehlen, dass alle IoT-Geräte mit einem solchen Mechanismus ausgestattet werden. Während heutzutage viele Firmware-Update-Mechanismen im Einsatz sind, gibt es keinen modernen interoperablen Ansatz, der sichere Aktualisierungen von Firmware in IoT-Geräten ermöglicht [21].

Die präsentierten Ansätze definieren keine neuen Transport- oder Discoverymechanismen. Sie beschreiben, wie bereits existierende Mechanismen für eine sichere Software-Update-Architektur genutzt werden können. Die wichtigen Komponenten der präsentierten Architekturlösungen für Software-Aktualisierungen sind

- ein Mechanismus, um Software-Images zu kompatiblen Geräten zu transportieren;
- ein Manifest, um die Metadaten über die zu installierende Software zu beschreiben. Solche Metadaten sind typescherweise ein Software-Identifier, eine passende Hardware, Abhängigkeiten, kryptographische Information zum Schutz des Images;
- das Software-Update-Image selbst.

3 Wahl des Technologie-Stacks

Zusätzlich zu den vorliegenden technischen und fachlichen Anforderungen sind am Anfang der Implementierung weitere Vorbereitungen nötig. Zu den wichtigsten Aufgaben zu Beginn der Softwareentwicklung zählt die Auswahl des passenden Technologie-Stacks. Eine besondere Herausforderung dieser Phase liegt darin, dass die am Anfang der Entwicklung ausgewählten Plattformen und Bibliotheken eine Basis bilden müssen. Auf der Basis lässt sich dann das gesamte System entwickeln. Eine unangemessen gewählte Technologiebasis kann die weitere Entwicklung des Systems, seine Erweiterbarkeit und Integration mit anderen Diensten erschweren.

Dieses Kapitel beschreibt die Auswahl des Technologie-Stacks, das für die weitere Implementierung eines Prototyps der Laufzeitumgebung angewendet wird. Die Analyse und Bestimmung der geeigneten Technologien basieren sich auf den im Unterkapitel 1.2 definierten Zielen und Anforderungen. Dabei werden Erfahrungen der im Kapitel 2 präsentierten verwandten Arbeiten berücksichtigt.

3.1 Programmiersprache

Grundlegend lassen sich die Programmiersprachen in zwei verschiedene Kategorien einteilen, und zwar solche die kompiliert und solche die interpretiert werden. Diese Unterscheidung ist allerdings vage, da sich einige Sprachen unter Verwendung beider Techniken implementieren lassen.

3.1.1 Kompilierung

Die Kompilierung bedeutet, dass der Quellcode einer bestimmten Programmiersprache zunächst in einen Maschinencode übersetzt wird. Das spezielle Computerprogramm, mit dem die Übersetzung stattfindet, wird Compiler genannt. Als Ergebnis der Übersetzung entsteht ein ausführbares Modul, das von einem Computer direkt ausgeführt werden kann [22].

In der Regel arbeiten die kompilierten Programme schneller und benötigen während der Ausführung keine zusätzlichen Programme, da sie bereits in den Maschinencode übersetzt sind. Allerdings ist es notwendig, nach jeder Änderung des Programmtextes den Code neu zu kompilieren, was den Entwicklungsprozess verlangsamen kann. Außerdem kann ein kompiliertes Programm nur auf Computern des gleichen Typs und mit dem gleichen Betriebssystem ausgeführt werden, für die der Compiler entwickelt worden ist. Um eine ausführbare Datei für einen anderen Maschinentyp zu erstellen, ist eine neue Kompilierung mit einem zu diesem System passenden Compiler erforderlich.

3.1.2 Interpretation

Die Interpretation bedeutet, dass Befehle des Quellcodes direkt ausgeführt werden, ohne sie zuerst durch einen Compiler in maschinensprachliche Anweisungen zu übersetzen. Die Anweisungen des Quellcodes werden dabei Schritt für Schritt mit einem speziellen Programm (Interpreter) abgearbeitet [23].

Die interpretierten Sprachen können mit der Ausführung eines Programms direkt oder unmittelbar nach Veränderungen beginnen, auch auf verschiedenen Arten von Maschinen und Betriebssystemen. Somit wird die Portabilität gewährleistet und die dynamische Ausführung der Programme ermöglicht, ohne dass diese ihre Arbeit unterbrechen müssen. Die Portabilität eines interpretierten Programms wird nur durch die Verfügbarkeit der Interpreter für gezielte Hardwareplattformen bestimmt. Der Preis für diese Vorteile sind allerdings oft Perfomanceprobleme, da die zusätzliche Interpreter-Schicht die Ausführung verlangsamt. Außerdem, wenn ein Programm einen Fehler enthält, bleibt dies unerkannt, bis der Interpreter die fehlerhafte Stelle im Code erreicht. Viele der interpretierenden Programmiersprachen bieten zusätzliche Funktionen wie dynamische Codegenerierung an. Somit kann ihre Kompilation dynamisch und während der Laufzeit ausgeführt werden (Just-in-time compilation) [24].

3.1.3 Diskussion

Wichtige Anforderungen an die zu entwickelnde Laufzeitumgebung sind die Interoperabilität und Dynamik. Es muss möglich sein, dieselbe Software auf unterschiedlichen IoT-Geräten in Laufzeit zu installieren, auszuführen und auszutauschen. Aus diesem Grund wird für die Implementierung eine interpretierende Programmiersprache gewählt. Die dabei entstehende Gefahr, bestimmte Fehler nur während der Laufzeit zu entdecken, soll durch eine sichere Software-Architektur begegnet werden.

3.1.4 JavaScript

Wie im vorherigen Abschnitt erläutert, muss für die Implementierung eine interpretierte Programmiersprache benutzt werden. Dadurch wird die Möglichkeit der dynamischen Codeausführung realisiert. Auch muss die Sprache die Plattformunabhängigkeit (cross-platform) anbieten. So wird es möglich, dieselbe Version der Laufzeitumgebung auf möglichst vielen IoT-Plattformen zu installieren und auszuführen. Darüber hinaus muss die Laufzeitumgebung in der Zukunft auch in die existierenden IoT-Plattformen eingebettet werden.

Die im Kapitel 2 diskutierten IoT-Plattformen bieten ihren Nutzern für die Verwaltung der Geräte die RESTful-API Schnittstellen an. Eigene Implementierung dieser Schnittstellen erfordert einen zusätzlichen Aufwand. Um die zukünftige Einbettung der Laufzeitumgebung wesentlich zu erleichtern, ist es sinnvoll, die gleichen Werkzeuge zu nutzen, die zurzeit von den existierenden IoT-Plattformen in den SDKs unterstützt werden. Als eine Schnittmenge dieser Anforderungen ergibt sich JavaScript.

JavaScript ist eine interpretierte vollständige Programmiersprache [25], die ihre Anwendung auch im IoT-Bereich findet [26]. Dank der Node.js-Plattform wird JavaScript von engspezialisiert in eine Sprache mit einem sehr breiten Anwendungsspektrum umgewandelt [27]. Wegen der Erfüllung der oben gesetzten Anforderungen wird für die weitere Implementierung eines Prototyps JavaScript unter Verwendung von Node.js benutzt.

3.2 Authentifizierung

Die Sicherheit stellt heutzutage eine der größten Anforderungen an die angeschlossenen IoT-Geräte. Nicht genug gesicherte Geräte und Netzwerke bieten Angreifern Möglichkeiten, Informationen zu manipulieren oder Geräte für andere, kriminelle Zwecke zu missbrauchen. Software-Aktualisierungen öffnen eine weitere Sicherheitslücke, durch welche IoT-Geräte angegriffen werden können. Aus diesem Grund muss jedes Software-Image vor seiner Installation verifiziert werden.

Der Prozess der Bindung einer Identität an ein Objekt heißt die Authentifizierung [28]. Um die Authentizität einer behaupteten Eigenschaft nachzuweisen, muss die Entität bestimmte Information bereitstellen. Diese Information kann aus einem oder mehreren der folgenden Merkmale geliefert werden:

- Wissen (was die Entität kennt, wie zum Beispiel Passwörter oder andere vertrauliche Informationen),
- Besitz (was die Entität hat, wie zum Beispiel ein sicherer RSA-Schlüssel),
- Biometrie (was über der Entität bekannt ist, wie Fingerabdrücke oder die Netzhaut),
- Lokation (wo die Entität sich befindet, wie zum Beispiel die Adresse des Netzes).

In Folgenden werden grundlegende Technologien zur Authentifizierung im Internet vorgestellt: die Authentifizierung mit Kennwörtern und die Authentifizierung mit einer digitalen Signatur.

3.2.1 Authentifizierung mit Kennwörtern

Eine Möglichkeit der Authentifizierung in einem Computersystem besteht in der Eingabe eines Benutzernamen und eines Kennworts. Das authentische Login-Passwort-Paar wird in einer speziellen Datenbank gespeichert. Die entsprechende Authentifizierung hat den folgenden gemeinsamen Algorithmus:

- Der Benutzer fordert den Zugriff auf das System an und gibt seinen persönlichen Benutzernamen und sein Kennwort ein.

- Die eingegebenen eindeutigen Daten werden an den Authentifizierungsserver übermittelt, wo sie mit dem bereits gespeicherten Login-Passwort-Paar verglichen werden.
- Bei der Übereinstimmung wird die Authentifizierung als erfolgreich erkannt. Bei der Differenz muss der Benutzer vom ersten Schritt erneut anfangen.

Das vom Benutzer eingegebene Kennwort kann auf zwei Arten im Netzwerk übermittelt werden:

- Unverschlüsselt auf der Basis des Password Authentication Protocol [29].
- Unter Verwendung der SSL- oder TLS-Protokolle. In diesem Fall werden die vom Subjekt eingegebenen eindeutigen Daten verschlüsselt über das Netzwerk übertragen.

Die Verwendung von wiederverwendbaren Kennwörtern weist erhebliche Nachteile auf. Das Hauptkennwort selbst oder sein Hash-Bild wird auf dem Authentifizierungsserver gespeichert. Falls das Kennwort ohne kryptografische Transformationen in den Systemdateien gespeichert wird, besteht die Gefahr, dass bei einem Angriff auf vertrauliche Informationen zugegriffen werden kann. Außerdem ist der Benutzer gezwungen, sich an sein Mehrfachbenutzungs-Kennwort zu erinnern. Darüber hinaus, wenn der Benutzer sein Kennwort selbst wählt, ohne dieses mit Hilfe der Pseudozufallszahlengeneratoren zu erstellen, kann es sein, dass das Kennwort nicht sicher genug ist. In diesem Fall kann ein Angreifer mit ausreichend Zeit das Passwort mit einer einfachen Suche hacken.

Sobald ein Angreifer ein wiederverwendbares Kennwort erhalten hat, hat er den ständigen Zugriff auf Informationen. Eine Lösung dieses Problems ist die Nutzung zufälliger oder zeitlich begrenzter Kennwörter. Der Kern dieser Methode besteht darin, dass das Kennwort nur für eine nachfolgende Anmeldung gültig ist. Ist ein neuer Zugriff auf das System erforderlich, muss das bestehende Kennwort geändert werden.

3.2.2 Authentifizierung mit digitaler Signatur

Eine weitere Möglichkeit der Authentifizierung ist die Verwendung einer digitalen Signatur. „Dabei handelt es sich um eine Art elektronisches Siegel, welches das Dokument umschließt und bei Manipulation zerstört wird. Es kann also jede Manipulation eindeutig nachgewiesen werden. […] Die Digitale Signatur ist also eine gute Möglichkeit die Authentizität, Integrität und Terminierung elektronischer Dokumente nachzuweisen“ [30].

Für die Signierung eines Dokuments werden asymmetrische Verschlüsselungsverfahren verwenden. Eine digitale Signatur wird mit einem privaten Schlüssel erstellt. Wenn ein elektronisches Dokument signiert wird, wird sein ursprünglicher Inhalt nicht verändert, sondern wird ein zusätzlicher Datenblock hinzugefügt, die sogenannte elektronische digitale Signatur. Das Herstellen dieses Blocks kann in zwei Phasen unterteilt werden. Im ersten Schritt wird ein „Fingerabdruck“ des Dokuments mittels mathematischer Funktionen berechnet. Diese entstandene Information hat dabei folgende Eigenschaften:

- eine feste Länge, unabhängig von der Nachrichtenlänge;
- die Eindeutigkeit des Abdrucks für jede Nachricht;
- die Unmöglichkeit, das Dokument auf Basis des Abdrucks wiederherzustellen.

Wenn das Dokument geändert wird, ändert sich auch sein Abdruck, was sich während der Verifizierung erkennbar ist.

Im zweiten Schritt wird der Dokumentabdruck mit dem privaten Schlüssel des Autors verschlüsselt. Das Entschlüsseln der erzeugten Signatur und somit das Bekommen des originalen Dokumentabdrucks ist nur unter der Verwendung des öffentlichen Schlüsselzertifikats des Autors möglich. Somit schützt die Berechnung des Dokumentenabdrucks vor Veränderungen nach dem Unterschreiben (Integrität). Die Verschlüsselung mit dem privaten Schlüssel des Autors bestätigt den Datenursprung des Dokuments (Authentizität).

Eine Überprüfung der digitalen Signatur des Dokuments erfolgt in mehreren Schritten. Der Empfänger verwendet das öffentliche Schlüsselzertifikat des Autors, um den signierten Abdruck zu entschlüsseln und damit den Ab-

druck des Originaldokuments zu erhalten. Mit Hilfe der speziellen mathematischen Funktionen wird aus dem empfangenen Dokument sein Abdruck berechnet. Abschließend werden die Abdrücke der originalen und empfangenen Dokumente verglichen. Das Ergebnis der Überprüfung ist eine der Antworten „wahr“ / „falsch“.

3.2.3 Diskussion

Neben den beschriebenen Problemen hat die Authentifizierung mit Kennwörtern einen weiteren Nachteil, der ihre Anwendung für die zu entwickelnde Laufzeitumgebung verhindert. Die Authentifizierung mit Kennwörtern ermöglicht die Authentizität nur solcher Entitäten nachzuweisen, die sich an einem System anmelden. In diesem Kontext muss der Hersteller der zu installierenden Software sich selber erst an einem IoT-Gerät anmelden. Nur dann kann das IoT-Gerät dem Hersteller vertrauen und anschließend das Software-Image verifizieren und installieren. Solche Lösung der Authentifizierung ist schlecht skalierbar und für den IoT-Bereich, der aus mehreren tausenden Geräten besteht, ungeeignet.

Die Verwendung der Authentifizierungsmethode mit einer digitalen Signatur löst die Probleme der Skalierbarkeit. So wie im Kapitel 2.2 beschrieben, ist es damit möglich, das erstellte Software-Image nur einmal zu signieren und weiter an einen Dateiserver zu übertragen. Der benutzte Server muss sogar nicht vertrauenswürdig sein, da jede Veränderung der Ursprungsdatei erkannt werden kann. Jedes einzige IoT-Gerät kann dabei das Image für sich selber herunterladen und muss nur über das Zertifikat bzw. den öffentlichen Schlüssel des Autors verfügen. Asymmetrische Schlüsselverfahren, die heutzutage für die Erstellung digitaler Signaturen verwendet werden, gelten bei entsprechenden Schlüssellängen als praktisch sicher. Ein Beispiel ist die zurzeit häufig angewandte RSA-Verschlüsselung.

Obwohl die Verwendung des asymmetrischen Schlüsselverfahrens das gestellte Sicherheitsproblem löst, kann diese Art der Authentifizierungsmethode nicht an jedem IoT-Gerät durchgeführt werden. Die Verschlüsselung beim Erstellen der Signatur sowie die Entschlüsselung bei ihrer Überprüfung sind mit

der Berechnung sehr komplexer mathematischer Operationen gekoppelt. Leider ist nicht jede Kategorie der IoT-Geräte in der Lage, das Schlüsselmaterial zu erzeugen oder dies schnell genug zu berechnen. Es gibt auch relativ einfache IoT-Geräte, die nur für eine konkrete Aufgabe entwickelt sind und deren Hardware-Leistung sehr begrenzt ist. In diesem Kontext wird die weitere Entwicklung der adaptiven Laufzeitumgebung auf gewisse Geräteklassen begrenzt. Diese IoT-Geräte müssen über eine ausreichende Hardware-Leistung verfügen, um die Laufzeitumgebung zu installieren und digitale Signaturen entschlüsseln zu können.

3.3 Beschreibungssprache

Der Informationsaustausch auf dem elektronischen Weg erfordert eine gegenseitige Vereinbarung, wie die Informationen zwischen dem Sender und dem Empfänger interpretiert wird. In der zu entwickelnden Laufzeitumgebung wird ein gemeinsames Verständnis zwischen den Herstellern neuer Software und den IoT-Geräten benötigt. Ansonsten wird ein neu vorbereitetes Software-Update-Image von IoT-Geräten nicht verstanden und erzeugt keine bzw. nicht die gewünschte Wirkung. Eine Lösung wäre die Verwendung einer Beschreibungssprache, die zum Verständnis zwischen beiden Endseiten dient und entsprechende Schnittstellen definiert.

Technische Einschränkungen der Vergangenheit (Speicherplatz, Übertragungskapazitäten) forderten zahlreiche Abkürzungen und Codierungen der übertragenen Information. Die benötigten Datenmengenreduzierungen führten oft zu schwer verständlichen Formen des Informationsaustauschs. Mit steigenden Leistungen der heutzutage verwendeten Hardware sowie der zunehmenden Vernetzung mit Hochbreitband-Technologien wächst der Bedarf an die Verwendung einfacherer Beschreibungssprachen. Für eine leicht verständige Form der Software-Images werden solche Lösungsansätze gesucht, die eine einfache Strukturierung von Informationen anbieten. Eine der im Abschnitt 1.2 definierten Anforderung ist die leichte Administration des zu entwickelnden Systems. Anhand dessen muss die Beschreibungssprache aufgrund ihrer Struktur auch für Menschen gut lesbar sein.

Die Darstellung einer Beschreibungssprache ist ein wesentlicher Aspekt. Es gibt unterschiedliche Konzepte von Objektnotationen und Metasprachen, die für diesen Zweck verwendet werden können. Aufgrund ihrer Beliebtheit werden JSON und XML als mögliche Kandidaten ausgewählt, die in einer Vielzahl von Anwendungsfällen verwendet werden.

3.3.1 JSON

Die JavaScript Object Notation (JSON) ist ein schlankes textbasiertes und sprachunabhängiges Datenaustauschformat. JSON definiert nur einen kleinen Satz von Formatierungsregeln für die Repräsentation der strukturierten Daten. Deswegen ist die Notation so konzipiert, dass sie effizient, selbstbeschreibend und einfach zu lesen und zu verwenden ist. [31]

Aufgrund seiner Kompaktheit ist das JSON-Format gut für Serialisierungen komplexer Strukturen geeignet. JSON ist eine Untermenge der JavaScript-Programmiersprache und daher ist es einfach, JSON in einer JavaScript-Umgebung zu verarbeiten. Darüber hinaus ist es möglich, darin vollständige und arbeitsfähige JavaScript-Funktionen zu verwenden.

3.3.2 XML

Die Extensible Markup Language (XML) ist eine Strukturbeschreibende Auszeichnungssprache, die für den Austausch strukturierter Daten im Textformat verwendet wird. Auf Basis der XML ist dem Anwender möglich, durch strukturelle und inhaltliche Einschränkungen andere anwendungsspezifische Sprachen zu definieren. Diese Einschränkungen können mit Hilfe der Document Type Description (DTD) oder des XML Schemas realisiert werden. [32]

Die XML-Spezifikation ermöglicht eine Validierung der Dokumentsyntax. Auch mit der Verwendung von DTD können die im Dokument definierten Attribute und Elemente kontrolliert werden. DTD repräsentiert grammatikalische Regeln, um Dokumente eines bestimmten Typs zu deklarieren. DTD beschränkt den Satz von Elementen und deren Attributen, die in einem XML-

Dokument verwendet werden können, die Reihenfolge, in der sie auftreten können, und deren zulässige Eltern/Kind-Beziehungen.

3.3.3 Diskussion

JSON ist ein Datenaustauschformat, während XML eine Auszeichnungssprache ist. Die Syntax von JSON ist viel kompakter gestaltet und deswegen ist JSON oft lesbarer und leichter schreibbar als XML. Auch benötigt JSON in der Regel keine Zusatzinformation zur Übermittlung und hat damit einen geringeren Overhead im Vergleich zu XML. JSON ist besser für den Datenaustausch an flexiblen Schnittstellen geeignet.

Während JSON die Flexibilität anbietet, ermöglicht XML, die Struktur von Dokumenten zu beschreiben sowie den Inhalt von Elementen und Attributen zu beschränken. Damit ist XML besser für Systeme mit rigiden Schnittstellen einsetzbar.

Um möglich mehr Kontrolle über den Software-Update-Mechanismus zu schaffen, müssen die Struktur und die Grammatik der Software-Images validiert und verifiziert werden. Aus diesem Grund wird für die Durchführung der Software-Updates die XML als Format der Beschreibungssprache verwendet.

4 Design und Architektur

Nachdem die Anforderungen an die zu entwickelnde Laufzeitumgebung definiert sind, soll im Folgenden die Frage, wie das System die Anforderungen erfüllt, beantworten werden. Dieses Kapitel beschreibt die Festlegung der Systemarchitektur, Beziehungen zwischen ihren Komponenten und die wichtigsten Systemsinteraktionen.

4.1 Statische Sichtweise

Die statische Sicht auf die Software-Architektur definiert konstante Hauptkomponenten des Gesamtsystems und ihren Zusammenhang. Die folgende Abbildung stellt diese schematisch dar:

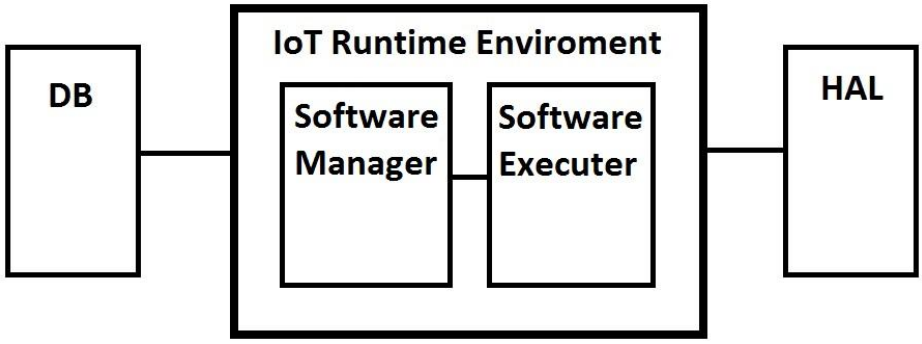

Abbildung 4.1: Statische Hauptkomponenten des Gesamtsystems

Das Gesamtsystem basiert sich auf drei Komponenten: einer Datenbank (DB), einer Hardwareabstraktionsschicht (HAL, engl. Hardware Abstraction Layer) und der Laufzeitumgebung (IoT Runtime Enviroment) selbst. Die Datenbank enthält für die Installation vorbereitete Software-Images. Die Hardwareabstraktionsschicht ermöglicht Zugriffe auf die Hardware eines IoT-Gerätes. Die Laufzeitumgebung ist die zentrale Komponente der Architektur und dient zur dynamischen Installation, Ausführung und Verwaltung der Software am IoT-Gerät.

4.1.1 Datenbank

Analog zum nicht vertrauenswürdigen Dateiserver, der von der Architekturlösung aus dem Abschnitt 2.2.1 verwendet wird, enthält die Datenbank Software-Images für ihre weitere Installation an IoT-Geräten. Bei einem Software-Update fordern IoT-Geräte ein entsprechendes Software-Image von der Datenbank an. Dazu muss der Standort der Datenbank bekannt und erreichbar sein.

Um die Sicherheitsanforderung zu erfühlen, muss die Datenbank nur solche Software-Images beinhalten, die von ihren autorisierten Herstellern kryptografisch signiert sind. Dies ermöglicht zukünftig dem IoT-Gerät, die zu installierenden Software zu verifizieren und das Update zu akzeptieren.

Beispiele für die Datenbank als Komponente des Gesamtsystems sind FTP-Server, Web-Server oder USB-Sticks. Diese Bachelorarbeit befasst sich nicht mit der Implementierung der Datenbank, sondern nutzt diese als gegeben.

4.1.2 Hardwareabstraktionsschicht

Eine HAL ist eine Programmierschnittstelle, die zu Interaktionen mit der Hardware dient. Mit der Einsetzung der HAL in die Softwarearchitektur wird der adaptiven Laufzeitumgebung ermöglicht, auf die Hardware eines konkreten IoT-Geräts zuzugreifen.

Die HAL ist eine wichtige Komponente des Gesamtsystems, denn durch die HAL wird die Interoperabilität der Software unterstützt. Um die zu entwickelnde Laufzeitumgebung auf unterschiedliche IoT-Geräte und ihre Betriebssysteme anzupassen, muss nur die HAL oder ein Teil der HAL geändert werden. Jedes IoT-Gerät, an dem die Laufzeitumgebung installiert wird, muss eine detaillierte Beschreibung seiner unterstützten Hardwarefunktionen zur Verfügung stellen. Dabei muss besonders auf die Namensgebung der Hardwarefunktionen beachtet werden. Die Namensgebung muss einer bestimmten Konvention entsprechen, um die Interoperabilität zu ermöglichen.

Das Ziel dieser Konvention ist die Abstrahierung von einzelnen Erscheinungen der Funktionen mit ihren individuellen Eigenschaften. Der Sinn dieser Abstraktion der Namensgebung kann mit einem Beispiel aus dem Gebiet Smart-Home verdeutlicht werden. Eine typische Funktion einer Smart-Lampe ist die Einstellung ihrer Helligkeit. Eine Funktion eines Fensterrollladens ist die Einstellung seiner Höhe. Mit der Verwendung eines allgemeinen abstrakten HAL-Funktionsnamen „einstellen(int Prozent)“ für entsprechende Hardwareaufrufe wird es möglich, dieselbe Software, die diese Funktion nutzt, an beiden IoT-Geräten zu installieren. Dabei muss es nicht unterscheiden, um welchen IoT-Gerätetyp genau es sich handelt. So eine Abstrahierung und somit die Interoperabilität kann mit Hilfe der Interfaces erreicht werden. Damit wird ein IoT-Gerät nur solche Software-Images akzeptieren, deren genutzte HAL-Funktionen von diesem Gerät unterstützt werden.

4.1.3 Laufzeitumgebung

Die zentrale Komponente der Architektur ist die adaptive Laufzeitumgebung, die an IoT-Geräten installiert wird. Die Laufzeitumgebung übermittelt Software aus der Datenbank an das Gerät, installiert und führt sie aus, in dem die Laufzeitumgebung die HAL anspricht. Dabei besteht die Laufzeitumgebung selbst aus zwei weiteren Komponenten: dem Software-Manager und dem Software-Executor.

Software-Manager

Wie der Name des Managers sagt, ist er für die Verwaltung der Software-Images an IoT-Geräten zuständig. Der Software-Manager importiert für das Software-Update vorbereitete Images aus der Datenbank, installiert und verwaltet sie am IoT-Gerät. Dabei übernimmt der Manager die an das System gesetzten Anforderungen der Sicherheit, Atomarität, Dynamik und Administration.

Der Software-Manager validiert jede Software vor ihrer Installation. Durch das Überprüfen der digitalen Signatur entscheidet der Manager, ob der Autor der Software berechtigt ist, die Aktualisierungen vorzunehmen. Auch

erkennt somit der Manager Veränderungen an der zu installierenden Software. Falls die Software verifiziert wird, wird sie zusätzlich geprüft, ob die benutzten Hardware-Aufrufe von der HAL unterstützt werden.

Außerdem gewährleistet der Manager die Atomarität der Installation der akzeptierten Software. Ein Software-Update wird komplett oder gar nicht installiert.

Der Software-Manager erfüllt die Dynamik-Anforderung. Solche Dienste wie der Import, die Validierung oder die Installation einer neuen Software werden zur Laufzeit durchgeführt. Die bereits laufende Software darf dadurch nicht unterbrochen werden.

Der Software-Manager ist der Teil des Gesamtsystems, mit dem der Nutzer interagiert. Aus diesem Grund bietet der Manager eine einfache und leicht verständliche Verwaltung der Laufzeitumgebung und der installierten Software an. Damit erfüllt der Software-Manager die Administration-Anforderung.

Software-Executor

Der zweite innere Teil der adaptiven Laufzeitumgebung ist der Software-Executor. Der Executor führt die installierte Software aus. Falls die laufende Software auf Hardware des IoT-Geräts zugreifen möchte, ruft der Software-Executor die entsprechenden HAL-Funktionen auf. Der Executor übernimmt die weiteren Anforderungen an das System: die Robustheit und die Fail-Safeness.

Der Executor führt die Software in einem anderen Speicherplatz aus, als der, in dem die Software gespeichert wird. Diese Speichertrennung schützt das IoT-Gerät vor dem Ausfall bei unerwarteten Fehlern bei der Ausführung der Software. Falls die auszuführende Software fehlschlägt, wird es trotzdem möglich, auf den Speicherplatz mit den vorherigen und arbeitsfähigen Versionen der Software zuzugreifen und ein Rollback-Mechanismus durchzuführen.

4.2 Dynamische Sichtweise

Die dynamische Sicht auf die Software-Architektur beschreibt Prozesse in der adaptiven Laufzeitumgebung und ihre Interaktionen.

4.2.1 Software-Manager

Die Einsetzung des Software-Managers in die Architektur gewährleistet die Installation neuer Software und die Verwaltung der bereits installierten Software. Die folgende Abbildung visualisiert die Zustände des Managers während der Laufzeit des Systems und ihre Übergangsbedingungen:

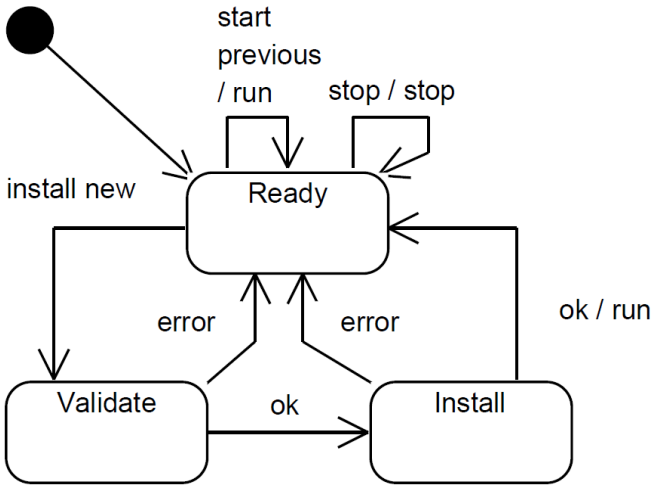

Abbildung 4.2: Zustandsübergangsdiagramm des Software-Managers

In dem Anfangszustand „Bereit“ (Ready) ermöglicht der Manager Interaktionen mit dem System. So kann der Nutzer die Ausführung einer laufenden Software unterbrechen oder aus einer Liste der bereits installierten Software eine Bestimmte wählen. In dem Fall sendet der Manager dem Software-Executor ein entsprechendes Kommando und wartet auf die weiteren Angaben vom Nutzer.

Wenn der Nutzer des IoT-Geräts sich entscheidet, eine neue Software aus der Datenbank zu installieren, wird der Software-Manager in den Validierungszustand versetzt. In dem Zustand wird die zu installierende Software durch folgende Punkte validiert:

- Die Größe der Software. Jedes IoT-Gerät verfügt über unterschiedliche Größe seines Speicherplatzes, den das Gerät für die Installation der Software freigeben kann. Diese Einschränkung des Speicherplatzes wird in den Einstellungen der Laufzeitumgebung konfiguriert. Die Größe der zu installierenden Software wird dadurch kontrolliert.
- Die digitale Signatur. Damit werden die Authentizität und die Integrität der Software überprüft.
- Die Struktur der Software. Um mehr Kontrolle über die zu installierende Software zu schaffen, werden strukturelle und inhaltliche Einschränkungen für Software-Images eingesetzt. Die Struktur der Software-Images muss den definierten Regeln entsprechen.
- Die Syntax des auszuführenden Codes in Software. Um eine neue Software richtig auszuführen, muss ihr Code eine korrekte Syntax haben. Dazu wird der komplette Quellcode analysiert und seine Syntax geprüft.
- Die HAL. Alle Hardware-Aufrufe, die die zu installierende Software verwendet, müssen von der HAL unterstützt werden.

Falls die Software validiert wird, wird der Manager in den Installationszustand versetzt. Hier wird die Software am IoT-Gerät gespeichert und installiert. Nach der erfolgreichen Installation wird dem Software-Executor ein Kommando gegeben, das signalisiert, dass die neue Software gestartet werden kann. Der Manager wird anschließend in seinen Anfangszustand versetzt.

Falls ein erneuter Start der bereits validierten und installierten Software benötigt wird, kann dies direkt durch ein Signal an den Executor erfolgen. In diesem Fall ist die Durchführung der oben beschriebenen Vorbereitungsschritte nicht mehr nötig.

Jede Abweichung von den Regeln bei der Software-Validierung sowie jeder Fehler während der Software-Installation führt zur Unterbrechung der Installation und versetzt den Manager zurück in den Bereit-Zustand.

4.2.2 Software-Executor

Die folgende Abbildung stellt die Zustände, die der Software-Executor durchläuft, dar:

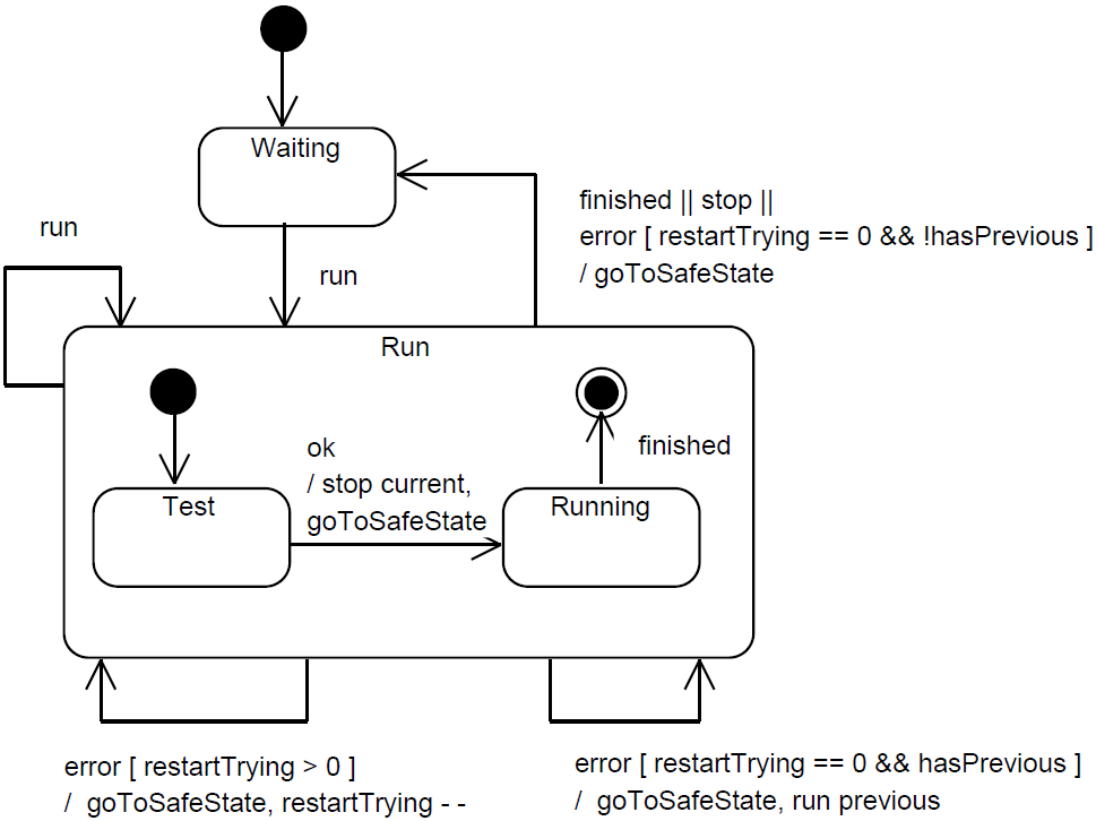

Abbildung 4.3: Zustandsübergangsdiagramm des Software-Executors

Nach dem die adaptive Laufzeitumgebung an einem IoT-Gerät installiert und gestartet wird, befindet sich der Software-Executor in dem Wartezustand und wartet auf die Kommandos vom Software-Manager. Entscheidet sich der Manager, eine neue Software am Gerät ausführen zu lassen, wird der Software-Executor in den Rechnend-Zustand versetzt.

Bevor eine neue Software gestartet wird, müssen ihre funktionalen Einzelteile am IoT-Gerät getestet und auf korrekte Funktionalität geprüft werden.

Jeder Hersteller der zu installierenden Software muss ein Testszenario definieren. Das Testszenario muss überprüfen, ob die Software an einem konkreten IoT-Gerät spezifikationsgemäß funktioniert. Der Software-Manager hat während seines Validierungszustands schon überprüft, dass alle in der Software verwendeten Hardwareaufrufe von der HAL des IoT-Geräts unterstützt werden. Es heißt aber noch nicht, dass die Hardware korrekt für diese Software funktioniert. Das ist zum Beispiel der Fall, wenn die Funktionalität einer Software temperaturabhängig ist und ein Temperatursensor des Geräts Daten liefert, diese aber Werte aus einem nicht unterstützen Bereich sind. In solcher Situation muss der Start der neuen Software verhindert und dem Nutzer eine entsprechende Meldung angezeigt werden. Nur wenn alle Tests erfolgreich bestanden sind, darf die aktuell auszuführende Software beendet und die neue gestartet werden.

Wenn die Ausführung einer installierten Software beendet wird (unabhängig vom Grund: durch den Manager, wegen eines Fehlers oder terminiert selbst), muss das IoT-Gerät in einen sicheren Zustand (engl. Safe State) gebracht werden. Diese Funktion des Software-Executors schützt das Gerät vor einer unkorrekt beendeten Ausführung der Software. Zum Beispiel, falls die Software Motoren des Geräts aktiviert hat und dann ihre Ausführung abgebrochen wurde, sollen alle Motoren wieder deaktiviert werden. Um diese Funktionsweise des Software-Executors zu erfüllen, muss jedes IoT-Gerät in der HAL eine Schnittstelle implementieren, die das Gerät in seinen Safe State versetz.

Eine weitere wichtige Eigenschaft des Software-Executors ist seine Reaktion auf unerwartete Fehler während der Software-Ausführung. Dabei sind weitere Maßnahmen denkbar:

- auf eine Eingabe vom Nutzer warten,
- eine vorherige und lauffähige Version der Software starten,
- die unterbrochene Software erneut starten.

Ein verteiltes Netz kann aus mehreren tausenden IoT-Geräten bestehen. In dem Fall ist das Warten, bis der Nutzer die aufgetretenen Fehler manuell aufhebt, nicht möglich. Die weitere Lösung, in der eine vorherige und lauffähige Version der Software gestartet wird, löst das Problem der Skalierbarkeit und benötigt keine Unterstützung von Nutzern. Eine Unterbrechung der Soft-

ware heißt aber nicht immer, dass die Software fehlerhaft ist, sondern es können andere Grunde dafür geben. So zum Beispiel, dass die Hardware nicht rechtzeitig geantwortet hat, da das Gerät sich zum Zeitpunkt des Aufrufs unter einer hohen Rechenlast befand. Für diesen Fall ist die Lösung besser geeignet, die die abgestürzte Software erneut startet.

Da es nicht immer möglich ist, die genaue Ursache des Fehlers automatisch zu bestimmen, soll der Software-Executor eine optimale Lösung verwenden. So wird jedes IoT-Gerät in der Konfiguration der adaptiven Laufzeitumgebung je nach seinem Typ eine Anzahl der Versuche definieren, wie oft eine Software im Fehlerfall erneut gestartet werden muss. Dieser Wert kann auch von der auszuführenden Software je nach ihrer Komplexität überschrieben werden. Falls der Fehler sich nach dem Neustart wiederholt, wird diese Software vom System als fehlerhaft notiert und eine vorherige und arbeitsfähige Version der Software gestartet. Für sie gilt im Fehlerfall das gleiche Verfahren. Soll es keine arbeitsfähige Software am Gerät geben, wird eine „default“ Software ausgeführt. Als „default“ Software ist hier ein Systemzustand bezeichnet, in dem die Laufzeitumgebung auf die weiteren Eingaben vom Nutzer wartet. Mit diesem Verfahren wird im Fehlerfall der Rollback-Mechanismus des Systems realisiert. Damit führt das System zu keinem Zeitpunkt zum Ausfall des Gerätes und kann immer mit dem Nutzer interagieren.

4.2.3 Diskussion

Die vorgeschlagene Architektur bietet nicht nur eine Lösung für die gestellten Anforderungen an, sondern unterstützt auch durch ihren Minimalismus möglichst viele unterschiedliche Klassen der IoT-Geräte. Je komplizierter die Architektur der Laufzeitumgebung ist, desto mehr Ressourcen werden für ihre Funktionalität benötigt. Je nach Art haben IoT-Geräte unterschiedliche Hardware-Leistungen. Die minimale Form der Architektur ermöglicht die Verwendung der Laufzeitumgebung auch an einfachen Geräten.

Außerdem benötigt eine einfache Architektur eine minimale Bearbeitungszeit interner Prozesse. Damit wird die Laufzeitumgebung die Ausführung der installierten Software nicht verlangsamen. So wird es möglich, die Lauf-

zeitumgebung auch an IoT-Geräten mit Echtzeit-Anforderungen zu installieren. D.h. an den Geräten, deren Reaktionszeit auf bestimmte Ereignisse von hoher Bedeutung ist.

Die minimale Form der Architektur spiegelt sich auch in der Robustheit der Laufzeitumgebung wieder. Eine kleinere Zahl der Zustände bietet eine schnellere Reaktion auf aufgetretene Fehler an. So wird die Zeit für die Ausführung des Rollback-Mechanismus minimalisiert. Dieser Punkt ist für die Echtzeit-Geräte sehr kritisch, da während des Rollback-Mechanismus keine Software ausgeführt wird.

5 Implementierung eines Prototyps

Der im vorherigen Kapitel erarbeitete Entwurf und die Architektur werden in Form eines Prototyps umgesetzt. Im folgenden Teil der Arbeit werden die Einzelheiten der Umsetzung veranschaulicht.

Wie im Abschnitt 3.1. diskutiert, wird für die Implementierung eines Prototyps JavaScript unter Verwendung von Node.js benutzt. Die Einbindung diverser Bibliotheken und Werkzeuge ist durch den Paketmanager NPM (Node Package Manager [33]) ermöglicht. Die Realisierung des Prototyps gliedert sich in zwei Bereiche: die Administration der Laufzeitumgebung und die Verwaltung sowie die Ausführung der Software.

5.1 Administration

5.1.1 REST-Web-Service

Für die Durchführung der Software-Updates müssen entsprechende Software-Images aus der Datenbank an IoT-Geräte übertragen werden. Bei der Implementierung eines Prototyps wird diese Kommunikationsstrecke sowie die gesamte Verwaltung der Laufzeitumgebung durch den Nutzer bedient. Ein natürlicher und einfacher Zugriff auf Systemschnittstellen wird für den Nutzer über das Netz durch Web-API (Application Programming Interface) und zwar REST-Web-Services (Representational State Transfer [34]) realisiert. Dadurch werden Interaktionen mit der Laufzeitumgebung ermöglicht und eine virtuelle Repräsentation des Systemzustands dargestellt.

Ein wichtiger Grundsatz der REST-Web-Services ist die Adressierbarkeit. In REST-Architekturen wird jede Informationseinheit als Ressource betrachtet und die Ressourcen werden mithilfe Uniform Resource Identifiers (URI) adressiert [35]. So erfolgt der Zugriff auf Benutzerschnittstellen durch den Aufruf der URL:

```
http://host:port
```

Als `host` wird die IP-Adresse des IoT-Geräts eingesetzt. `port` ist der Port, der am Gerät für die adaptive Laufzeitumgebung verwendet wird, und ist in den Einstellungen der Umgebung konfigurierbar.

5.1.2 Kommunikation

Eine der Basiseigenschaften eines REST-Web-Services ist die explizite Verwendung von HTTP-Methoden, in der Form, wie sie im RFC 2616 definiert sind [36]:

- GET – Anforderung einer Ressource;
- POST – Erzeugung einer Ressource an der Serverseite;
- PUT – Änderung des Zustands einer Ressource;
- DELETE – Löschen einer Ressource.

Die Kommunikation zwischen dem Nutzer und der am IoT-Gerät installierten Laufzeitumgebung ist als Request/Response realisiert, und zwar durch die Verwendung der GET- und POST-Übertragungsmethoden. Die folgende Tabelle stellt die unterstützten Anfragen dar, die der Nutzer an die Laufzeitumgebung senden kann:

Methode	Pfad	Bedeutung
GET	`/`	Ruft die Startseite der Laufzeitumgebung mit einem Formular auf, um ein neues Software-Image aus der Datenbank zu wählen.
POST	`/upload`	Sendet ein aus der Datenbank gewähltes Software-Image an die Lauzeitumgebung und startet somit den Installations- und Ausführungsprozess.
GET	`/status`	Zeigt am Gerät installierte Software mit ihrem aktuellen Status. Es werden drei verschiedene Status-Codes verwendet: • 1 – aktuell auszuführende Software • 0 – bereits installierte Software, deren vorherige Tests und Ausführung fehlerfrei sind • -1 – bereits installierte Software, deren vorherige Tests oder Ausführung fehlschlug
GET	`/run/<softwareName>`	Beendet die laufende Software und startet den Ausführungsvorgang einer bereits installierten Software mit dem eingegebenen Identifikationsnamen
GET	`/stop`	Beendet die laufende Software
GET	`/hal`	Zeigt alle vom IoT-Gerät unterstützten HAL-Funktionen

Tabelle 5.1: Benutzeranfragen an die Laufzeitumgebung

5.2 Funktionsweise

5.2.1 Systemeigenschaften

Wie im Abschnitt 3.1. diskutiert, ist der Prototyp der Laufzeitumgebung unter Verwendung von Node.js realisiert. Aus diesem Grund sind bei der Implementierung besondere Eigenschaften zu benennen.

Die entwickelte Architektur ist ereignisgesteuert. Sie beginnt mit dem Auslösen eines Ereignisses und endet mit einer beliebigen Reaktion auf das Ereignis. Diese Eigenschaft hat wesentliche Vorteile. Die Architektur verbraucht weniger Arbeitsspeicher im Vergleich zu Anwendungen, die fürs Zusammenspiel unabhängiger Komponenten mehrere Threads starten [37]. Wenn es keine Ereignisse gibt, befindet sich die Laufzeitumgebung im Ruhezustand. Das ermöglicht die Installation der Laufzeitumgebung auch auf den IoT-Geräten, die mit Akkus versorgt werden. Es werden keine kontinuierlichen Operationen im Ruhezustand ausgeführt, damit verbraucht die CPU keinen zusätzlichen Strom.

Ein weiterer Fokus liegt auf der Performance. Für Zugriffe auf das Dateisystem wird das nonblocking I/O eingesetzt. Die Zugriffe werden in getrennten Threads ausgeführt und damit den Ablauf der anderen Anweisungen nicht verlangsamen.

5.2.2 Gliederung in Module

Die Laufzeitumgebung beinhaltet mehrere Prozesse. Software-Images werden am IoT-Gerät installiert, ausgeführt und verwaltet. Auch werden Kommandos vom Nutzer verarbeitet. Um diese Operationen zu bearbeiten, gliedert sich die Laufzeitumgebung in folgende Module.

- Der Request-Handler, der Anfragen vom Nutzer annimmt, entsprechende Bearbeitungsvorgänge startet und eine Antwort bzw. den aktuellen Zustand der Laufzeitumgebung dem Nutzer zurückliefert.

- Der Software-Validator, der die zu installierende Software gegen die im Abschnitt 4.2.1 beschriebenen Punkte validiert und die weitere Installation der Software verifiziert.
- Der Software-Tester, der die korrekte Funktionalität der Software vor ihrer Ausführung testet.
- Der Software-Executor, der die installierte Software ausführt und gegebenenfalls die benötigten HAL-Funktionen aufruft. Aufgrund der Robustheitsanforderungen findet die Ausführung in einem isolierten Speicherplatz innerhalb einer virtuellen Maschine statt. Dadurch ist das IoT-Gerät vor Schadcode geschützt.
- Der Software-Manager, der die Kontrolle über jede installierte Software übernimmt. Der Manager reagiert auf mögliche Fehler während der Ausführung der Software und startet bei Bedarf den Rollback-Mechanismus.

5.3 Struktur des Software-Images

5.3.1 Schnittstellen

Für die Installation der Software muss eine eindeutige Struktur des Software-Images zwischen dem Software-Hersteller und dem IoT-Gerät vereinbart werden. Jedes Software-Image muss folgende minimale Schnittstellen definieren.
Information über die zu installierende Software:

- Ein eindeutiger Identifikationsname des Software-Images. Dies wird benötigt, um Software-Images in der Datenbank sowie am IoT-Gerät zu verwalten.
- Namen der Hardware-Funktionen, die die zu installierende Software verwendet. Anhand dieser Information wird überprüft, ob der Typ des IoT-Geräts für die Ausführung dieser Software geeignet ist.
- Der Quellcode der Software, der nach der erfolgreichen Installation vom IoT-Gerät ausgeführt wird.

- Die Tests, die vor dem Start der Software ausgeführt werden und die korrekte Funktionalität dieser Software am IoT-Gerät überprüfen.

Information über den Hersteller der Software:

- Der Name des Herstellers. Der Hersteller muss der adaptiven Laufzeitumgebung bekannt sein. So wird festgestellt, ob dieser Hersteller berechtigt ist, Aktualisierungen vorzunehmen.
- Die Signatur des Herstellers, um die möglichen Manipulationen am Software-Image zu erkennen.

5.3.2 XML Schema

Wie im Abschnitt 3.3. diskutiert, werden die oben beschriebenen Schnittstellen eines Software-Images in einem XML-Format dargestellt. Um die Struktur solches XML-Dokuments zu definieren und einschließlich zu validieren, wird ein XML-Schema, abgekürzt XSD (XML Schema Definition) verwendet [38]. Anders als bei den klassischen XML-DTDs wird die Struktur in Form eines XML-Dokuments beschrieben und somit besser für Menschen lesbar. Darüber hinaus wird durch die XSD eine große Anzahl von Datentypen unterstützt.

Das folgende Listing zeigt das Schema für die Schnittstellendefinition der zu installierenden Software-Images.

```
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="document">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="installationSoftware">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="softwareName"/>
              <xs:element name="requiredHalFunctions">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:string" name="function"
                                maxOccurs="unbounded" minOccurs="0"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:string" name="software"/>
              <xs:element type="xs:string" name="test"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="softwareProvider">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="name"/>
              <xs:element type="xs:string" name="signature"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 5.1: XML-Schema der Struktur eines Software-Images

6 Experimente

Im Folgenden werden der entwickelte Prototyp und seine Architektur durch Experimente getestet. Dabei wird die Zeit analysiert, die für die Installation und die Ausführung einer Software benötigt wird.

Die gemessenen Werte hängen stark von der Hardware-Leistung eines konkreten IoT-Geräts ab. Also liefern absolute Werte keine aussagekräftige Information. Aufgrund dessen konzentrieren die folgenden Experimente sich auf das relative Verhalten der Laufzeitumgebung und den wesentlichen Trend. Die Analyse wird auf dem Raspberry Pi 3 Model B (CPU: ARMv7 64 Bit Quard Core 1.2 GHz, RAM: 1024 MB) durchgeführt.

6.1 Kosten der Middleware

Die Ausführung einer Software am IoT-Gerät wird durch die adaptive Laufzeitumgebung realisiert. D.h. die Laufzeitumgebung bildet eine zusätzliche Schicht zwischen der auszuführenden Software und dem Betriebssystem des Geräts. Außerdem wird die Software innerhalb einer virtuellen Maschine gestartet. Diese Faktoren können die Ausführung der Software verlangsamen. Das folgende Experiment zeigt, wie die Rechenzeit einer direkt gestarteten Software sich von der Rechenzeit derselben Software innerhalb der Laufzeitumgebung unterscheidet.

6.1.1 Experimentaufbau

Um die CPU-Last zu erzeugen, wird der folgende JavaScript-Code ausgeführt und die für die Bewältigung benötige Rechenzeit gemessen:

```
var sum = "";
for (var i = 0; i < 10*1000*1000; i++) {
    sum += "a";
}
```

Der Code wird in einer JavaScript-Datei gespeichert und mit Hilfe der Node.js-Plattform gestartet. Die mittlere Rechenzeit beträgt 10 824 Millisekunden.

Nun wird der Code in einem für die Laufzeitumgebung akzeptierten Software-Image-Format (s. Abschnitt 5.3.) dargestellt. Der entwickelte Prototyp wird gestartet und das vorbereitete Software-Image wird ihm übergeben. Die mittlere Ausführungszeit beträgt 116 686 Millisekunden (ca. 2 Minuten). Also die Rechenzeit unter der Laufzeitumgebung ist ca. 11-mal größer, als wenn der Code direkt am IoT-Gerät ausgeführt wird.

6.1.2 Diskussion

Der Verlangsamungsfaktor 11 ist kein Faktor, der vernachlässigt werden darf, und muss genauer untersucht werden. Die Architektur der Laufzeitumgebung ist auf Basis der im Kapitel 1 gesetzten Anforderungen realisiert. Unteranderem sind es die Robustheit und die Sicherheit.

Die Installation einer neuen Software fordert die Vertraulichkeit ihrer Quelle. Dafür verwendet die Laufzeitumgebung eine digitale Signatur für jedes Software-Image. Dennoch muss ein ausreichender Schutz des Geräts gegen Schadsoftware gewährleistet werden. Typische Beispiele eines schadhaften Codes sind Kommandos „`rm -rf /`“, das alle Dateien löscht, oder „`process.exit(0)`“, das die Ausführung der Laufzeitumgebung unterbrechen kann. Um den notwendigen Schutz zu erreichen, werden Rechte und der Zugriffsbereich der auszuführenden Software begrenzt. So fährt der entwickelte Prototyp für die Ausführung der Software eine isolierte virtuelle Maschine hoch. Damit hat die Ausführung der oben stehenden Schadcode-Beispiele keine Auswirkung auf die Laufzeitumgebung sowie das IoT-Gerät. Der Start der virtuellen Maschine und die Ausführung des Codes darin sind die Gründe für eine so große Verlangsamung der Rechenzeit. Wie das Experiment zeigt, ist die im Prototyp verwendete „vm“ (virtuelle Maschine) aus der Node.js-Bibliothek [39]

keine ideale Lösung des Sicherheitsproblems. Selbstverständlich kann dies unter Verwendung anderer Werkzeuge oder Einstellungen deutlich optimiert werden. Aber auf keinen Fall darf die Performance auf Kosten der Sicherheit verbessert werden.

Unter der Annahme, dass man der installierten Software absolut vertraut, wird nun die Software nach dem Validierungsprozess gleich von der Laufzeitumgebung ausgeführt (ohne virtuelle Maschine). Die Messung der Rechenzeit zeigt absolut identische Werte, wie eine direkte Ausführung des Codes am IoT-Gerät. Dies ist möglich dank der ereignisgesteuerten Architektur. Denn nachdem die installierte Software gestartet wird, befindet sich die Laufzeitumgebung im Ruhezustand und erzeugt damit keine CPU-Last.

6.2 Kosten der Verfügbarkeit

Laut der Dynamik-Anforderung muss jedes Software-Update parallel zu der aktuell laufenden Software durchgeführt werden. Also während eine installierte Software ausgeführt wird, darf die Laufzeitumgebung dadurch nicht blockiert werden und muss für die weiteren Anfragen immer verfügbar sein. Das folgende Experiment zeigt, dass der Installationsprozess einer neuen Software gleichzeitig mit der Ausführung der bereits installierten Software bearbeitet wird und wie diese zwei nebenläufigen Prozesse einander verlangsamen können.

Um die Parallelität zu analysieren, wird das Experiment auf zwei unterschiedlichen Geräten durchgeführt. Die erste Hardware ist derselbe Raspberry Pi 3 Model B aus dem vorherigen Experiment, der über vier Kerne verfügt und damit mehrere Aufgaben „echt-gleichzeitig“ verarbeiten kann. Die zweite Hardware ist ein Raspberry Pi Model B (CPU: ARM1176JZF Single Core 700 MHz, RAM: 512 MB), der nur einen Kern hat und damit mehrere Prozesse nur „pseudo-parallel“ verarbeiten kann.

6.2.1 Experimentaufbau

Mehrkernprozessor: echt-parallel (Raspberry Pi 3B)

Fürs Experiment werden zwei Software-Images vorbereitet. Der auszuführende Code der ersten Software besteht aus derselben for-Schleife wie im vorherigen Experiment. Die aus dem ersten Experiment bekannte Rechenzeit beträgt 116 686 Millisekunden. Sobald die erste Software startet, wird der Laufzeitumgebung das zweite Software-Image für die Installation übergeben. Um eine hohe CPU-Last bei der Installation zu erzeugen, wird im Software-Image als Test eine endlose while-Schleife verwendet:

```
while (true) {}
```

Somit startet die Laufzeitumgebung einen „teuren“ Installationsprozess, der parallel mit der Ausführung der ersten Software verarbeitet wird. Mit dieser extra CPU-Last beträgt die mittlere Rechenzeit der laufenden Software 118 444 Millisekunden. Somit benötigt die Ausführung der Software um 1,5% mehr Zeit für die Terminierung.

Einzelkernprozessor: pseudo-parallel (Raspberry Pi B)

Die Laufzeitumgebung wird auf Raspberry Pi Model B installiert und das Software-Image aus dem vorherigen Experiment wird der Umgebung übergeben. Da die Generation B älter als 3B ist und sie unterschiedliche Hardware-Leistung hat, wird die Software langsamer ausgeführt. So ist die mittlere Rechenzeit ca. neunmal größer als beim Raspberry Pi 3B und beträgt 1 030 603 Millisekunden (ca. 17 Minuten).

Nachdem die Rechenzeit in dem ruhigen Zustand gemessen ist, wird die Software nochmal gestartet. Parallel wird die Laufzeitumgebung mit der Installation des Software-Images mit dem unendlichen Test belastet. So braucht die erste Software für die Terminierung 2 012 585 Millisekunden (ca. 33.5 Minuten). Also hat sich die Ausführungszeit der Software um 95,3% vergrößert.

6.2.2 Diskussion

Das durchgeführte Experiment zeigt, dass die entwickelte Architektur der Laufzeitumgebung ihre Aufgaben nebenläufig bearbeitet. So ist es möglich, einen neuen Software-Installationsprozess parallel zur Ausführung der existierenden Software durchzuführen.

Die für den ersten Teil des Experiments genutzte Hardware hat einen Mehrkernprozessor. Damit können mehrere Aufgaben echt-gleichzeitig verarbeitet werden. Deshalb sind die gemessenen Zeitwerte ohne und mit einem nebenläufigen Installationsprozess vergleichbar.

Je nach Hardware-Leistung der IoT-Geräte kann eine parallel laufende Installation die Ausführung der Software wesentlich verlangsamen. Insbesondere ist es für Einzelkernprozessoren wichtig, bei denen zu jedem Zeitpunkt nur genau ein Prozess verarbeitet werden kann. Somit kann die Umgebung potentielle Laufzeitprobleme haben. Abhängig vom Scheduler kann die Ausführung in nicht genau vorhersagbare Laufzeitverteilung kommen. Wie der zweite Teil des Experiments zeigt, kann die Verarbeitung bis zu einem Faktor von zwei verlangsamt werden.

Die Verlangsamung der Softwarebearbeitung ist bei den Geräten mit einer harten Deadline zu beachten. Falls während eines Software-Updates das Gerät notwendige Ergebnisse nicht mehr rechtzeitig bearbeitet, kann es zu einer Katastrophe kommen.

6.3 Kosten der Installation

Für die Installation einer neuen Software startet die adaptive Laufzeitumgebung erst einen Validierungsprozess. Dabei werden rechenaufwändige Operationen durchgeführt, wie die Überprüfung der Signatur und der Image-Struktur und die Analyse des Quellcodes. Das folgende Experiment stellt die Kostenaufteilung der einzelnen Installationsoperationen in Abhängigkeit von der Größe des zu installierenden Software-Images dar.

6.3.1 Experimentaufbau

Für die Durchführung des Experimentes werden der Laufzeitumgebung zur Installation Software-Images mit unterschiedlichen Größen des auszuführenden Codes übergeben. Dabei wird die Rechenzeit der folgenden Vorbereitungsschritte gemessen:

- Struktur: das Parsing des Inhaltes des Images sowie die Validierung gegen das definierte Schema.
- Signatur: das Prüfen der digitalen Signatur.
- Quellcode: die Analyse des auszuführenden Codes.
- Installation: das Speichern der Software im für die Umgebung akzeptablen Format.

Die folgende Tabelle zeigt, wie die Rechenzeit der Installationsoperationen sich mit der Vergrößerung des Software-Images verändert. Die erste Spalte stellt die Größe der zu installierenden Datei dar, die mit der Zeilenanzahl des auszuführenden Codes gemessen wird. Für eine im Computersystem gewöhnliche Darstellung der Dateigröße ist in Klammern die Größe des gesamten Software-Images in Kilobyte eingetragen. Die weiteren Spalten beinhalten die gemessene Rechenzeit der oben beschriebenen Installationsschritte sowie ihre gesamte Zeit. Um die Zeitaufteilung zwischen einzelnen Operationen zu präsentieren, sind neben den absoluten Werten ihre prozentualen Anteile von der Gesamtzeit eingetragen. Die Gesamtzeit bezeichnet die Zeit, die für die komplette Installation des Images benötigt wird.

Codezeilen im Software-Image (Größe der Datei)	Operationszeit (Prozent von der gemeinsamen Zeit)				
	Struktur	Signatur	Quellcode	Installation	Insgesamt
1 (< 1 kB)	6 ms (1 %)	21 ms (2 %)	943 ms (96 %)	10 ms (1 %)	980 ms
10 (1,5 kB)	7 ms (1 %)	25 ms (3 %)	929 ms (96 %)	11 ms (1 %)	972 ms
100 (10 kB)	13 ms (1 %)	53 ms (5 %)	1 031 ms (93 %)	14 ms (1 %)	1 111 ms
1 000 (89 kB)	78 ms (4 %)	91 ms (5 %)	1 644 ms (88 %)	51 ms (3 %)	1 864 ms
10 000 (880 kB)	920 ms (11 %)	789 ms (9 %)	6 000 ms (69 %)	951 ms (11 %)	8 660 ms
100 000 (8 790 kB)	4 809 ms (8 %)	6 718 ms (12 %)	41 698 ms (72 %)	4 953 ms (8 %)	58 178 ms

Tabelle 6.1: Rechenzeit der Installationsoperationen

Für die grafische Darstellung der Messwerte sind sie in der folgenden Abbildung präsentiert. Jeder Installationsoperation ist eine Kurve zugewiesen. Die x-Achse steht für die Größe des Software-Images. Die y-Achse präsentiert die Rechenzeit der Operationen.

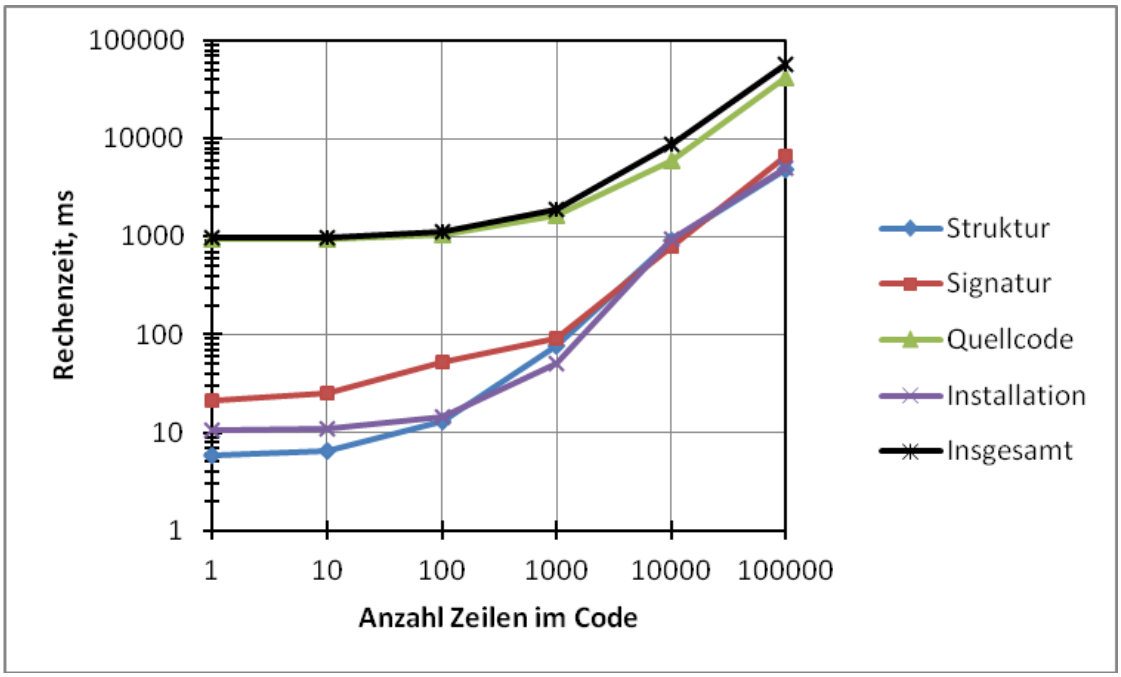

Abbildung 6.1: Rechenzeit der Installationsoperationen

6.3.2 Diskussion

Die Komplexität jedes Installationsschrittes wächst mit der Vergrößerung des Software-Images. Die gemessenen Zeitwerte zeigen asymptotisch eine lineare Abhängigkeit der Bearbeitungszeit von der Größe der Eingangsdaten (O(n)).

Wie das Experiment zeigt, ist die Analyse des Quellcodes die aufwändigste Operation und hat die größte Rechenzeit im Vergleich zu den anderen. Somit benötigt die Prüfung des Quellcodes bis zu 96 % von der gesamten Installationszeit für relativ kleine Software-Images und ca. 70 % für den Quellcode ab 10 000 Zeilen. Zu dieser Analyse zählt die Erkennung folgender Probleme: Syntaxfehler, unkorrekte Typkonvertierungen, die Verwendung nicht definierter Variablen, usw. Um solche Fehler zu entdecken, muss jede einzelne Zeile des Quellcodes analysiert werden. Für die Durchführung dieser Operationen wird der ganze Quellcodekontext verfolgt, was die hohe Komplexität erklärt.

7 Schlussfolgerungen

Das Ziel dieser Arbeit war die Präsentation eines Systems, das durch sein Design die Interoperabilität der Software für unterschiedliche IoT-Geräte unterstützen kann. Dabei gab es außer der Unterstützung der Software-Interoperabilität weitere wichtige Anforderungen an das System. Unter anderem waren es die Dynamik, in der Software-Updates zur Laufzeit durchgeführt werden müssen, die Sicherheit, dass zu installierende Software erst verifiziert werden muss und die Robustheit, die besagt, dass das System sowie das Gerät selbst ausfallsicher sein muss.

Für den Ansatz wurde eine passende Technologie-Auswahl getroffen, mit der den gesetzten Anforderungen genüge getan wurde. Während der Arbeit wurde eine Architektur einer Laufzeitumgebung für IoT-Plattformen entworfen, die eine dynamische Installation der Software auf IoT-Geräten anbietet. Das entwickelte Design ermöglicht, dieselbe Software auf unterschiedlichen Arten von IoT-Geräten auszuführen. Außerdem wurden in Rahmen der Arbeit Lösungen zur Sicherheit und Robustheit des Systems vorgeschlagen. Anhand der entwickelten Architektur wurde ein Prototyp der adaptiven Laufzeitumgebung implementiert und seine Funktionsweise mit Experimenten getestet.

In den Experimenten hat es sich gezeigt, dass nicht alle Probleme der Verwendung der Umgebung gelöst sind. Die diskutierte Laufzeitumgebung ermöglicht die Durchführung der Software-Updates unabhängig von der individuellen Hardware. Allerdings ist die Verwendung der Laufzeitumgebung nicht an jedem IoT-Gerät möglich. Die in der Arbeit vorgeschlagenen Lösungen fordern von den IoT-Geräten eine ausreichende Hardware-Leistung, um die Laufzeitumgebung zu installieren und ihre internen „teuren“ Prozesse, z.B. die Software-Validierungsprozesse, durchzuführen.

Auch zeigen die analysierten Experimente, dass die Verwendung der Umgebung potenzielle Laufzeitprobleme verursachen kann. Obwohl das System die Nebenläufigkeit der internen Prozesse hat, verlangsamen das parallel laufende Software-Update sowie die im Prototyp verwendeten Werkzeuge die Aus-

führung der Software. Das kann den Einsatz der Laufzeitumgebung an Geräten mit harten Deadlines verhindern.

Die erläuterten Nachteile fordern eine weitere Analyse dieser Probleme und eine Untersuchung alternativer Lösungen dafür. Allerdings kann die Integration der Laufzeitumgebung in existierende IoT-Plattformen die Verwaltung der Geräte erleichten. Denn die Entwicklung der Software für IoT-Geräte kann verallgemeinert werden. Das Problem der Software-Interoperabilität ist aktuell und muss weiter erforscht werden. Einer der weiteren Schritte kann die Publikation der Laufzeitumgebung als zu installierendes Packet in NPM sein. Somit können mehrere Entwickler die Laufzeitumgebung in eigenen Geräten/Projekten einsetzen und deren Funktionalität weiter untersuchen.

8 Literaturverzeichnis

[1] IEEE. Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated. URL: https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated (Zugriff am 11.02.2018).
[2] Kaa. IoT Use Cases. URL: https://www.kaaproject.org/iot-use-cases/ (Zugriff am 11.02.2018).
[3] ITU. Internet of Things Global Standards Initiative. URL: https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx (Zugriff am 11.02.2018).
[4] ReadWrite. Why The Internet Of Things Is Still Roadblocked. URL: http://readwrite.com/2014/08/04/internet-of-things-obstacles-roadblocks/ (Zugriff am 11.02.2018).
[5] JAXenter. Die Probleme des Internet der Dinge. URL: https://jaxenter.de/die-probleme-des-internet-der-dinge-801 (Zugriff am 11.02.2018).
[6] Bhumi Nakhuva, Prof. Tushar Champaneria. STUDY OF VARIOUS INTERNET OF THINGS PLATFORMS. International Journal of Computer Science & Engineering Survey (IJCSES) Vol.6, No.6, December 2015.
[7] NIST. Definition of Cloud Computing. URL: https://csrc.nist.gov/publications/detail/sp/800-145/final (Zugriff am 15.02.2018).
[8] Microsoft Azure. Übersicht über den Azure IoT Hub-Dienst. URL: https://docs.microsoft.com/de-de/azure/iot-hub/iot-hub-what-is-iot-hub (Zugriff am 15.02.2018).
[9] Microsoft Azure. Azure und das Internet der Dinge. URL: https://docs.microsoft.com/de-de/azure/iot-hub/iot-hub-what-is-azure-iot (Zugriff am 15.02.2018).
[10] Microsoft Azure. Unterstützen zusätzlicher Protokolle für IoT Hub. URL: https://docs.microsoft.com/de-de/azure/iot-hub/iot-hub-protocol-gateway (Zugriff am 15.02.2018).
[11] Microsoft Azure. Overview of device management with IoT Hub. URL: https://docs.microsoft.com/de-de/azure/iot-hub/iot-hub-device-management-overview (Zugriff am 16.02.2018).

[12] Amazon. Amazon AWS IoT Overview. URL: https://aws.amazon.com/de/iot-platform/how-it-works/ (Zugriff am 15.02.2018).
[13] Amazon. AWS IoT Core – Funktionen. URL: https://aws.amazon.com/de/iot-core/features/ (Zugriff am 15.02.2018).
[14] Amazon. AWS IoT Core. URL: https://aws.amazon.com/de/iot-core/ (Zugriff am 15.02.2018).
[15] ISO. ISO/IEC 20922:2016 Information technology -- Message Queuing Telemetry Transport (MQTT). URL: https://www.iso.org/standard/69466.html (Zugriff am 15.02.2018).
[16] Amazon. AWS IoT Device Management. URL: https://aws.amazon.com/iot-device-management/ (Zugriff am 16.02.2018).
[17] Arpan Pal, Balamuralidhar Purushothaman. IOT Technical Challenges and Solutions, S. 89. 2017, Artech House.
[18] IETF. Internet Engineering Task Force. URL: https://www.ietf.org/ (Zugriff am 18.02.2018).
[19] Brendan Moran, Milosch Meriac, Hannes Tschofenig. A Firmware Update Architecture for Internet of Things Devices. 29.01.2018. URL: https://datatracker.ietf.org/doc/draft-moran-suit-architecture (Zugriff am 18.02.2018).
[20] Suhas Nandakumar, Cullen Jennings, Shaun Cooley. Solution Architecture - Secure Firmware Upgrade. 30.10.2017. URL: https://datatracker.ietf.org/doc/draft-nandakumar-suit-secfu-solution-arch/ (Zugriff am 18.02.2018).
[21] IETF. Software Updates for Internet of Things. URL: https://datatracker.ietf.org/group/suit/about/ (Zugriff am 18.02.2018).
[22] Ramesh Bangia. Dictionary of Information Technology, S. 111. 2010, Firewal Media.
[23] Kent D. Lee. Foundations of Programming Languages, S. 21. 2017, Springer.
[24] R. Nageswara Rao, Kogent Solutions Inc. Core Java: An Integrated Approach: Covers Concepts, Programs and Interview Questions, S. 12. 2008, Dreamtech.
[25] David Flanagan. JavaScript: das umfassende Referenzwerk, 3. Auflage, S. 2. 2007, O'Reilly.

[26] Olaf Göllner. Tessel: JavaScript-Entwicklerboard fürs "Internet der Dinge". URL: https://www.heise.de/make/meldung/Tessel-JavaScript-Entwicklerboard-fuers-Internet-der-Dinge-1936379.html (Zugriff am 25.02.2018).
[27] Node.js. About The Node.js Foundation. URL: https://foundation.nodejs.org/about (Zugriff am 25.02.2018).
[28] Yaacov Apelbaum. User Authentication Principles, Theory and Practice, S. 14. 2004, Technology Press.
[29] IETF Network Working Group. PPP Authentication Protocols RFC 1334. URL: https://tools.ietf.org/html/rfc1334 (Zugriff am 03.03.2018).
[30] Marko Pfeiffer. Auswirkung der Umsetzung der EU-Richtlinie „Digitale Signatur“ auf Geschäftsprozesse im Rechnungswesen, S. 4. 2001, Diplomica Verlag GmbH.
[31] IETF Network Working Group , T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159. 2014.
[32] Tim Bray et al. “Extensible markup language (XML).” In: World Wide Web Journal 2.4 (1997), pp. 27–66.
[33] NPM. URL: https://www.npmjs.com/ (Zugriff am 24.04.2018).
[34] IBM. RESTful Web services: The basics. URL: https://www.ibm.com/developerworks/webservices/library/ws-restful/ (Zugriff am 09.03.2018).
[35] IBM. Create RESTful Web services with Java technology. URL: https://www.ibm.com/developerworks/webservices/library/wa-jaxrs/index.html (Zugriff am 09.03.2018).
[36] IETF. RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1, S. 52-57. URL: https://www.ietf.org/rfc/rfc2616.txt (Zugriff am 09.03.2018).
[37] Node.js. About Node.js. URL: https://nodejs.org/en/about/ (Zugriff am 29.04.2018).
[38] W3C. XML Schema Definition Language (XSD). URL: https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ (Zugriff am 24.04.2018).
[39] Node.js. Node.js v10.0.0 Documentation. URL: https://nodejs.org/api/vm.html (Zugriff am 01.05.2018).

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 14.06.2018 *Anton Dementyev*