Universität Rostock Traditio et Innovatio

Deterministische und interoperable Kommunikation mit dem Constrained Application Protocol

Der Fakultät für Informatik und Elektrotechnik
der Universität Rostock zur Erlangung des
akademischen Grades eines

Dr.-Ing.

vorgelegte Dissertation von
Björn Konieczek

Gutachter:
Prof. Dr.-Ing. Dirk Timmermann
Institut für Angewandte Mikroelektronik
und Datentechnik
Universität Rostock
Richard Wagner Str. 31
18119 Rostock-Warnemünde

Zweitgutachter:
Prof. Dr. Wolfgang Kastner
Institut für Computer Engineering
Technische Universität Wien
Treitlstraße 3
1040 Wien
Österreich

https://doi.org/10.18453/rosdok_id00002579

Dissertation
Fakultät für Informatik und Elektrotechnik

Gutachter:

- Prof. Dr.-Ing. Dirk Timmermann (Universität Rostock, Fakultät für Informatik und Elektrotechnik, Institut für Angewandte Mikroelektronik und Datentechnik)

- Prof. Dr. Wolfgang Kastner (Technische Universität Wien, Fakultät für Informatik, Institut für Computer Engineering)

Tag der Einreichung: 05.06.2019
Tag der Verteidigung: 08.11.2019

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Rostock, den 05.06.2019

_____

Björn Konieczek

Zusammenfassung

In der jüngeren Vergangenheit konnten erhebliche Fortschritte bei Protokollen und Anwendungen für das Internet der Dinge (Internet of Things - IoT) erzielt werden. Neueste Bestrebungen zielen darauf ab, IoT-Technologien auch auf andere Bereiche, wie die Industrieautomation oder Medizintechnik, zu übertragen. Diese Entwicklung wird auch als vierte industrielle Revolution, Industrie 4.0 oder Industrial Internet of Things (IIoT) bezeichnet. Diese neuen Anwendungsgebiete stellen jedoch auch neue Anforderungen an die im IoT bewährten Technologien. Einen Kernpunkt bildet dabei die Echtzeitfähigkeit, also die zeitliche Vorhersagbarkeit der Interaktionen zwischen den Geräten. Bisher werden Feldbus-Systeme oder echtzeitfähige Ethernet-Lösungen (Industrial Ethernet - IE) genutzt, um diesen Anforderungen gerecht zu werden. Diese basieren jedoch mehrheitlich auf proprietärer Hardware, welche zu hohen Anschaffungskosten und der Bindung an einen einzelnen Hersteller führen (sog. Vendor Lock-in).

In dieser Arbeit wird daher untersucht, inwieweit sich Echtzeitanforderungen mit einer rein Software-basierten Lösung auf Basis etablierter IoT-Standards unter Verwendung von Standardhardware erfüllen lassen. Hierzu werden zunächst mehrere IoT-Protokolle bezüglich ihrer prinzipiellen Eignung für Echtzeitszenarien gegenüber gestellt. Im Ergebnis dieser Untersuchung hat sich das Constrained Application Protocol (CoAP) als besonders geeignet erwiesen. Anschließend wird schrittweise gezeigt, wie sich eine zeitlich deterministische Kommunikation allein auf Basis des CoAP-Standards erreichen lässt. Dabei wird die Wirksamkeit der erarbeiteten Verfahren mit verschiedenen Experimenten in einer Testumgebung mit realen Geräten belegt. Abschließend wird aus den beschriebenen Mechanismen eine eigene Standarderweiterung für CoAP abgeleitet.

Abstract

In the recent past, significant progress has been made in protocols and applications for the Internet of Things (IoT). Newest efforts aim at transferring IoT technologies to other areas, such as industrial automation or medical applications. This development is also referred to as the fourth industrial revolution, Industry 4.0 or Industrial Internet of Things (IIoT). However, these new areas of application also introduce new demands on the IoT technologies. A key requirement is the real-time capability of the interactions between the devices. Until now, fieldbus systems or real-time Ethernet solutions (Industrial Ethernet - IE) have been used to meet these requirements. However, the majority of these solutions is based on proprietary hardware, which leads to high acquisition costs and encourages vendor lock-in (binding to a single manufacturer).

Therefore, this paper examines the extent to which real-time requirements can be met with a purely software-based solution based on established IoT standards using standard hardware. In the beginning, several IoT protocols will be compared with regard to their suitability for environments with real-time requirements. As a result of this investigation, the Constrained Application Protocol (CoAP) has proven to be particularly suitable. Subsequently, it is shown step by step how a temporally deterministic communication can be achieved solely by utilizing standard CoAP mechanisms. The effectiveness of the developed methods will be proven with different experiments in a test environment with real devices. Finally, a separate standard extension for CoAP is derived from the described mechanisms.

Danksagung

Diese Arbeit entstand maßgeblich während meiner Zeit als Stipendiat in dem DFG-geförderten Graduiertenkolleg MuSAMA der Universität Rostock zwischen Mai 2013 und August 2016. Auch wenn es sich bei einer Dissertation gemeinhin um den Nachweis der Fähigkeit zum eigenständigen wissenschaftlichen Arbeiten handelt, so haben mich bei der Entstehung dieser Arbeit viele Menschen begleitet und auf unterschiedliche Art und Weise unterstützt. Ich möchte diese Gelegenheit nutzen, um diesen Personen meinen innigsten Dank auszusprechen.

Großer Dank gilt meinem Doktorvater Herrn Prof. Dr.-Ing. Dirk Timmermann, der diese Arbeit durch seine Betreuung im Rahmen des Graduiertenkollegs MuSAMA erst möglich gemacht hat. Er war jederzeit zu intensivem Austausch bereit. Von seiner Anregung und Kritik konnte ich stets profitieren.

Weiterhin gilt mein Dank Herrn Prof. Dr. Wolfgang Kastner von der TU Wien für sein Engagement als Zweitgutachter.

Darüber hinaus möchte ich den Mitarbeitern des Instituts für Angewandte Mikroelektronik und Datentechnik der Universität Rostock danken, die nicht nur durch fachliche Diskussionen dafür gesorgt haben, dass ich meine Promotionszeit in schöner Erinnerung behalten werde. Dies gilt besonders für meinen langjährigen Freund und Kollegen Michael Rethfeldt. Er hat nicht nur fachlich mit seinen kritischen Anmerkungen und konstruktiven Diskussionsbeiträgen, sondern auch persönlich einen großen Beitrag zum Gelingen dieser Arbeit geleistet.

Mein ganz besonderer Dank gilt meiner Ehefrau Dr. jur. Angela Konieczek, die mit ihrer ständigen Unterstützung, Liebe und Fürsorge entscheidend zum Gelingen dieses Promotionsvorhabens beigetragen hat. Sie half mir mit ihrer positiven Art und ihrer Fähigkeit, andere zu begeistern durch viele schwierige Momente und hielt mir in entscheidenden Situationen den Rücken frei.

Schlussendlich möchte ich auch meinen Eltern Thorolf Konieczek und Petra Konieczek sowie meinen Schwiegereltern Wolfgang Westphal und Cornelia Westphal für ihre Unterstützung in allen Lebenslagen danken.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

„The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.“

- Mark Weiser -

Mit diesen Worten beschrieb Mark Weiser 1991 seine Vision vom Ubiquitous Computing, die er schon 1988 erstmals formulierte [109]. Seitdem findet eine technische Entwicklung statt, die eine Erfüllung dieser Vision immer greifbarer macht. Die Strukturgrößen und Fertigungskosten integrierter Systeme sind stetig gesunken. Daraus ergibt sich die Möglichkeit, unsere Alltagsumgebung immer weiter mit elektronischen Helfern auszustatten. Die meisten dieser Helfer verfügen allerdings nur über sehr begrenzte Ressourcen in Bezug auf Rechenleistung, Speicher oder Energievorrat. Diese Ressourcenlimitierung ergibt sich aus den harten Kosten- und Größenanforderungen, welche das Verschmelzen der Geräte mit unserer Umgebung erst ermöglichen. Aus diesen Einschränkungen wurde die Idee des Internets der Dinge (Internet of Things - IoT) geboren. Hierbei werden die ressourcenbeschränkten Geräte miteinander vernetzt. Auf diese Weise wird es möglich, die Geräte auf eine einzelne oder eine kleine Menge von Aufgaben zu spezialisieren. Um komplexe Aufgaben zu erfüllen, schließen sich die Geräte zu sogenannten Ensembles zusammen, in denen sie miteinander interagieren und kommunizieren [33]. Eine solche komplexe Aufgabe stellt beispielsweise die Intentionserkennung im Ambient Assisted Living (AAL) dar. Dabei wird das Verhalten des Nutzers von einer Vielzahl von Sensoren in seiner Umgebung beobachtet. Auf Grundlage dieser Beobachtungen werden Schlüsse auf das gewünschte Handlungsziel, die Intention des Nutzers, gezogen. Die Erkenntnisse aus der Intentionserkennung werden anschließend verwendet, um einen Aktionsplan für die Geräte in der Umgebung zu erstellen, sodass diese das Erreichen des Handlungsziels für den Nutzer möglichst stark vereinfachen [43].

Um die spontane Bildung dieser Ensembles und die Interaktion zwischen den Geräten zu ermöglichen, stützt sich das IoT auf Netzwerk- und Anwendungsprotokolle, die drei wesentliche Eigenschaften erfüllen. Zum einen müssen sie sehr leichtgewichtig sein, um auch ressourcenschwachen Geräten die Teilnahme an den Ensembles zu ermöglichen. Zum anderen müssen sie eine einfache Integrierbarkeit in bereits bestehende Netzwerke erlauben. Den Protokollen sollten also weit verbreitete Standards zugrunde liegen, um die Entstehung von Insellösungen zu vermeiden. Von besonderer Bedeutung ist darüber hinaus das Sicherstellen der Interoperabilität zwischen den Geräten im Sinne der Maschine-zu-Maschine-Kommunikation (M2M) [6]. Um eine hohe Kooperationsfähigkeit zu realisieren, haben sich zwei Ansätze als besonders vielversprechend herausgestellt. Der erste Ansatz nutzt

Service-Orientierte Architekturen (SOA) auf der Grundlage von Web Services. Web Services bieten durch ihre Flexibilität optimale Voraussetzungen. Jedoch verfügen eingebettete Systeme nur über wenig leistungsfähige Hardware und meist begrenzten Energievorrat, sodass die bereits aus dem Internet bekannten und bewährten Protokolle und Service-Architekturen an diese Gegebenheiten angepasst werden müssen. Das Devices Profile for Web Services (DPWS) bietet eine mögliche Grundlage hierfür. Eine andere Möglichkeit, die Kooperationsfähigkeit von Systemen zu erhöhen, bieten sogenannte RESTful-Architekturen (Representational State Transfer - REST). Das Constrained Application Protocol (CoAP) versucht die Kernpunkte von RESTful-Architekturen auf ressourcenbeschränkte eingebettete Systeme zu übertragen.

Jüngste Bemühungen zielen darauf ab, die Technologien aus dem IoT auch in andere Anwendungsbereiche zu übertragen. Diese neuen Felder umfassen auch zeitkritische Bereiche wie die Industrieautomation. Dieser Prozess wird auch als vierte industrielle Revolution, Industrie 4.0 oder Industrial Internet of Things (IIoT) bezeichnet. Grund für diese Entwicklung ist die zu erwartende Flexibilitätssteigerung bei der Zusammenstellung von Produktionsstraßen (Horizontale Integration). Eine weitere Motivation stellt das Einbeziehen der Geräte auf der Feldebene (z.B. Fertigungsroboter in der Fabrikhalle) in Geschäftsprozesse auf höheren Ebenen dar (Vertikale Integration) [6, 54]. Diese Integrationsziele sind in Abbildung 1 illustriert. Das langfristige Ziel dieser Entwicklung ist das Erreichen der Losgröße 1. Dies bedeutet, dass speziell an die Kundenwünsche angepasste Produkte als Einzelstücke gefertigt werden können, ohne dass aufwändig manuelle Veränderungen an der Fertigungsstrecke vorgenommen werden müssen.

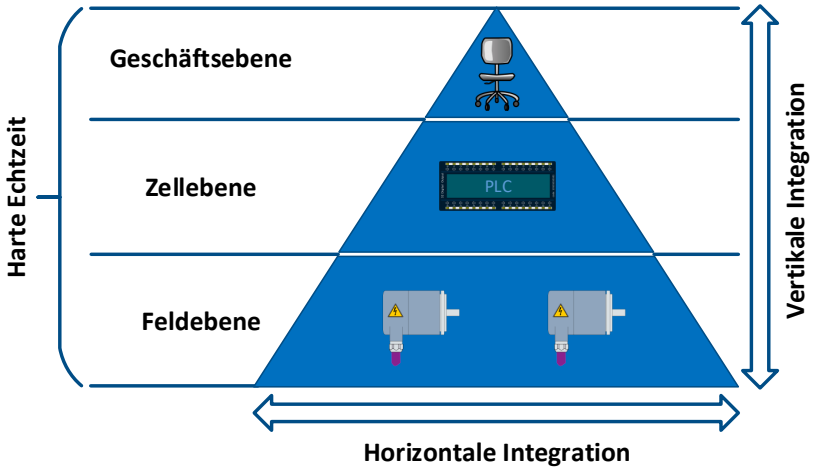

Abbildung 1: Integrationsziele im Rahmen der „Industrie 4.0“-Bewegung.

Neben der Industrieautomation halten IoT-Technologien auch in andere zeitkritische Bereiche, wie die Medizintechnik, Einzug [82]. Hier soll die Interoperabilität aus dem IoT dazu führen, Medizingeräte flexibler in einem OP-Saal zusammenstellen zu können. Darüber hinaus soll ein sogenannter

Vendor Lock-in, die Abhängigkeit von einem bestimmten Hersteller, verhindert werden, indem auf die Nutzung proprietärer Lösungen verzichtet wird.

Diese neuen Einsatzbereiche stellen zusätzliche Anforderungen an die IoT-Protokolle, die bisher keine Berücksichtigung finden. Ein elementarer Bestandteil dieser Anforderungen ist die Echtzeitfähigkeit (engl. Real Time, RT). Man spricht von echtzeitfähigen Systemen, wenn die Einhaltung einer oberen Zeitschranke für das einmalige Durchlaufen der Daten vom Eingang bis zum Ausgang des Systems garantiert werden kann. Diese Zeitschranken werden auch als Deadlines bezeichnet. Das Verletzen dieser Garantien kann in kritischen Bereichen, wie der Industrieautomation oder der Medizintechnik, weitreichende Folgen haben, von Vermögensverlusten bis hin zu Personenschäden. Um die Angabe von zeitlichen Garantien zu ermöglichen, muss sich das System zeitlich deterministisch verhalten, also vorhersagbar sein. Auf Grund der verteilten Natur von IoT-Anwendungen wird deren Ausführungszeit stark von der Kommunikation zwischen den Geräten beeinflusst. Daher genügt es nicht, wenn sich jedes Gerät für sich deterministisch verhält. Vielmehr muss die Kommunikation in die Betrachtungen miteinbezogen werden.

1.2 Problemstellung

Bisherige Lösungsansätze beschränken sich bei der Betrachtung des Problems der Echtzeitkommunikation auf die unteren Schichten des ISO/OSI-Schichtenmodells (vgl. Abbildung 2). Dies führt zu erheblichen Nachteilen, welche dem Gedanken des IoT und den Zielen der „Industrie 4.0“-Bewegung entgegenstehen. Das Grundproblem besteht dabei darin, dass die entstandenen Lösungen auf proprietärer Hardware basieren oder nicht-standardkonforme Veränderungen an den Netzwerkprotokollen auf den unteren Schichten vornehmen. Beispiele hierfür sind die in der Industrieautomation weit verbreiteten Feldbus-Systeme und „Industrial Ethernet“-Lösungen (IE).

Im Ergebnis führen die Protokollanpassungen und speziellen Hardwareanforderungen zu Inkompatibilitäten sowohl zwischen den einzelnen Echtzeit-Lösungen als auch mit weit verbreiteten Standards in der Netzwerktechnik. Dies begünstigt einerseits einen Vendor Lock-in. So entstehen hohe Kosten bei der Anschaffung und Einrichtung der benötigten Infrastruktur. Außerdem wird die Flexibilität bei der Auswahl anzuschaffender Geräte erheblich eingeschränkt, da diese die nötigen proprietären Schnittstellen bereitstellen müssen. Andererseits ergibt sich daraus die Teilung in unterschiedliche Kommunikationsdomänen, zwischen denen eine Interaktion nur schwer möglich ist. Dies steht dem zentralen Interesse der totalen horizontalen und vertikalen Integration im Wege. Abbildung 3 stellt diese Problematik grafisch dar.

Weit verbreitete Standard-Technologien, wie beispielsweise Ethernet, erlauben allerdings keine deterministische Kommunikation. Die Gründe hierfür sind das Zwischenpuffern von Nachrichten inner-

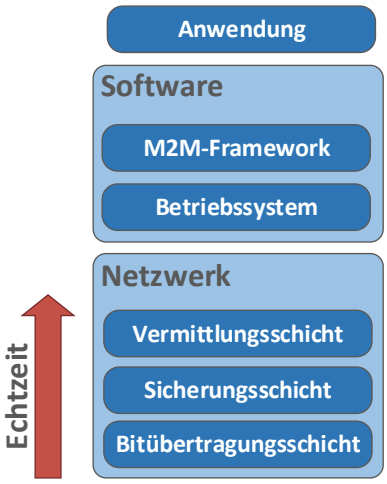

Abbildung 2: Schematische Darstellung des bisher verfolgten Bottom-Up-Ansatzes zur Echtzeitkommunikation.

halb von Infrastruktrurgeräten, wie Switches, sowie die beschränkte Bandbreite der Switches. Dies führt zu einem nicht-vorhersagbaren Zeitverhalten bei der Kommunikation.

1.3 Zielstellung

Die bisherigen Lösungsansätze für die Problematik der Echtzeitkommunikation können die Anforderungen zukünftiger IoT-Szenarien nicht erfüllen. Sie sind durch ihren hohen Konfigurationsaufwand zu unflexibel und kostenintensiv in der Anschaffung. Die Verwendung spezieller Hardware oder angepasster Protokolle auf den unteren Schichten führt zudem zu Problemen bei der Integration in bereits bestehende Netzwerke. Um diese Probleme zu umgehen und trotzdem ein zeitlich deterministisches Kommunikationsverhalten zu ermöglichen, wird in dieser Arbeit ein rein Software-basierter Top-Down-Ansatz untersucht (vgl. Abbildung 4).

Auf diese Weise soll zum einen die Flexibilität erhalten bleiben, die von den im IoT verwendeten Protokollen bereitgestellt wird. Zum anderen soll die Integrierbarkeit verbessert werden, da so auf spezielle Hardware verzichtet werden kann. Die Verwendung von Commercial-Off-The-Shelf (COTS) Hardware führt außerdem zu erheblich niedrigeren Anschaffungskosten für die Netzwerkinfrastruktur. Darüber hinaus können per se nicht-echtzeitfähige Geräte durch ein einfaches Software-Update in das Echtzeitnetz integriert werden. Mit dem HaRTKad-Framework, welches als Middleware Echtzeitkommunikation über Standardhardware erlaubt, existiert bereits ein ähnlicher Lösungsansatz [104]. Jedoch definiert HaRTKad keine Schnittstellen zu übergeordneten Standardprotokollen des IoT und erschwert somit die Integration.

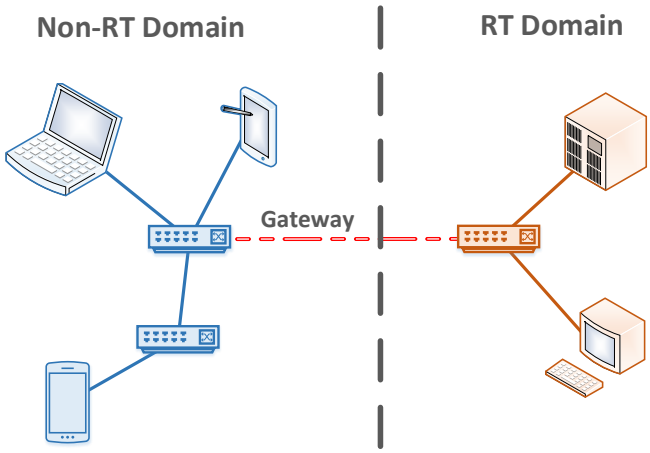

Abbildung 3: Aufspaltung industrieller Netzwerke in zwei unterschiedliche Kommunikationsdomänen für echtzeitfähige und nicht-echtzeitfähige Kommunikation.

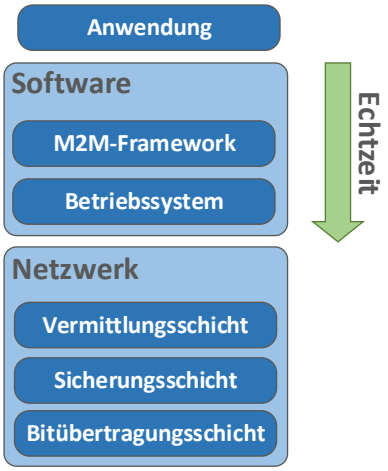

Abbildung 4: Schematische Darstellung des in dieser Arbeit vorgeschlagenen Top-Down-Ansatzes.

Daher wird in dieser Arbeit eine direkte Integration der Echtzeitfähigkeit in ein Standardprotokoll des IoT angestrebt. Die Basis hierfür bilden die eingangs eingeführten Protokolle DPWS und CoAP. Diese werden zunächst hinsichtlich ihrer Eignung für den Einsatz in Echtzeitszenarien gegeneinander abgewogen. Darauf aufbauend soll eines dieser Protokolle ausgewählt und so erweitert werden, dass ein deterministischer Datenaustausch über Standard-Technologien ermöglicht wird. Die notwendigen Änderungen und Erweiterungen werden in eine Echtzeit-Standarderweiterung für das gewählte Protokoll überführt. Die Umsetzung der entwickelten Mechanismen wird in einer plattfomunabhängigen Softwarelösung erprobt, um der heterogenen Gerätearchitektur im IIoT gerecht zu werden und eine größtmögliche Flexibilität zu erreichen.

1.4 Aufbau

Nachfolgend wird der Aufbau dieser Arbeit beschrieben und es werden die einzelnen Kapitel erläutert. In Kapitel 2 werden die nötigen Grundlagen im Bereich der Web Services dargelegt. Darüber hinaus werden die beiden Architektur-Konzepte SOA und REST näher erläutert und gegeneinander abgewogen. Hierbei werden die Vor- und Nachteile beider Ansätze tiefergehend betrachtet und deren Tauglichkeit für den Einsatz in Echtzeitsystemen geprüft. Abschließend wird CoAP als REST-basierte Lösung für IoT-Szenarien und M2M-Anwendungen ausführlich beschrieben. Das dritte Kapitel setzt sich mit bestehenden Ansätzen zur echtzeitfähigen Gerätekommunikation auseinander. Hierbei werden zunächst die für das weitere Verständnis nötigen Grundlagen beschrieben. Anschließend wird die Entwicklung von einfachen Feldbus-Systemen hin zu Industrial-Ethernet-Lösungen dargelegt. Weiterhin wird besonders auf das HaRTKad-Framework, eine Peer-to-Peer-(P2P)-basierte Lösung zur Echtzeitkommunikation, eingegangen. Dabei werden auch die bestehenden Limitierungen von HaRTKad, technische Probleme und einige Lösungsmöglichkeiten diskutiert. Darauf aufbauend wird ein Alternativansatz auf der Grundlage von CoAP mit dem Fokus auf Standardkonformität und Plattformunabhängigkeit entwickelt. Kapitel 4 beschreibt den Aufbau des jCoAP-Frameworks, einer plattformunabhängigen Java-Implementierung von CoAP. In diesem Zusammenhang wird untersucht, inwieweit jCoAP mit anderen verfügbaren CoAP-Umsetzungen kompatibel ist. Dabei wird auch das Zeitverhalten von jCoAP betrachtet und mit anderen Implementierungen verglichen. In Kapitel 5 werden Modifikationen sowohl im Protokoll- und Software-Stack als auch in jCoAP selbst untersucht, mit deren Hilfe sich eine zeitlich deterministische Protokollverarbeitung erreichen lässt. In diese Betrachtungen wird neben dem Betriebssystem und jCoAP auch die Java Virtual Machine (JVM) miteinbezogen. Anschließend werden in Kapitel 6 Mechanismen für einen zeitlich deterministischen Datenaustausch mit CoAP beschrieben. Neben der Integration dieser Mechanismen in jCoAP werden zudem die Ergebnisse einer praktischen Untersuchung in einer Testumgebung mit realen Geräten präsentiert. Kapitel 7 erläutert das Konzept und die Umsetzung eines verteilten Zeit-Servers mit CoAP. Ziel der Verteilung der Zeit-Server-Funktionalität ist hierbei, die Skalierbarkeit und die Robustheit der in Kapitel 6 beschriebenen Mechanismen zu erhöhen. Die Vorteile des verteilten Zeit-Servers werden mit Hilfe von Messwerten aus einer realen Testumgebung belegt. Kapitel 8 fasst die in den beiden vorangegangen Kapiteln beschriebenen Ansätze zusammen und leitet daraus eine ganzheitliche Echtzeiterweiterung für den CoAP-Standard ab. Kapitel 9 fasst die Erkenntnisse dieser Arbeit zusammen und diskutiert die erzielten Ergebnisse. Darüber hinaus werden weitere Ansätze für zukünftige Forschungsarbeiten aus den Ergebnissen abgeleitet.

Abbildung 5 beschreibt den Aufbau dieser Arbeit grafisch. Hierbei werden reine Grundlagenkapitel in der Farbe Weiß dargestellt, während Abschnitte, in denen eigene Entwicklungen vorgestellt werden, in blau unterlegt sind.

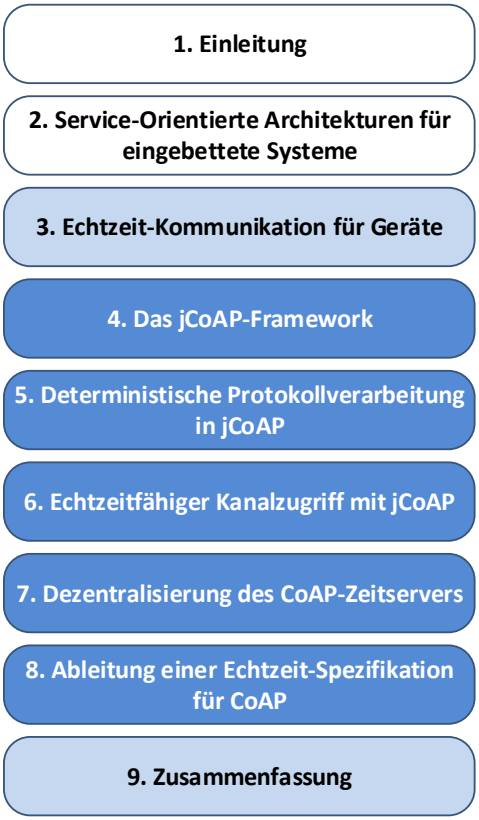

Abbildung 5: Grafische Aufbereitung der Struktur dieser Arbeit.

2 Service-Orientierte Architekturen für eingebettete Systeme

2.1 Grundlagen

2.1.1 ISO/OSI-Referenzmodell

Das Open Systems Interconnection (OSI) Referenzmodell wurde 1984 erstmals von der International Organization for Standardization (ISO) als Standard veröffentlicht und beschreibt die Vernetzung von Computern auf allgemeingültige Art und Weise [89]. Dabei teilt es die Netzwerkkommunikation in sieben Schichten ein, von denen jeder eine bestimmte Teilaufgabe bei der Kommunikation zugeordnet wird [106]. Diese Aufteilung führt zu einer Kapselung der einzelnen Aufgabenbereiche, sodass für jede Schicht Lösungen entworfen werden können, die jeweils von der Umsetzung der anderen Schichten unabhängig sind. Parallel zum OSI-Referenzmodell existiert das Internet-Modell, das die oberen drei Schichten zu einer zusammenfasst [40]. Diese Zusammenfassung ergibt sich aus der Tatsache, dass die Aufgaben dieser Schichten bei der Kommunikation im Internet üblicherweise von einer einzelnen Anwendung, z.B. einem Browser, übernommen werden. Abbildung 6 stellt die beiden Modelle gegenüber. Nachfolgend werden die einzelnen Schichten des OSI-Referenzmodells von unten nach oben näher erläutert.

Abbildung 6: Das ISO/OSI-Referenzmodell (links) und das Internet-Modell (rechts).

Bitübertragungsschicht

Die Bitübertragungsschicht bildet die Schnittstelle zwischen der Computerhardware und dem Kommunikationsmedium [97]. Neben den Schnittstellen beschreibt sie, wie ein eingehender Bitstrom aufzubereiten ist, sodass er über das Medium zum Empfänger übertragen werden kann. Hierzu zählen

u.a. die Darstellung der einzelnen Bits auf dem Medium in Form elektrischer (drahtgebunden) oder optischer Signale (Lichtwellenleiter) oder elektromagnetischer Wellen (drahtlos).

Sicherungsschicht

Die Sicherungsschicht soll eine zuverlässige Verbindung über das Kommunikationsmedium ermöglichen. Hierzu wird der Datenstrom in kleinere Blöcke, sogenannte Frames, unterteilt [106]. Zusätzlich können auf dieser Schicht Mechanismen zur Fehlerkontrolle, wie beispielsweise Prüfsummen, umgesetzt werden. Auf diese Weise ist es für den Empfänger möglich, verlorengegangene oder fehlerhafte Frames zu erkennen oder die Fehler sogar zu korrigieren. Darüber hinaus ermöglicht die Sicherungsschicht eine Adressierung der Geräte, sodass der Sender und der Empfänger eindeutig identifiziert werden können. Weiterhin wird auf dieser Schicht der Zugriff auf das Kommunikationsmedium kontrolliert. Wollen mehrere Sender gleichzeitig auf dasselbe Medium zugreifen, wird hier entschieden, welches Gerät senden darf.

Vermittlungsschicht

In Netzwerken sind Sender und Empfänger nur in den seltensten Fällen direkt miteinander verbunden. Üblicherweise muss der Datenstrom über mehrere Zwischenstationen oder -netze geleitet werden. Die Aufgabe der Vermittlungsschicht ist es dabei, sicherzustellen, dass der Datenstrom beim richtigen Empfänger ankommt [97]. Hierfür definiert die Vermittlungsschicht übergeordnete logische Adressen. Des Weiteren fällt das Routing der Daten in den Aufgabenbereich dieser Schicht. Als Routing bezeichnet man dabei die Entscheidung, an welche nächste Zwischenstation auf dem Weg zum Empfänger die Daten weitergeleitet werden sollen.

Transportschicht

Die Hauptaufgabe der Transportschicht ist die Ende-zu-Ende-Kontrolle der Kommunikation zwischen den Prozessen auf den anwendungsorientierten Schichten 5-7 [89]. Da auf jedem System mehrere Prozesse ausgeführt werden können, die über das Netzwerk kommunizieren wollen, wird jedem Prozess ein Port zur Adressierung zugewiesen. Anhand des Ports können eingehende Pakete eindeutig einem Prozess zugeordnet werden. Für den Datenaustausch stehen zwei Kommunikationsarten zur Verfügung, die paketorientierte (verbindungslose) und die verbindungsorientierte Kommunikation. Bei dem paketorientierten Datenaustausch wird jede Dateneinheit als unabhängiges Paket gesendet, wohingegen bei der verbindungsorientierten Kommunikation zunächst eine zuverlässige Verbindung aufgebaut wird [67]. Anschließend werden die Daten übertragen. Nachdem die Datenübertragung abgeschlossen ist, wird die Verbindung wieder abgebaut. Weiterhin ermöglicht die Transportschicht die Segmentierung der zu versendenden Daten. Dabei werden die Daten in mehrere Segmente unterteilt, die in Einzelpaketen verschickt werden. Auf der Empfänger-Seite findet dann die Re-Assemblierung statt, bei der die einzelnen Segmente wieder zu den ursprünglichen Daten zusammengefügt werden. Hierbei spielt die Reihenfolge der Segmente eine wichtige Rolle. Zu den weiteren Teilaufgaben der

Transportschicht zählen ebenfalls die Fehlererkennung und die Flusskontrolle, die sich anders als auf den unteren Schichten nur auf die Kommunikation zwischen den Prozessen beziehen und nicht auf die physikalische Verbindung [97].

Sitzungsschicht

Die Sitzungsschicht dient dazu, die Interaktionen zwischen zwei Prozessen zu koordinieren [89]. Hierzu zählen der Aufbau und die Erhaltung einer stabilen Sitzung sowie die Synchronisation zwischen den Teilnehmern. Die Synchronisation dient dazu, festgelegte Punkte für die Wiederaufnahme der Kommunikation zu schaffen, falls es während der Interaktion zu Verbindungsabbrüchen kommt [106].

Darstellungsschicht

Die Darstellungsschicht bereitet die empfangenen Daten so auf, dass sie von der Anwendungsschicht gelesen und interpretiert werden können. Dies kann beispielsweise auch eine Konvertierung der Daten erforderlich machen. Darüberhinaus fallen auch die Ver- und Entschlüsselung sowie die Kompression und De-Kompression der Daten in den Aufgabenbereich dieser Schicht [67].

Anwendungsschicht

Die Anwendungsschicht stellt die Verbindung zu den unteren Schichten her und ermöglicht so die Kommunikation zwischen zwei Anwendungen. Zu ihren Aufgaben gehören des Weiteren die Ein- und Ausgabe von Daten sowie deren Verarbeitung.

2.1.2 Ausgewählte Kommunikationsprotokolle

Um die Funktionalitäten der einzelnen Schichten des ISO/OSI-Referenzmodells zu erfüllen, wurde eine Vielzahl von Ablaufbeschreibungen für die Kommunikation, sogenannte Protokolle, spezifiziert. Die verschiedenen Protokolle lassen sich in der Regel eindeutig einer der Schichten im Referenzmodell zuordnen. Bei der Kommunikation zwischen zwei Anwendungen fügt jedes der Protokolle auf den einzelnen Schichten den Nutzdaten eigene Steuerinformationen in Form eines Headers hinzu. Nachfolgend werden einige ausgewählte Protokolle näher erläutert.

Internet Protocol (IP)

Das Internet Protocol ist auf der Vermittlungsschicht des ISO/OSI-Referenzmodells anzusiedeln. Seine Hauptfunktion besteht in der logischen Adressierung der Netzwerkteilnehmer, anhand derer das Routing der Netzwerkpakete zwischen den Teilnehmern ermöglicht wird. Es werden aktuell zwei Versionen des IP genutzt: das ältere IPv4 und das IPv6, welches langfristig IPv4 ersetzen soll. Bei IPv4 werden die Geräte über einen 32-Bit-Wert adressiert (IP-Adresse). Dieser ist in einen Netzwerk-Teil und einen Host-Teil untergliedert. Dabei identifiziert der Netzwerk-Teil ein bestimmtes Sub-

netz, während der Host-Teil einem bestimmten Host innerhalb dieses Subnetzes zugeordnet ist. Wie viele Bits der Adresse den unterschiedlichen Teilen der IP-Adresse jeweils zuzuordnen sind, wird durch die Netzmaske beschrieben. Der Netzwerk-Teil der IP-Adresse wird dabei üblicherweise von einer übergeordneten Instanz wie dem Internet Service Provider (ISP) zugewiesen. Der Host-Teil kann hingegen lokal frei gewählt beziehungsweise vom Administrator festgelegt werden. Neben der Adressierung erlaubt IP die Fragmentierung und Re-Assemblierung der zu versendenden Daten. Dies ermöglicht die Nutzung von Kommunikationstechnologien mit unterschiedlichen maximalen Paketgrößen (Maximum Transmission Unit, MTU) auf den unteren Schichten [106]. Abbildung 7 zeigt den Aufbau eines IPv4-Headers.

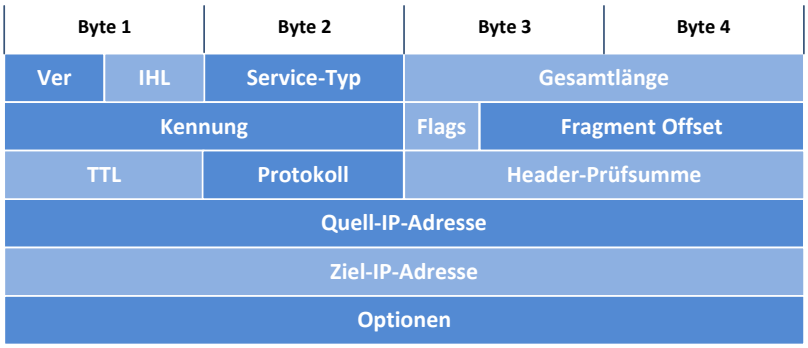

Abbildung 7: Aufbau des IPv4-Headers [2].

Der Header enthält die Version des Internet-Protokolls, nach der das Paket erstellt wurde sowie die Länge des IP-Headers (IHL). In dem Service-Typ-Feld können Quality of Service (QoS)-Parameter gesetzt werden. Weiterhin enthält der IP-Header die Gesamtlänge des Datenpaketes. Die Kennung dient der Identifizierung von Datenpaketen und enthält einen Zahlenwert, der fortlaufend für jedes Paket erhöht wird. Das TTL-Feld (Time to live) gibt die Lebensdauer eines Paketes in Hops (Sprünge beziehungsweise Weiterleitungen) an. Jede Station, die ein Paket weiterleitet, zieht eins von dem Wert des TTL-Feldes ab. Auf diese Weise soll verhindert werden, dass nicht-zustellbare Pakete unendlich lange im Netzwerk zirkulieren. Weiterhin enthält der Header eine Kennung für das verwendete Transportprotokoll und eine Prüfsumme zur Kontrolle der Datenintegrität. Anschließend folgen die Quell- und Ziel-IP-Adressen und ein Optionen-Feld mit variabler Länge.

Auf Grund der Entwicklung hin zum Internet der Dinge ist die Anzahl der direkt mit dem Internet verbundenen Geräte in den letzten Jahren stark angestiegen. Dieser Trend wird sich Schätzungen zufolge zukünftig noch verstärken [32]. Dies führt dazu, dass nicht mehr allen Geräten eine IPv4-Adresse zugewiesen werden kann, da die Anzahl der verfügbaren Adressen auf ca. 4,2 Mrd. beschränkt ist. Aus dieser Motivation heraus wurde die neuere Variante des Internet-Protokolls, IPv6, spezifiziert [27]. Bei IPv6 wird die Breite der IP-Adressen auf 128 Bit erhöht, womit die Anzahl der verfügbaren IP-Adressen auf $3,4 * 10^{38}$ steigt. Darüber hinaus führt IPv6 die zustandslose Autokonfiguration von

IP-Adressen ein, sodass auf eine zentrale Stelle zur Adressvergabe in vielen Anwendungsfällen verzichtet werden kann.

Transmission Control Protocol (TCP)

Bei TCP handelt es sich um ein verbindungsorientiertes Protokoll, das der Transportschicht des ISO/OSI-Referenzmodells zuzuordnen ist. Es erlaubt eine zuverlässige Ende-zu-Ende-Kommunikation zwischen zwei Anwendungen beziehungsweise Prozessen auf zwei über ein Netzwerk verbundenen Hosts [97]. Hierfür muss zunächst eine Verbindung aufgebaut werden. Dies geschieht bei TCP mit einem 3-Wege-Handshake wie er in Abbildung 8 dargestellt ist.

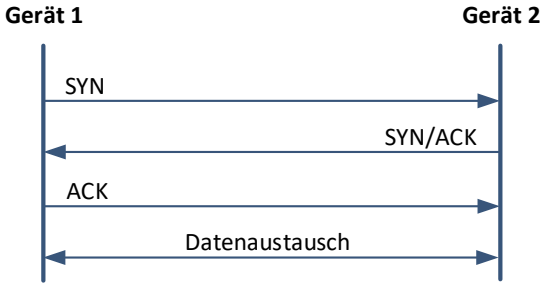

Abbildung 8: Ablauf des 3-Wege-Handshakes von TCP [106].

Der Initiator der Verbindung sendet eine SYN-Nachricht (vgl. engl. synchronize) mit einer Sequenz-Nummer an den Ziel-Host. Dieser bestätigt den Empfang der SYN-Nachricht und seinen Willen, die Verbindung zuzulassen, mit einem SYN/ACK (synchronize acknowledgement). Der Empfang der SYN/ACK-Nachricht wird wiederum vom Initiator der Verbindung mit einem Acknowledgement quittiert. Anschließend können Daten über die aufgebaute Verbindung gesendet werden, wobei der Empfang der Nachrichten jeweils mit einem ACK bestätigt werden muss, um die Zuverlässigkeit der Verbindung sicherzustellen. Wird für ein gesendetes Paket keine ACK-Nachricht erhalten, wird die Übertragung wiederholt. TCP erlaubt auch das Streamen von Daten. Hierbei werden die Daten automatisch segmentiert, in mehreren Nachrichten versendet und auf der Empfängerseite wieder re-assembliert. Weiterhin stellt TCP auch Mechanismen zur Flusssteuerung bereit [7]. So enthalten Acknowledgements für die einzelnen Datensegmente eine Angabe, wie viele Segmente der Empfänger noch aufnehmen kann, ohne dass es zu einem Puffer-Überlauf kommt. Dieser Wert wird als *Receiver's Advertised Window (RWND)* bezeichnet. Zusätzlich erlaubt TCP es auch, mehrere Datensegmente mit nur einem einzelnen ACK zu bestätigen. Wie viele Segmente gesendet werden, bevor ein ACK des Empfängers erwartet wird, wird durch das Congestion Window (CWND) bestimmt. Dies wird überlicherweise mit 1 initialisiert und anschließend in der sogenannten *Slow-Start*-Phase für jedes erhaltene ACK verdoppelt. Hat das CWND einen gewissen Wert erreicht, wird in die *Congestion-Avoidance*-Phase gewechselt. Hier wird das CWND nur jeweils um eins erhöht. Kommt es zu Segment-Verlusten, wird dies auf eine Überlastung der Verbindung zurückgeführt, sodass das

CWND wieder verkleinert wird. Dies kann nach verschiedenen Mustern erfolgen, welche in [7] näher erläutert werden. Abbildung 9 zeigt den Aufbau eines TCP-Headers.

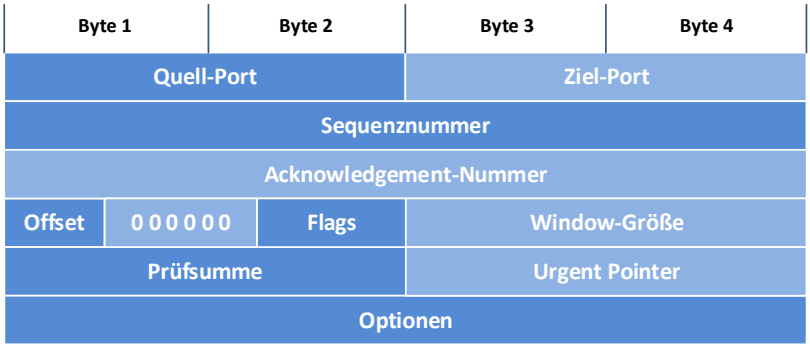

Abbildung 9: Aufbau des TCP-Headers [3].

Der TCP-Header enthält zunächst die Portnummer des Quell- und des Zielprozesses. Darauf folgen eine jeweils 4 Byte breite Sequenz- und Acknowledgement-Nummer, welche der Zuordnung von Acknowledgements zu den jeweiligen Nachrichten dienen. Anschließend folgt ein Offset, der angibt, ab welchem Byte in dem Paket die transportierten Nutzdaten beginnen. Darüber hinaus enthält der Header einige Kontrollfelder (Flags) sowie die Fenstergröße (CWND/RWND), eine Prüfsumme und diverse Optionen. Die Fenstergröße gibt dabei das Minimum aus der empfängerseitig signalisierten Empfangspuffergröße (RWND) und der senderseitig abgeschätzten Überlastfenstergröße (CWND) an. Die Länge der Optionen kann je nach Art der verwendeten Optionen unterschiedlich sein [106].

User Datagram Protocol (UDP)

Das User Datagram Protocol (UDP) stellt ebenfalls einen Vertreter der Transportschicht des ISO/OSI-Referenzmodells dar [84]. Anders als bei TCP handelt es sich bei UDP aber um ein verbindungsloses Protokoll. Jede Nachricht wird also nur für sich betrachtet [67]. Weiterhin stellt UDP keine Mechanismen zur Flusskontrolle oder zur zuverlässigen Übertragung (vgl. ACK-Nachrichten in TCP) bereit. Die maximale Größe eines UDP Datagrams beträgt 65.535 Byte. Für größere Datenmengen muss eine Segmentierung auf den darüber liegenden Schichten erfolgen, da UDP selbst keine Segmentierung und Re-Assemblierung der Daten vornimmt. Dabei müssen die Prozesse auf den höheren Schichten selbst auf die richtige Reihenfolge der empfangenen Datensegmente achten [106]. Abbildung 10 zeigt den Aufbau des UDP-Headers.

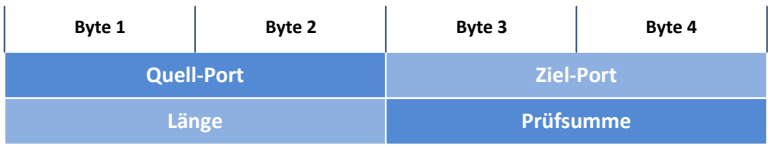

Abbildung 10: Aufbau des UDP-Headers [84].

Der UDP-Header ist mit einer Länge von 8 Byte deutlich kleiner als der TCP-Header. Er enthält die Ports des Quell- und des Ziel-Prozesses sowie ein Feld, das die Länge der Nachricht in Byte angibt. Weiterhin enthält der Header eine Prüfsumme, mit deren Hilfe die Integrität der empfangenen Daten überprüft werden kann. Trotz seiner Unzuverlässigkeit und den fehlenden Möglichkeiten zur Flusskontrolle gibt es einige Einsatzfelder, in denen UDP zu bevorzugen ist. Hierzu zählen insbesondere Anwendungen, bei denen eine erneute Übertragung verloren gegangener Daten keinen Nutzen hat und somit unnötig Kommunikationsressourcen in Anspruch nehmen würde. Dies ist beispielsweise bei der Live-Übertragung von Audio- oder Video-Streams der Fall. Hier läuft die Übertragung beziehungsweise das übertragene Ereignis weiter, sodass der Inhalt der verlorenen Datensegmente bereits in der Vergangenheit liegt. Somit sind die Daten für den Nutzer nicht mehr von Bedeutung. Darüber hinaus eignet sich UDP für das Versenden gleicher Daten an mehrere Empfänger (Broadcast und Multicast), was mit TCP nicht möglich ist, und für Anwendungsbereiche, in denen eine Fluss- oder Überlastkontrolle explizit unerwünscht ist.

2.2 Service-Orientierte Architekturen

Die Service-Oriented Architecture (SOA) bezeichnet ein Architektur-Konzept für verteilte Systeme, bei dem die von Netzwerkteilnehmern bereitgestellten Funktionalitäten in Form von Diensten (engl. Services) gekapselt werden [74, 81]. Der Grundgedanke hinter einer SOA ist, Interoperabilität zwischen verschiedenartigen Diensten und Softwarekomponenten zu ermöglichen, ohne dass schon zur Entwicklungszeit apriorisches Wissen über diese Dienste vorhanden sein muss. Hierzu stützt sich das SOA-Konzept maßgeblich auf vier Säulen: Verteiltheit, lose Kopplung, Wiederverwendbarkeit und Interoperabilität.

Die *Verteiltheit* bezeichnet die Verteilung einzelner Softwarekomponenten auf verschiedene physische Systeme, welche über ein Netzwerk miteinander verbunden sind. Die einzelnen Komponenten können über dieses Netzwerk miteinander interagieren. Hierbei liegt der Fokus auf dem Anbieten, Suchen und Nutzen von Diensten. Hierzu kann beispielsweise ein zentrales Diensteregister zum Einsatz kommen. Als *lose Kopplung* bezeichnet man die Trennung zwischen den Schnittstellen zur Nutzung einer Softwarekomponente bzw. eines Dienstes und der eigentlichen Implementierung. Diese Trennung wird in einer SOA durch die Kapselung der Funktionalitäten in Services erreicht. Die Dienste können dabei über eine separat abrufbare Beschreibung der Service-Schnittstelle verfügen [81]. So ist es zum einen möglich, Dienste zu nutzen, deren Schnittstelle bei der Entwicklung einer Softwarekomponente noch nicht bekannt war. Zum anderen lässt sich die Implementierung hinter der Schnittstelle beliebig austauschen, ohne die Nutzbarkeit des Dienstes zu beeinflussen. Die lose Kopplung begünstigt zudem die *Wiederverwendbarkeit* von Softwarekomponenten sowie von Schnittstellen und deren Beschreibung. Die *Interoperabilität* ergibt sich bei einer SOA zu großen Teilen aus der losen Kopplung und der Schnittstellenbeschreibung. Allerdings ist auch die Verwendung von eta-

blierten Standards bei der Kommunikation und der Beschreibung und Nutzung der Dienste für die Interoperabilität von besonderer Bedeutung [74].

SOA eignen sich auf Grund ihrer Flexibilität und des Fokus auf Interoperabilität hervorragend für den Einsatz in IoT-Szenarien. Eine SOA lässt sich dabei auf verschiedene Arten umsetzen. Im Internet haben sich zwei Varianten besonders durchgesetzt: Web Services auf Basis der WS*-Protokolle und sogenannte REST-basierte Systeme. Als leichtgewichtiges Web-Service-Profil für den Einsatz in ressourcenbeschränkten Umgebungen hat sich das Devices Profile for Web Services (DPWS) etabliert. Nachfolgend werden beide Varianten tiefergehend erläutert und gegeneinander abgewogen.

2.3 Representational State Transfer

Das Paradigma des Representational State Transfer (REST) wurde erstmals im Jahr 2000 von *Roy Fielding* in seiner Dissertation beschrieben [39] und bezeichnet einen Architekturansatz, der auch zur Umsetzung einer SOA genutzt werden kann. Die Kernkonzepte hinter REST leitete *Fielding* aus den zur damaligen Zeit im Internet vorherrschenden Software-Architekturen ab. Dabei konnte *Fielding* sechs wesentliche Eigenschaften herauskristallisieren, die von einem Großteil der im Internet vorhandenen Softwarelösungen erfüllt werden. Diese sechs Eigenschaften bilden die Hauptbestandteile des REST-Paradigmas und werden nachfolgend tiefergehend erläutert. Das wohl bekannteste Protokoll, das dem REST-Prinzip folgt, ist das Hypertext Transfer Protocol (HTTP).

Client-Server-Prinzip

Das Client-Server-Prinzip beschreibt die Aufteilung der Funktionalitäten einer Netzwerk-Applikation in zwei unterschiedliche Teile. Auf der einen Seite steht der Server, der bestimmte Dienste und Daten im Netzwerk anbietet. Auf der anderen Seite steht der Client, der die Dienste nutzen oder die Daten abrufen möchte. Hierzu sendet er einen Request (engl. für Anfrage) über das Netzwerk an den Server. Dieser kann den Request entweder verwerfen oder die angefragte Aktion ausführen. Anschließend sendet der Server eine entsprechende Antwort an den Client. *Fielding* spricht in diesem Zusammenhang von einer *„separation of concerns“*, einer Trennung der Aufgabenbereiche. Diese Trennung ermöglicht es, die Software-Komponenten des Servers erheblich zu vereinfachen, da ein großer Teil der Aufgaben, wie beispielsweise die Darstellung der Antworten für den Nutzer, auf den Client ausgelagert wird. Auf diese Weise wird die Systemlast auf dem Server verringert und somit die Skalierbarkeit in Form von zeitgleich verwaltbaren Client-Verbindungen erhöht. Darüber hinaus lassen sich beide Seiten, Client und Server, unabhängig voneinander weiterentwickeln, solange die Schnittstellen zur Kommunikation gleich bleiben [39].

Zustandslosigkeit

Die Zustandslosigkeit in einem REST-basierten System ist dadurch gekennzeichnet, dass jeder Request eines Clients alle Informationen enthalten muss, die der Server benötigt, um die Anfrage zu

verstehen und zu verarbeiten. Der Client kann sich also nicht auf Kontextinformationen verlassen, die auf dem Server gespeichert sind. *Fielding* bezeichnet diese Requests auch als *„self-contained“*, also „in sich geschlossen“. Die Verwaltung des Anwendungszustandes wird damit auf den Client ausgelagert. Die Zustandslosigkeit bringt zwei wesentliche Vorteile mit sich. Zum einen wird die Skalierbarkeit erhöht. Da der Server keine Kontextinformationen speichern muss, können Systemressourcen schneller wieder freigegeben werden. Zum anderen wird die Zuverlässigkeit gesteigert. Durch die in sich geschlossenen Anfragen wirkt sich eine fehlerhafte Anfrage nicht auf die gesamte Kommunikation aus. Jedoch bleibt zu beachten, dass sich daraus auch eine höhere Netzwerklast ergeben kann, da für eine Interaktion potentiell mehrere Nachrichten gesendet werden müssen, um zu dem vom Client gewünschten Ergebnis zu kommen. Des Weiteren verliert der Server jegliche Möglichkeit, den korrekten Anwendungsverlauf zu überprüfen. Er muss vielmehr darauf vertrauen, dass die Interpretation seiner Antworten auf der Client-Seite auch über verschiedene Implementierungen hinweg gleich bleibt [39].

Caching

Um den erhöhten Kommunikationsaufwand, der sich aus der Zustandslosigkeit ergibt, abzufedern, führt *Fielding* Caching als weiteres Merkmal REST-basierter Systeme ein. Hierbei speichert der Client die Antworten des Servers auf seine Anfragen in einem Cache zwischen. Möchte der Client später eine äquivalente Anfrage stellen, kann er die frühere Antwort des Servers wiederverwenden, ohne die Anfrage tatsächlich zu versenden. Die hierbei eingesparten Interaktionen mit dem Server entlasten nicht nur das Netzwerk und den Server, sie führen auch zu einer aus Nutzersicht höheren Performance. Allerdings muss sichergestellt werden, dass die im Cache gespeicherten Daten nicht zu stark von den vom Server bereitgestellten Daten abweichen. Dies kann der Fall sein, wenn die Daten auf dem Server nach der ersten Abfrage von einem anderen Client geändert wurden oder sich stetig selbst verändern, wie beispielsweise die von einem Temperatursensor gemessenen Werte [39].

Einheitliche Schnittstellen

Einheitliche Schnittstellen zwischen den einzelnen Komponenten verringern die Komplexität des Gesamtsystems. Weiterhin ermöglichen sie die Entkoppelung der Schnittstellen von den eigentlich bereitgestellten Diensten. Auf diese Weise können die Implementierungen der Dienste auf dem Server weiterentwickelt werden, ohne dass sich die Schnittstellen für deren Nutzung ändern. Typischerweise werden Dienste und Daten in REST-basierten Systemen als Ressourcen dargestellt, welche über einen sogenannten Uniform Resource Identifier (URI) eindeutig adressiert werden können. Auf jede Ressource lässt sich die gleiche Menge von Operationen anwenden, unabhängig von der dahinterstehenden Dienstimplementierung. Bei der Kommunikation eines Clients mit einem Server werden Repräsentationen dieser Ressourcen, nicht die Ressourcen selbst, ausgetauscht. Solche Repräsentationen halten dabei den aktuellen oder den gewünschten Zustand einer Ressource fest und bestehen meist aus einer Byte-Sequenz und Metadaten, die diese Bytes beschreiben. Es ist aber zu beachten, dass ei-

ne Verallgemeinerung der Schnittstellen zu einer Reduktion der Effizienz bei der Interaktion führt. So können in einigen Anwendungsfällen mehrere Requests nötig sein, um eine bestimmte Aktion auszuführen. Dies ließe sich mit speziell auf den jeweiligen Anwendungsfall angepassten Schnittstellen umgehen [39].

Hierarchische Ebenen

REST-Systeme können sich aus mehreren hierarchischen Ebenen zusammensetzen. Dabei kann jede Komponente nur die Ebenen einsehen, mit denen sie direkt verbunden ist. So wird die Komplexität des Gesamtsystems aus Sicht der einzelnen Komponenten weiter verringert. Darüber hinaus können so ältere, sogenannte „Legacy"-Komponenten gekapselt werden. Weiterhin erlaubt die Einteilung in mehrere Ebenen das Anwenden von Sicherheitsregeln auf Nachrichten, die Ebenengrenzen überschreiten [39].

Code-on-Demand

Als Code-on-Demand bezeichnet *Fielding* die Möglichkeit, ausführbaren Programm-Code z.B. in Form von Scripten an den Client zu übertragen. Auf diese Weise können komplexe Operationen auf den Client ausgelagert werden. Dies erhöht einerseits die Skalierbarkeit des Servers. Andererseits wird der Client stärker belastet. Hinzu kommen mögliche Sicherheitsrisiken, da so auch schädlicher Code auf den Client übertragen werden kann [39].

2.4 DPWS - Das Devices Profile for Web Services

Bei dem Devices Profile for Web Services (DPWS) handelt es sich um eine Teilmenge der WS*-Protokolle, ein sogenanntes Profil. Es wurde im Jahr 2009 als offizieller OASIS-Standard veröffentlicht und soll die Verwendung von klassischen, aus dem Internet bekannten Web Services zum Aufbau einer SOA in ressourcenbeschränkten Umgebungen ermöglichen. Um den Ressourcenbedarf dabei möglichst weit zu reduzieren, umfasst DPWS nur die wesentlichen Bestandteile der WS*-Protokolle. Abbildung 11 zeigt den vollständigen DPWS-Protokollstapel.

Bei der Kommunikation wird in DPWS zwischen einem Device und einem Client unterschieden. Ein Device kann mehrere Dienste anbieten, welche als *Hosted Services* bezeichnet werden. Diese werden von einem übergeordneten Dienst, dem *Hosting Service* nach außen hin verfügbar gemacht. Hosted Services stellen dabei die Funktionalität eines Devices bereit, während der Hosting Service eine wichtige Rolle bei der Suche nach Diensten spielt. Der Nachrichtenaustauch in DPWS erfolgt über das Simple Object Access Protocol (SOAP). Dabei handelt es sich um ein menschenlesbares und frei erweiterbares XML-Format (Extensible Markup Language). Eine SOAP-Nachricht wird auch als Envelope (engl. für Umschlag) bezeichnet. Die SOAP-Nachrichten können entweder per HTTP POST Request und TCP oder direkt per UDP an das Device gesendet werden. Der Versand von SOAP über

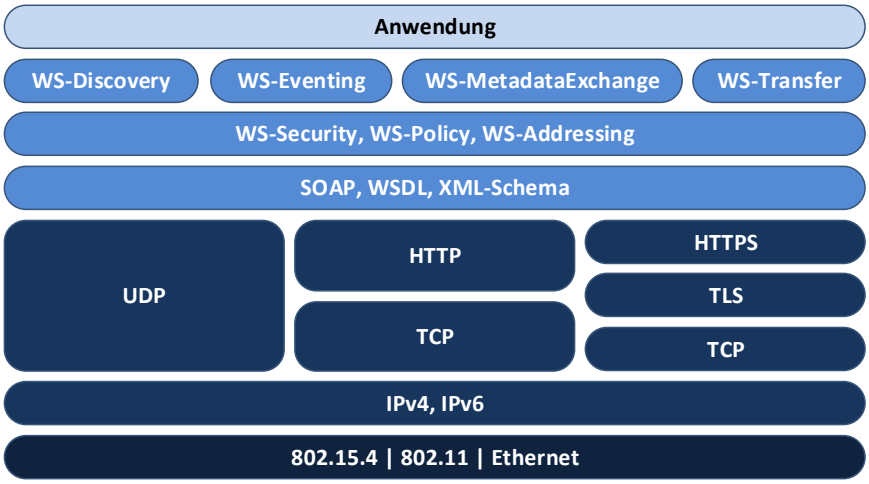

Abbildung 11: Darstellung des DPWS-Protokollstapels [16].

UDP wird dabei allerdings vorrangig bei der Multicast- und weniger bei der direkten Kommunikation mit einem Device verwendet. Dies liegt darin begründet, dass UDP keine zuverlässige Übertragung erlaubt und DPWS keine eigenen Mechanismen bereitstellt, um dies zu kompensieren. Da TCP als verbindungsorientiertes Protokoll jedoch keine Multicast-Kommunikation unterstützt, kommt in diesen Fällen trotzdem UDP zum Einsatz. Der Ablauf von Interaktionen in DPWS lässt sich in mehrere Kommunikationsphasen gliedern, die nachfolgend erläutert werden [16].

2.4.1 Service Discovery

Als Service Discovery bezeichnet man das Suchen und Finden von bestimmten Diensten. DPWS nutzt hierfür das WS-Discovery-Protokoll [75]. Dabei wird zwischen der impliziten und expliziten Discovery unterschieden. Als implizite Discovery wird dabei das Versenden von Hello-Nachrichten als Broadcast-Nachricht bezeichnet. Diese Hello-Nachrichten werden von jedem Device versendet, wenn es einem Netzwerk beitritt. Unter einer expliziten Discovery versteht man das aktive Suchen nach Diensten durch einen Client. Dies kann entweder im *Managed Mode* oder im *Ad Hoc Mode* erfolgen. Der Managed Mode setzt das Vorhandensein eines Service Brokers im Netzwerk voraus (vgl. Abbildung 12) [75].

Devices registrieren ihre Hosted Services bei dem Service Broker. Sucht ein Client nach einem bestimmten Dienst, sendet er eine Suchanfrage an den Broker. Dieser antwortet dann mit einer Liste von Diensten, welche die gesuchten Eigenschaften aufweisen, und den Kontaktinformationen des jeweiligen Devices. Bei dem Ad Hoc Mode kann auf einen Service Broker verzichtet werden. Hier durchsucht der Client selbstständig das Netzwerk. Dies geschieht mit Hilfe von Probe-Nachrichten, welche per Multicast an alle Devices gesendet werden. Die Probe-Nachrichten werden vom Hosting

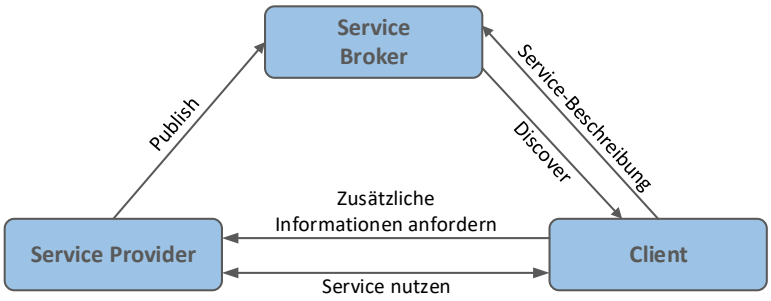

Abbildung 12: Discovery mit Hilfe eines Service Brokers [75].

Service empfangen und verarbeitet. Dabei wird überprüft, ob ein Device einen Hosted Service bereitstellt, der dem gesuchten Dienst entspricht. Ist dies der Fall, generiert der Hosting Service eine ProbeMatch-Nachricht, die eine Liste der entsprechenden Hosted Services enthält. Das ProbeMatch kann zusätzlich die *Endpoint Reference* enthalten. Der Aufbau einer Endpoint Reference ist in WS-Addressing beschrieben. Sie wird vom Client benötigt, um einen Dienst direkt ansprechen zu können. Ist diese Information nicht bereits im ProbeMatch enthalten, kann der Client eine Resolve-Nachricht versenden. Ebenso wie Probe wird ein Resolve immer als Multicast versendet und dient dazu, die Endpoint Reference eines Dienstes anhand seines Namens zu ermitteln. Abbildung 13 stellt den Ablauf einer expliziten ad hoc Discovery dar [75].

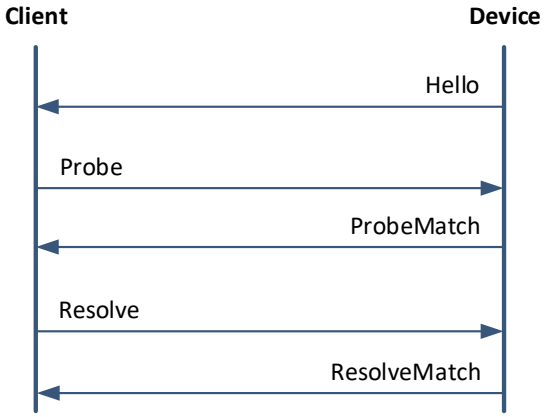

Abbildung 13: Ablauf eines ad hoc Discovery-Prozesses in DPWS [75].

Nachdem ein Dienst gefunden und die nötigen Kontaktinformationen ermittelt wurden, können die Metadaten des Dienstes abgerufen werden [75].

2.4.2 Metadaten-Austausch in DPWS

Der Abruf der Metadaten erfolgt über das Protokoll WS-MetadataExchange [25]. Hierzu sendet der Client zunächst einen GetMetadata-Request an den Hosted Service. Die GetMetadataResponse des

Dienstes enthält eine URI, unter der die Dienstebeschreibung per HTTP GET-Request abgerufen werden kann. Die Beschreibung eines Dienstes erfolgt mit Hilfe der Web Service Description Language (WSDL). Dabei handelt es sich um ein standardisiertes, XML-basiertes Format, mit dessen Hilfe die Funktion eines Dienstes und seiner Operationen sowie Anforderungen an die Eingabeparameter charakterisiert werden können. Des Weiteren enthält die WSDL-Beschreibung auch Informationen über die Ausgaben einer Operation. Sie gibt aber keinen Aufschluss darüber, wie die Funktionen des Dienstes intern umgesetzt sind. Nach dem Austausch der Metadaten und dem Auslesen der WSDL-Beschreibung des Dienstes ist der Client in der Lage, den Dienst zu nutzen [25].

2.4.3 Die Nutzung von DPWS-Services

Die Nutzung eines Dienstes kann auf zweierlei Arten geschehen. Zum einen kann der Client einzelne Operationen des Dienstes mit Invoke-Nachrichten ausführen. Hierbei muss er die nötigen Parameter der Operation wie in der WSDL-Beschreibung des Dienstes dargestellt an den Dienst übergeben. Zum anderen kann der Client WS-Eventing, den Publish/Subscribe-Mechanismus von DPWS, verwenden. Dies ist allerdings nur bei Diensten möglich, welche lediglich Daten zurückliefern und keine Eingabeparameter erfordern. Ein beispielhafter Anwendungsfall ist das Abrufen der aktuellen Temperatur von einem Temperatursensor. Wird kein WS-Eventing genutzt, muss der Client den Temperaturwert in regelmäßigen Abständen abrufen. Dieses Vorgehen wird auch als *Polling* bezeichnet. Polling führt zu einem erhöhten Kommunikationsaufwand, da der aktuelle Wert auch abgefragt wird, wenn er sich seit der letzten Abfrage nicht geändert hat. WS-Eventing erlaubt das Abonnieren von Änderungen eines Dienstes und umgeht damit die beim Polling entstehenden Nachteile. Hierzu versendet der Client eine Subscribe-Nachricht an den Dienst. Dieser antwortet daraufhin mit einer SubscribeResponse, um dem Client den Erhalt des Abonnements zu bestätigen. Erfolgen nun Änderungen an dem Dienst, wie zum Beispiel ein Anstieg der aktuellen Temperatur, versendet der Dienst automatisch eine Nachricht an den Client. Diese automatische Benachrichtigung wird auch als *Event* oder *Notification* bezeichnet. Hierbei entfällt nicht nur die unnötige Übertragung des Temperaturwertes bei eigentlich konstanter Temperatur, sondern auch die Wiederholung der Anfrage durch den Client. Zudem kann der Client schneller auf Änderungen reagieren, da er sofort benachrichtigt wird. Bei Polling kann die Reaktion im schlechtesten Fall um das Polling-Intervall (zeitlicher Abstand zwischen zwei Anfragen) verzögert sein. Um das Abonnement zu löschen, sendet der Client eine Unsubscribe-Nachricht an den Dienst, welcher den Client daraufhin von der Liste der Abonnenten entfernt. Abbildung 14 stellt die Kommunikationsabläufe für beide Interaktionsformen graphisch dar [94].

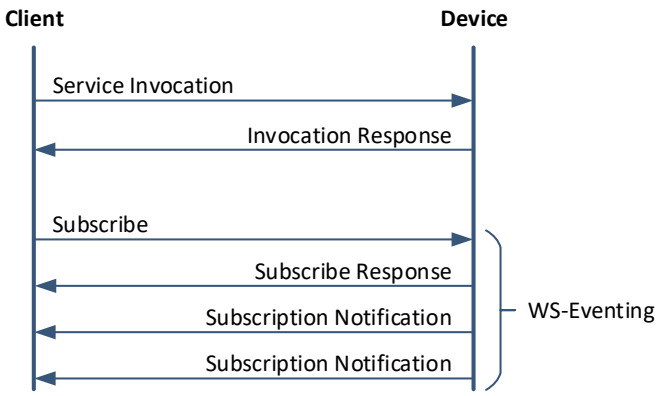

Abbildung 14: Nutzung von Diensten in DPWS mittels Invoke-Nachrichten und WS-Eventing [94].

2.5 CoAP - Das Constrained Application Protocol

Bei dem Constrained Application Procotol (CoAP) handelt es sich um ein Datentransfer-Protokoll, welches speziell für den Einsatz in ressourcenbeschränkten Umgebungen entworfen wurde. Begründet durch die Erkenntnis, dass DPWS-basierte Web Services nur sehr begrenzt auf stark limitierten Endgeräten genutzt werden können, wurde es 2010 erstmals von der CoRE Working Group (Constrained RESTful Environments) der Internet Engineering Taskforce (IETF) spezifiziert. Wie Abbildung 15 zeigt, setzt sich CoAP aus einer Basis- und mehreren Subspezifikationen zusammen. In der Basisspezifikation werden alle Grundregeln für den Nachrichtenaustausch, wie beispielsweise der Nachrichtenaufbau und das Kommunikationsmodell, beschrieben. Die Subspezifikationen erweitern die Basisfunktionalität um spezielles Verhalten, welches nicht zwangsweise von allen Geräten unterstützt werden muss.

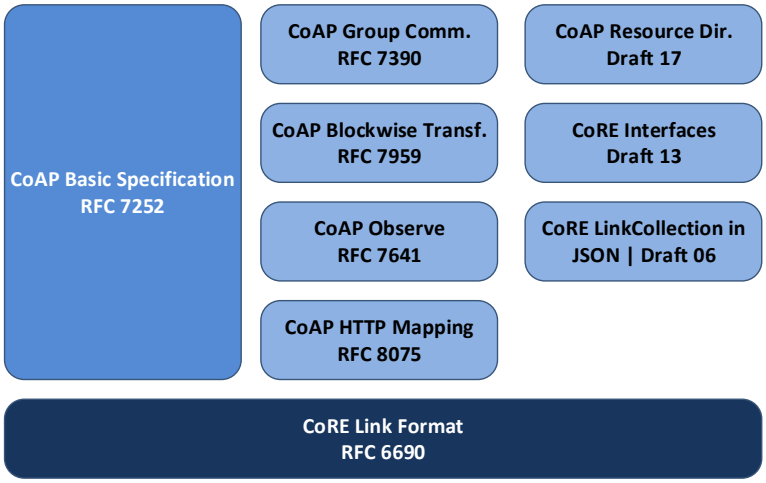

Abbildung 15: Status der Basis- und wichtigsten Subspezifikationen von CoAP.

Mittlerweile sind die Basisspezifikation, das CoRE Link Format sowie andere für das IoT wichtige Funktionen, wie die Gruppenkommunikation und Eventing (CoAP Observe), anerkannte Standards, während sich einige Subspezifikationen noch in der Entwicklung befinden und nur als Internet Draft vorliegen. Das CoRE Link Format legt dabei fest, wie CoAP-Geräte ihre angebotenen Dienste und deren Attribute beschreiben können. CoAP basiert, wie auch HTTP, auf den REST-Prinzipien. Auf Grund der vielen Gemeinsamkeiten wird CoAP auch häufig vereinfacht als binäres HTTP bezeichnet. Allerdings wird diese Beschreibung den Unterschieden zwischen den beiden Protokollen nur bedingt gerecht. Abbildung 16 zeigt den Protokollstapel von CoAP [99].

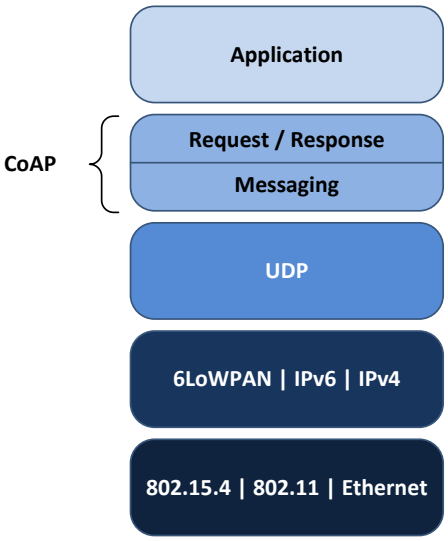

Abbildung 16: Der CoAP-Protokollstapel [99].

Im Gegensatz zu HTTP verzichtet CoAP auf die Nutzung von TCP, sondern basiert auf UDP. Hierdurch entfällt der aufwändige TCP Handshake vor jedem Kommunikationsvorgang. Auf diese Weise wird der Kommunikationsaufwand für das einzelne Gerät erheblich reduziert.
Bei der Kommunikation folgt CoAP, wie auch HTTP, dem Client-Server-Prinzip [99]. Ein CoAP-Server stellt Daten oder Dienste in Form von Ressourcen zur Verfügung, welche durch einen Unified Resource Identifier (URI) eindeutig adressierbar sind. Bei der Kommunikation werden Repräsentationen dieser Ressourcen ausgetauscht. Dabei setzt sich CoAP selbst aus zwei Subschichten zusammen, dem Message-Layer und dem Request/Response-Layer. Der Message-Layer definiert vier verschiedene Nachrichtentypen: confirmable (CON), non-confirmable (NON), acknowledgement (ACK) und reset (RST). Aufgrund der Tatsache, dass CoAP auf UDP basiert, wird per se keine zuverlässige Datenübertragung bereitgestellt. Um diesen Nachteil auszugleichen, ermöglicht CoAP mit CON-Nachrichten einen optionalen, zuverlässigen Nachrichtenaustausch. Wenn ein Server eine CON-Nachricht erhält, muss er diese mit einer ACK-Nachricht beantworten, um die erfolgreiche Übertragung zu bestätigen. Erhält ein Client bis zu einem bestimmten Timeout kein ACK von dem

Server, sendet er seinen Request erneut. Das Timeout ergibt sich aus einer festgelegten Zeit (Standard sind zwei Sekunden), die über eine Zufallszahl gewichtet und bei jeder Sendewiederholung verdoppelt wird. Die Anzahl der Sendeversuche ist standardmäßig auf vier begrenzt. Die Zuordnung der ACKs zu den jeweiligen CON-Nachrichten geschieht auf Grundlage einer Nachrichten-ID. Um den Kommunikationsaufwand zu verringern, kann der Server die Antwort auf die Anfrage des Clients direkt an das Acknowledgement anhängen (Piggybacked Response). Für einen unzuverlässigen Nachrichtenaustausch können NON-Nachrichten verwendet werden, da diese keine Empfangsbestätigung erfordern. Mit RST-Nachrichten können CoAP-Geräte auf Nachrichten reagieren, die fehlerhaft sind oder nicht interpretiert werden können.

Funktion	**Beschreibung**
GET	Die GET-Methode dient dazu, eine Repräsentation der Ressource mit der entsprechenden URI von einem Server abzurufen.
PUT	Mit einem PUT-Request wird eine Repräsentation einer Ressource an den Server übertragen. Ein Server, der einen PUT-Request erhält, muss die in der Request-URI spezifizierte Ressource mit der angehängten Repräsentation aktualisieren oder sie, falls sie noch nicht existiert, neu anlegen.
POST	Ein POST-Request wird verwendet, um den Server zur Verarbeitung einer angehängten Ressourcenrepräsentation aufzufordern. Was im Zuge der Verarbeitung geschieht, hängt von dem jeweiligen Server und der angefragten Ressource ab. Üblicherweise hat ein POST-Request die Aktualisierung oder das Anlegen einer Ressource zur Folge.
DELETE	Die DELETE-Methode wird verwendet, um die Ressource mit der entsprechenden URI auf dem Server zu löschen.

Tabelle 1: CoAP-Basismethoden und ihre Auswirkungen [99].

Der Request/Response-Layer ermöglicht das Request-Response-Schema zwischen den Netzwerkteilnehmern. Auf dieser Ebene werden die vier Grundoperationen GET, POST, PUT und DELETE ausgeführt, welche auf alle Ressourcen angewendet werden können. Tabelle 1 beschreibt die Wirkung der einzelnen Operationen im Detail. Die Zuordnung von Requests zu Responses kann anhand eines optionalen Tokens erfolgen. Der Aufbau einer CoAP-Nachricht ist in Abbildung 17 dargestellt.

Das erste Byte einer CoAP-Nachricht enthält die genutzte Protokollversion (V), den Nachrichtentyp (T) sowie die Länge des Tokens (TKL) in Bytes. Die Tokenlänge ist 0, wenn kein Token vorhanden ist. Das zweite Byte im CoAP-Header enthält den Message Code. Dieser gibt an, ob es sich bei

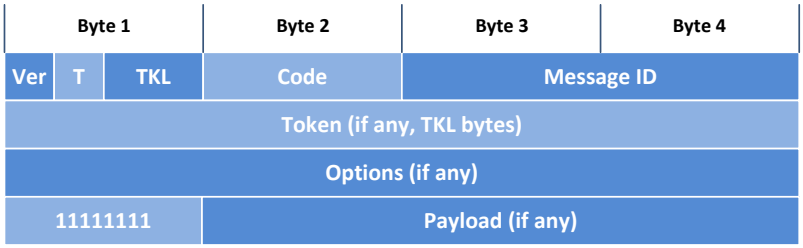

Abbildung 17: Struktur einer CoAP-Nachricht [99].

der Nachricht um einen Request (GET, PUT, POST oder DELETE) oder eine Response handelt. Bei einer Response enthält das Message-Code-Feld einen dem Request entsprechenden Antwort-Code. Dieser gibt eine Aussage darüber, ob die Anfrage des Clients erfolgreich verarbeitet werden konnte. Die Bytes drei und vier enthalten die Nachrichten-ID zur Zuordnung der Acknowledgemets zu den jeweiligen CON-Nachrichten. Anschließend folgt ein optionaler Token mit der im TKL-Feld definierten Länge sowie eine beliebige Anzahl von Optionen. Eine CoAP-Header-Option besteht aus einer Optionsnummer, der Länge der Option in Byte und dem Optionswert. Jede Option kann beliebig oft in einem CoAP-Header vorkommen. Die Optionen werden im Header aufsteigend nach den Optionsnummern angeordnet. Hierbei sind die Nummern für jeden Optionstyp eindeutig festgelegt und in [99] oder der jeweiligen Subspezifikation, die sie einführt, beschrieben. Enthält ein CoAP-Paket Nutzdaten, so folgt auf die Optionen ein 1 Byte langer Payload-Marker, bei dem alle Bits gesetzt sind. Der CoAP-Header wird binär kodiert, um zum einen die Header-Größe zu reduzieren. Zum anderen wird auf diese Weise der Aufwand zum Parsen der Header erheblich verringert, da hierbei nur auf Byte-Operationen zurückgegriffen werden muss. Ein typischer CoAP-Header hat dabei eine Länge zwischen 4 und 20 Bytes, je nachdem wie viele Optionen er enthält. So wird beispielsweise auch die URI der von einem Client angefragten Ressource in Form von Header-Optionen gespeichert. Die durchschnittliche Größe eines HTTP-Headers liegt hingegen zwischen 700 und 800 Bytes [47].
Die Unterschiede zwischen HTTP und CoAP beschränken sich jedoch nicht nur auf die Nachrichtengröße. CoAP wurde speziell auf den Bereich der Maschine-zu-Maschine-(M2M)-Kommunikation zugeschnitten, während der Einsatzbereich von HTTP eher in der Mensch-zu-Maschine-(H2M)-Kommunikation liegt. Im H2M-Bereich sucht ein Nutzer gezielt im Netzwerk nach Daten und Diensten. Diese werden dann mittels Protokollen wie HTTP von den jeweiligen Servern abgerufen und für den Nutzer lesbar dargestellt. Die Suche von Diensten und Interpretation der gewonnenen Informationen liegen also im Aufgabenbereich des Nutzers. In M2M-Szenarien entfällt die menschliche Komponente. Die Geräte müssen also eigenständig die richtigen Dienst- und Informationsquellen im Netzwerk finden und die Ergebnisse selbst interpretieren. Hierzu stellt CoAP einige Funktionalitäten bereit, welche im Folgenden näher erläutert werden. Tabelle 2 gibt abschließend einen zusammenfassenden Überblick über die Gemeinsamkeiten und Unterschiede zwischen HTTP und CoAP.

	HTTP	**CoAP**
Rollenmodell	Client/Server	Client/Server
Kommunikationsmuster	Request/Response	Request/Response
Transport	TCP	UDP
Service-/Datenmodell	Ressourcen	Ressourcen
Service-/Datenadressierung	URI	URI
Methoden	GET, PUT, POST, DELETE, TRACE, HEAD, OPTIONS, CONNECT	GET, PUT, POST, DELETE
Response-Codes	Informational 1xx Success 2xx Redirection 3xx Client Error 4xx Server Error 5xx	Success 2.xx Client Error 4.xx Server Error 5.xx
Header-Kodierung	ASCII	Binär
Proxy und Caching	HTTP zu HTTP	CoAP zu CoAP HTTP zu CoAP CoAP zu HTTP
Ressourcen-Discovery	nicht unterstützt	CoAP
Eventing	Long Polling/Comet, Web Sockets, ...	CoAP Observe
Sicherheit	SSL/TLS	DTLS

Tabelle 2: Gegenüberstellung von CoAP und HTTP [47].

2.5.1 Discovery von Ressourcen mit CoAP

Als Discovery bezeichnet man die Fähigkeit von Geräten, andere Geräte im Netzwerk sowie deren bereitgestellte Funktionalitäten zu entdecken. Dies ist im Bereich der M2M-Kommunikation von besonderer Bedeutung, da die Geräte ohne menschliches Eingreifen miteinander interagieren sollen. In CoAP wird dieses Problem mit Hilfe einer dedizierten Ressource gelöst, welche von jedem CoAP-Server bereitgestellt werden muss. Sendet ein Client einen GET-Request an die URI *„/well-known/core“*, erhält er eine Liste aller Ressourcen, die von dem Server bereitgestellt werden. Die Liste kann neben der URI der jeweiligen Ressource auch den Ressourcen-Typ (rt), die unterstützten Payload-Formate (Content Type, ct) sowie eine Interfacebeschreibung (if) enthalten.

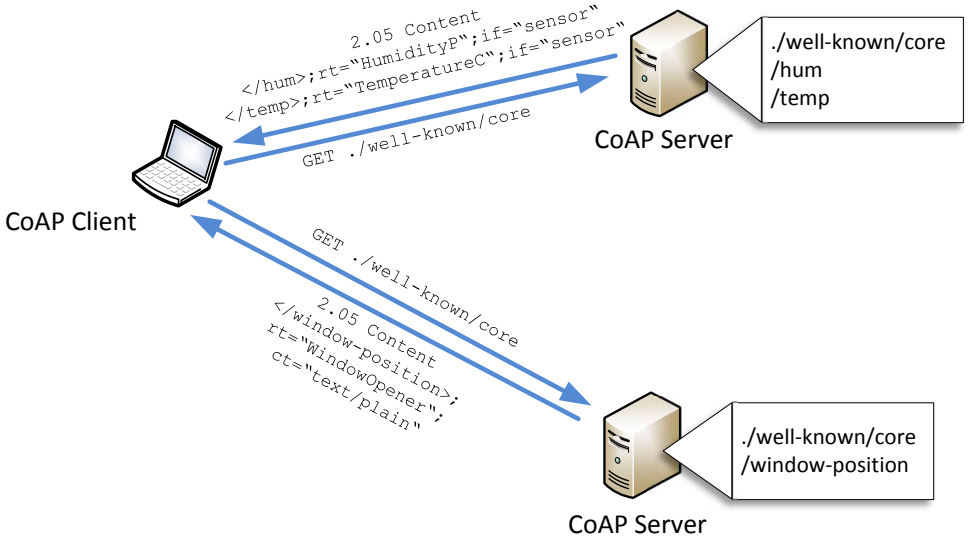

Abbildung 18: Ablauf des Discovery-Vorgangs in CoAP [99].

Um mehr als einen Server abzufragen oder alle Server im Netzwerk zu finden, kann der Client den GET-Request auch an die All-CoAP-Nodes-Adresse oder eine andere Multicast-Adresse senden (vgl. 2.5.4 Gruppenkommunikation in CoAP). Anschließend kann der Client aus den Listen diejenige Ressource auswählen, die am ehesten seinen Anforderungen entspricht, und diese über die entsprechende URI direkt anfragen. Abbildung 18 zeigt den Ablauf der Discovery mittels eines Multicast-Requests.

2.5.2 Publish/Subscribe-Mechanismen in CoAP

Ein typischer Anwendungsfall im Internet of Things (IoT) ist es, die Umgebung mittels Sensoren zu beobachten und anhand dieser Beobachtungen über Aktoren Aktionen auszuführen, welche die Umwelt beeinflussen. Hierzu müssen in regelmäßigen Abständen die Werte der Sensoren abgerufen werden. Dies kann entweder durch die Aktoren direkt geschehen oder durch eine dritte Instanz, die dann

ihrerseits die Aktoren ansteuert. Bei dem sogenannten Polling kommt es allerdings häufig vor, dass die aktuellen Sensorwerte abgerufen werden, obwohl sie keine Änderung erfahren haben. Dies ist zum einen sehr energieaufwändig, zum anderen wird unnötiger Datenverkehr im Netzwerk verursacht. Dieses Problem kann mit Publish/Subscribe-Mechanismen umgangen werden. Dabei registriert sich der Client bei einem Server für das Abonnement beispielsweise eines Sensorwertes. Erfährt dieser Sensorwert nun eine Änderung, sendet der Server eine Nachricht mit dem neuen Sensorwert an alle Abonnenten (Notification). Auch CoAP stellt mit CoAP Observe einen solchen Publish/Subscribe-Mechanismus zur Verfügung [49]. Um die Änderungen einer Ressource zu abonnieren, sendet ein Client einen GET-Request für die jeweilige Ressource an den Server und fügt diesem die Observe-Option hinzu. Der Wert dieser Option ist eine 2 Byte lange Sequenznummer. Ist diese in dem Request '0', bedeutet dies, dass der Client in die Liste der Beobachter für die jeweilige Ressource aufgenommen werden möchte. Ist dieser Wert '1', so will der Client aus dieser entfernt werden und keine weiteren Notifications mehr erhalten. Bei jeder Änderung der Ressource sendet der Server eine Response-Nachricht an alle Beobachter. Diese Response enhält ebenfalls die Observe-Option, um die Nachricht als Notification zu kennzeichnen. Hier stellt die Sequenznummer die Versionsnummer der Ressource dar. Wenn ein Client seinem ursprünglichen Request einen Token hinzufügt, muss dieser auch in allen Notifications vorhanden sein, um dem Client die Zuordnung der Notifications zu dem Request zu vereinfachen.

2.5.3 Übertragung großer Datenmengen mit CoAP

In ressourcenbeschränkten Umgebungen werden üblicherweise nur kleine Datenmengen ausgetauscht. Dies ist einerseits durch den erhöhten Resourcenbedarf auf dem Gerät bedingt. Andereseits verfügen die genutzten Geräte häufig nur über drahtlose Kommunikationsschnittstellen, die lediglich eine niedrige Datenrate bei der Übertragung ermöglichen (vgl. 6LoWPAN). Darüber hinaus ist die Größe der Daten, die mit einer Nachricht ausgetauscht werden können, durch den Einsatz von UDP begrenzt. In einigen Anwendungsfällen kann es jedoch erforderlich sein, auch größere Datenmengen zu übertragen. Ein typisches Szenario hierfür sind beispielsweise Firmware-Updates. Um dies zu ermöglichen, stellt CoAP mit dem Blockwise Transfer einen simplen Stop-And-Wait-Algorithmus zur Verfügung [17]. Dabei werden die Nutzdaten in kleinere Blöcke eingeteilt. Diese werden anschließend einzeln mit CON-Nachrichten verschickt, um die Zustellung der Nachricht zu garantieren. Daraus ergeben sich zwei wesentliche Vorteile. Zum einen kann die Datenübertragung jederzeit unterbrochen werden. Zum anderen muss im Falle eines Paketverlusts nur der einzelne Datenblock erneut gesendet werden. Um eine blockweise Übertragung zu kennzeichnen, werden die CoAP-Block-Optionen, Block1 und Block2, verwendet. Die Länge der Block-Optionen kann dabei zwischen 1 und 3 Byte liegen, je nach Gesamtmenge der zu übertragenden Daten. Beide Optionen sind wie in Abbildung 19 aufgebaut.

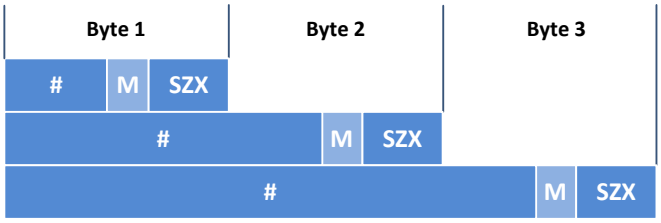

Abbildung 19: Aufbau der CoAP-Block-Option [17].

Die Block-Optionen enthalten als Optionswert die Blocknummer (#), die Blockgröße (SZX) und ein Feld, das anzeigt, ob weitere Datenblöcke folgen (M). Die Blockgröße kann hierbei zwischen 16 und 1024 Bytes liegen. Dabei kann sie entweder vom Client beim ersten Request festgelegt oder vom Server bestimmt werden (Late Negotiation). Unterliegen sowohl der Client als auch der Server Paketgrößenbeschränkungen, so wird die größte von beiden unterstützte Blockgröße gewählt. Welche der beiden Block-Optionen gewählt wird, hängt von dem jeweiligen Anwendungsfall ab. Will ein Client Daten mittels PUT- oder POST-Request an einen Server senden, so kommt die Block1-Option zum Einsatz. Dabei wird jeder Request des Clients von dem Server mit einem ACK beantwortet, welches die gleiche Block-Option enthält. Auf diese Weise wird dem Client signalisiert, dass der letzte Datenblock erfolgreich empfangen wurde. Anschließend sendet der Client den nächsten Datenblock. Dies wird wiederholt, bis alle Daten übertragen wurden. Ist dies der Fall, sendet der Server eine abschließende Response-Nachricht an den Client. Möchte ein Client hingegen Daten von einem Server mittels eines GET-Requests abrufen, wird die Block2-Option verwendet. Hierfür sendet der Client einen GET-Request mit der Block2-Option. Darauf antwortet der Server mit einer Response, die dieselbe Block-Option sowie den vom Client angefragten Datenblock enthält. Anschließend kann der Client den nächsten Datenblock mit einem separaten GET-Request abrufen. Abbildung 20 zeigt einen beispielhaften Nachrichtenaustausch für einen blockweisen GET- und PUT-Request.

2.5.4 Gruppenkommunikation in CoAP

In vielen Einsatzbereichen intelligenter verteilter Systeme, aber besonders in der Gebäudeautomation, kommt es häufig vor, dass ein und dieselbe Aktion auf mehreren Geräten nahezu zeitgleich ausgelöst werden soll. Ein Anwendungsszenario ist beispielsweise das Schließen aller Fenster und Brandschutztüren sowie das Auslösen weiterer Brandschutzmaßnahmen in einem bestimmten Gebäudeteil. Um dies zu erreichen, könnte nun das auslösende Gerät, z.B. ein Rauchmelder, an jede einzelne Fenster- und Türsteuerung einen separaten Request senden. Dieses Vorgehen ist allerdings sehr zeitintensiv und wirkt sich negativ auf den Energieverbrauch aus. Um dieses Problem zu lösen, wurde der optionale CoAP Group Communication Standard (RFC7390) spezifiziert [87]. Dieser zeigt, wie sich CoAP-Geräte zu Gruppen zusammenschließen lassen. Die Kommunikation innerhalb der

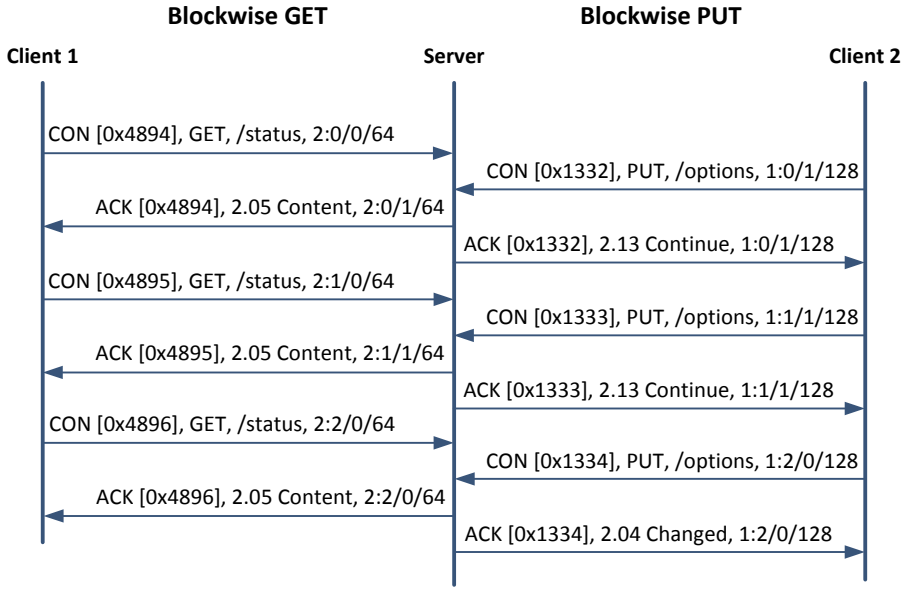

Abbildung 20: Beispielhafter Ablauf eines blockweisen GET- und PUT-Requests.

Gruppen erfolgt dann über IP-Multicast-Nachrichten. Dies hat den Vorteil, dass sie in den einzelnen Geräten leicht umgesetzt werden kann. Allerdings wird hierbei eine Netzwerkinfrastruktur benötigt, die IP-Multicasting unterstützt, da sonst nur eine link-lokale Kommunikation möglich ist. Eine CoAP-Gruppe bezeichnet dabei eine Menge von CoAP-Endpunkten, die Nachrichten empfangen und verarbeiten, die an die mit der Gruppe assoziierte IP-Multicast-Adresse gesendet werden. Sie wird entweder über ihre IP-Multicast-Adresse oder einen Hostnamen identifiziert, welcher z.B. über DNS zu der IP-Multicast-Adresse auflösbar sein muss. Ein Server, der Mitglied einer Gruppe ist, muss auf einem spezifizierten Port auf Nachrichten an die Multicast-Adresse lauschen. Dies ist üblicherweise der CoAP Default Port 5683 [99]. Nach dem Standard können auch andere Ports verwendet werden. Es ist jedoch zu beachten, dass alle Gruppenmitglieder den gleichen Port verwenden müssen. Die Gruppenzugehörigkeit eines Servers kann dabei entweder vor dem Ausbringen des Gerätes fest einprogrammiert oder zur Laufzeit konfiguriert werden. Hierzu wird im Group Communication Standard ein RESTful Interface beschrieben. Dabei werden Ressourcen-Repräsentationen dieses Interfaces im JSON-Format dargestellt und verfügen jeweils über ein Feld für den Hostnamen der Gruppe (n) und die zugehörige IP-Multicast-Adresse (a) (vgl. Abbildung 21, Group Membership Object).

```
{     "n": "All-Devices.floor1.example.com",
      "a": "[ff15::4200:f7fe:ed37:abcd]:4567" }
```

Abbildung 21: Aufbau eines Group Membership Objects [87].

Sofern möglich, wird immer die im Adressfeld gespeicherte IP-Adresse für Anfragen an die Gruppe verwendet, da so die Auflösung des Hostnamens mittels DNS enfällt. Die Ressource für die Gruppenkonfiguration kann dabei eine beliebige URI auf dem Server besitzen. Es wird jedoch empfohlen, die einzelnen Gruppenzugehörigkeiten einheitlich unter der URI *„/coap-group/<index>“* anzuordnen. Hierbei ist der Index ein beliebiger Wert, der jede Gruppenzugehörigkeit eineindeutig identifiziert. Mit einem PUT- oder POST-Request und einem Group Membership Objekt im Payload an diese URI kann nun eine Gruppenmitgliedschaft manipuliert (falls schon eine Gruppenzugehörigkeit mit dem Index vorhanden ist) oder neu angelegt werden. Mit einem DELETE können Gruppenmitgliedschaften gelöscht werden. Empfängt ein Server einen GET-Request an die URI *„/coap-group/“*, so muss er mit einer Liste all seiner Gruppenmitgliedschaften antworten. CoAP-Server, die Mitglieder einer Gruppe sind, antworten auf Anfragen an die Gruppe immer mit einer Unicast-Nachricht, um die Netzwerklast zu reduzieren.

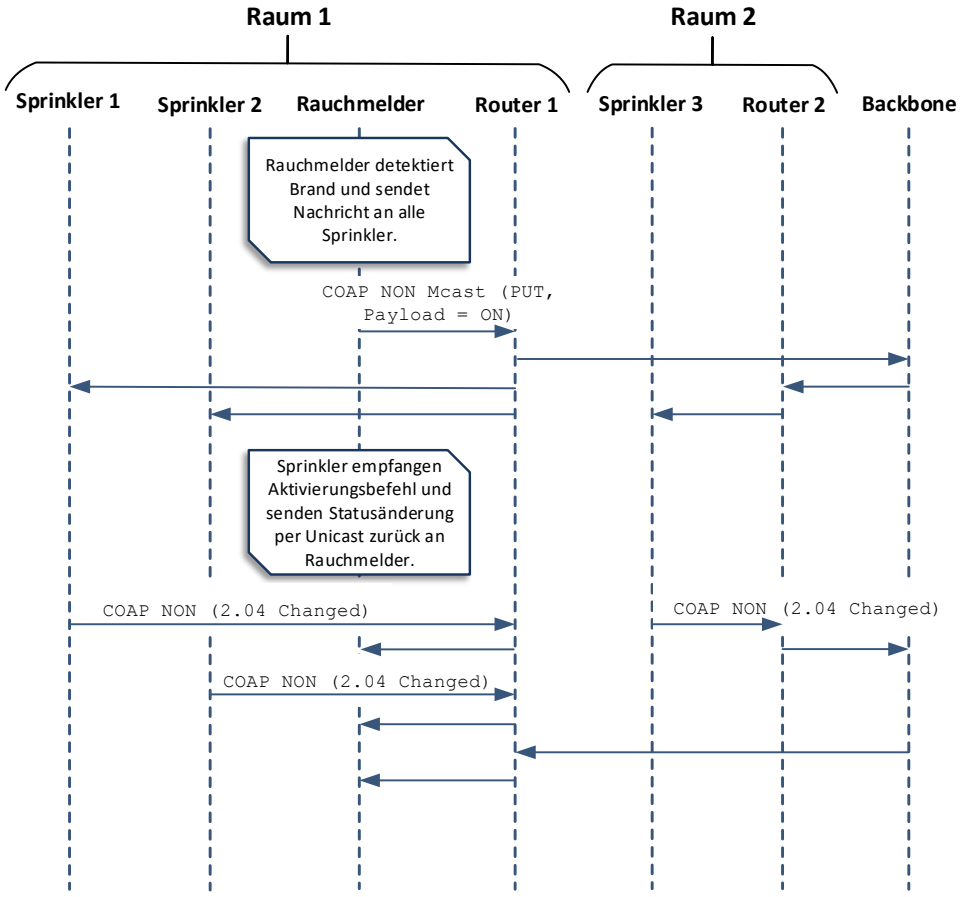

Abbildung 22: Beispiel einer Gruppenkommunikation in CoAP [87].

Abbildung 22 zeigt ein beispielhaftes Szenario mit einem Rauchmelder und mehreren Sprinkleranlagen. Der Rauchmelder detektiert einen Brand und sendet daraufhin einen Request zur Aktivierung

der Sprinkler an die entsprechende Multicast-Gruppe. Diese lösen nahezu gleichzeitig aus und senden ihrerseits eine Antwort zum Signalisieren der Statusänderung als Unicast an den Rauchmelder.

2.5.5 Anwendungsgebiete, Alternativen und Implementierungen

Aufgrund seiner hohen Flexibilität und des geringen Kommunikationsoverheads lässt sich CoAP in allen Bereichen der M2M-Kommunikation einsetzen. Der ursprünglich angedachte Zweck bestand darin, Sensornetzwerke leichter an bereits bestehende Netzwerke und das Internet anzubinden. In diesem Bereich werden vorrangig sehr leistungsschwache Geräte verwendet, welche über eine ressourcenschonende Funktechnologie wie beispielsweise IEEE 802.15.4 miteinander verbunden sind. Für den Nachrichtenaustausch wird hier hauptsächlich das Publish/Subscribe-Pattern verwendet. In diesem Einsatzszenario hat sich auch das Protokoll Message Queue Telemetry Transport (MQTT) bewährt. Bei MQTT handelt es sich jedoch um ein reines Eventing-Protokoll [11]. Dabei bieten die Geräte ihre Event-Quellen (z.B. Sensorwerte) über einen Broker an. Clients, die Änderungen der Sensorwerte abonnieren wollen, schreiben sich bei dem Broker für die entsprechenden Notifications ein. Die Sensorknoten müssen Events nur noch an die Broker melden. Diese verteilen die neuen Sensorwerte dann an die Abonnenten. Der größte Nachteil von MQTT besteht dabei darin, dass mit dem Broker eine zentrale Instanz benötigt wird [51]. Bei einem Ausfall des Brokers können auch keine Events mehr verteilt werden.

Ein weiteres Anwendungsgebiet von CoAP liegt in der Gebäudeautomation. Im Gegensatz zu Sensornetzwerken kommen hier viele Geräte zum Einsatz, welche über eine höhere Rechenkapazität und eine permanente Stromversorgung verfügen. Im Vordergrund steht dabei die einfache Anbindung der Geräte an Firmennetze oder das Internet, um beispielsweise die Fernwartung des Gebäudes zu ermöglichen. In diesem Anwendungsgebiet kann KNX als bewährte Alternative angesehen werden. KNX definiert alle Netzwerkschichten ab dem Link Layer aufwärts (vgl. OSI). Daher wird zwingend ein Gateway zur Anbindung an herkömmliche IP-Netze benötigt [72]. Darüber hinaus stellt KNX keinerlei Sicherheitsmaßnahmen bereit, was besonders im Bereich der Gebäudesteuerung zu einem erheblichen Risiko führt. Aufgrund der leistungsstärkeren Geräte kommt in diesem Bereich auch DPWS für die Anbindung in Frage. Weiterhin kann CoAP im Gesundheits- und Pflegewesen eingesetzt werden [28]. Hier werden vornehmlich Kleinstgeräte verwendet, da diese nahezu unsichtbar untergebracht werden können und sich so positiv auf die Patientenwahrnehmung auswirken. Dabei liegt der Fokus darauf, den Patienten in seinem heimischen Umfeld medizinisch zu überwachen oder bei der Bewältigung des Alltags zu unterstützen. Alternativen in diesem Bereich sind hauptsächlich DPWS und ZigBee,wobei DPWS sehr viel Overhead aufweist. Neben der Gebäude- bietet auch die Heimautomation ein breites Anwendungsfeld für CoAP. Dabei sollen vornehmlich Komfortfunktionen über Nutzergeräte, wie beispielsweise Smartphones, aus dem Heimnetzwerk oder dem Internet heraus gesteuert werden. In diesem Bereich ist die Z-Wave-Technologie sehr verbreitet [45]. Dabei handelt es

sich um ein Drahtlos-Protokoll mit kurzer Reichweite, welches in seinen Anwendungsmöglichkeiten sehr flexibel ist. Allerdings werden hierfür proprietäre Funkmodule benötigt, sodass für den Zugriff über andere Endgeräte wieder Gateways benötigt werden [72].

Implementierung	**Sprache**	**Funktionsumfang**
Californium	Java	CoAP Basis
		CoAP Block Transfer
		CoAP Observe
		CoAP Group Communication
		CoAP Resource Directory
		CoAP over DTLS
Erbium	C	CoAP Basis
		CoAP Observe
		CoAP Block Transfer
libcoap	C	CoAP Basis
		CoAP Observe
		CoAP Block Transfer
Copper	JavaScript	CoAP Basis
		CoAP Observe
		CoAP Block Transfer

Tabelle 3: Übersicht über die wichtigsten CoAP-Implementierungen.

Bisher sind viele Implementierungen von CoAP in den verschiedensten Programmiersprachen entstanden. Die wichtigsten unter ihnen sind in Tabelle 3 zusammengefasst. Hierbei sind zunächst die drei Implementierungen von Kovatsch et al. zu nennen. Bei Californium handelt es sich im eine Java-Implementierung mit sehr hohem Funktionsumfang [65]. Sie wird sehr häufig zum Entwurf von CoAP-Applikationen verwendet und dient als Referenzimplementierung im Rahmen der Eclipse Foundation. Erbium wurde hingegen in C umgesetzt und für den Einsatz auf sehr stark ressourcenbeschränkten Geräten konzipiert [63]. Erbium ist dabei in das Contiki-Betriebssystem für eingebettete Systeme integriert, was die Anzahl kompatibler Geräte stark einschränkt. Eine Alternative hierzu ist die C-Implementierung von Bergmann et al., libcoap [14]. Sie ist zwar ebenfalls in ein Betriebssystem für eingebettete System integriert (TinyOS), kann allerdings auch unabhängig von diesem verwendet werden. Bei Copper handelt es sich um ein von Kovatsch et al. entwickeltes Plugin für den Web Browser FireFox auf JavaScript-Basis [64]. Copper kann, anders als die vorher genannten Implementierungen, nur als CoAP Client genutzt werden. Daher kommt es häufig zum Testen anderer CoAP-Anwendungen zum Einsatz. Es existieren noch viele weitere CoAP-Implementierungen, die nur von geringer Bedeutung sind, da sie nicht dem neusten Stand der CoAP-Spezifikation entsprechen oder nur über einen sehr eingeschränkten Funktionsumfang verfügen. Obwohl bereits so eine große Vielfalt von CoAP-Implementierungen existiert, wurde keine der bisher genannten Umsetzungen auf ihre Echtzeitfähigkeit hin untersucht.

2.6 Vergleich von Web Services, REST und anderen Protokollen

Web Services und REST-basierte Systeme verfügen beide jeweils über sehr gute Eigenschaften in Bezug auf Skalierbarkeit, Interoperabilität und Flexibilität, bringen aber auch Nachteile mit sich. Welcher der beiden Ansätze zu bevorzugen ist, hängt dabei stark vom Einsatzszenario ab. Web Services erlauben eine detaillierte und erweiterbare Beschreibung von Diensten und deren Ein- und Ausgabeparametern. Sie lassen sich daher leicht mit semantischen Informationen anreichern, die weit über eine einfache Typenbezeichnung der Ein- oder Ausgabewerte hinaus gehen. Dies wirkt sich positiv auf die Interoperabilität aus, da die Beschreibung der Funktionalität eines Dienstes und die daraus resultierenden Auswirkungen auch von dem Dienstnutzer, dem Client, verstanden werden müssen. Forschungsprojekte wie Or.Net oder BaaS (Building as a Service) versuchen diese Eigenschaft zu nutzen, um domänenspezifische Semantiken für die Dienstebeschreibung im Medizinbereich und der Gebäudeautomation zu definieren [19, 56]. Im Fall von Or.Net führte dies sogar zur Schaffung eines eigenen Standards. Web Services sind dabei allerdings eher für leistungsfähigere Systeme geeignet. Zwar wurde mit DPWS ein eigenes Profil speziell für den Einsatz auf ressourcenbeschränkten Geräten entwickelt. Jedoch führt die Verwendung von XML-basierten, menschenlesbaren Datenformaten wie SOAP und WSDL weiterhin zu einem Missverhältnis zwischen den übertragenen Nutz- und Protokolldaten. So entstehen selbst für kleinste Datenmengen große Nachrichten. Für Kleinstgeräte mit begrenztem Energievorrat ist dies von Nachteil, da die beim Senden verbrauchte Energie insbesondere bei drahtloser Kommunikation besonders hoch ist und mit der Nachrichtengröße ansteigt. Auch das Auslesen der Nutzdaten aus den XML-basierten SOAP-Nachrichten, das sogenannte Parsen, erfordert erheblichen Rechenaufwand. Dies kann bei Geräten mit geringer Rechenleistung zu sehr langen Antwortzeiten führen.

Zwar haben die Arbeiten von Kyusakov et al. und Moritz et al. gezeigt, dass die Nachrichtengrößen durch Kompressionstechniken wie Efficient XML Interchange (EXI) stark reduziert werden können, allerdings entsteht hier zusätzlicher Aufwand bei der Kompression [68, 76]. Die Arbeiten von Lerche et al. und das daraus resultierende uDPWS zeigen darüber hinaus, dass sich der Speicherbedarf von DPWS erheblich verringern lässt [71]. Dies wird allerdings durch Techniken erreicht, welche die Flexibilität und den Feature-Umfang von DPWS stark einschränken. Einige Funktionen, wie WS-Security, werden in uDPWS gar nicht unterstützt. Weiterhin setzt DPWS verstärkt auf die Nutzung von TCP auf der Transportschicht. Dies schränkt die Eignung von DPWS für die Nutzung in Echtzeitsystem ein, da TCP durch seine unvorhersehbaren Anpassungen der Datenrate und automatische Sendewiederholung im Fehlerfall zu einem nicht-deterministischen Zeitverhalten der Kommunikation führt.

CoAP stellt eine vielversprechende REST-basierte Alternative zu den klassischen Web Services dar. Im Gegensatz zu dem weit verbreiteten HTTP verfügt es über wichtige Funktionalitäten zur M2M-Kommunikation, wie Discovery- und Eventing-Mechanismen. Dabei verwendet es, anders als DPWS,

aber sehr kleine binär kodierte Nachrichtenheader. Dies führt zu einem wesentlich besseren Verhältnis zwischen den Nutz- und Protokolldaten. Auf diese Weise reduziert sich außerdem der Aufwand zum Parsen der Nachrichten, was kürzere Antwortzeiten nach sich zieht. Weiterhin bietet CoAP durch die Nutzung von UDP bessere Voraussetzungen für den Einsatz in Echtzeitsystemen. Im Gegensatz zu TCP nimmt UDP keine Anpassungen der Datenrate oder automatische Sendewiederholungen vor. Dies führt zwar zu einer geringeren Zuverlässigkeit der Paketübertragung, eliminiert damit jedoch auch mögliche Quellen für ein zeitlich nicht-deterministisches Verhalten der Kommunikation.

Der große Nachteil von CoAP besteht in der fehlenden Standardisierung bei der Beschreibung von Ressourcen. Der CoAP-Standard selbst sieht keine Beschreibung vor. Daher ist es auch schwierig, bereitgestellte Dienste mit einer Semantik anzureichern. Dies kann zwar beispielsweise implizit durch die Strukturierung der Ressourcen-URI geschehen, jedoch bieten sich hier deutlich weniger Möglichkeiten als bei DPWS. Andere Ansätze verfolgen ein ähnliches Konzept wie DPWS. Dabei wird eine Ressourcen-Beschreibung unter einer speziell definierten URI angeboten [90]. Zwar wird hier mit der Web Application Description Language (WADL) auch wieder ein XML-basiertes Format verwendet. Allerdings erfolgt das Abrufen der Ressourcenbeschreibung für jeden Client nur einmal. Zur Nutzung der Ressource werden weiterhin binär kodierte Nachrichten verwendet. Bei DPWS kommen jedoch auch zur Nutzung von Diensten XML-basierte Nachrichtenformate zum Einsatz, sodass der Overhead für die Kommunikation deutlich größer ausfällt.

Fysarakis et al. stellen DPWS, CoAP und das MQTT direkt gegenüber [41]. Sie evaluieren die CPU-Auslastung, den Speicherbedarf und das Zeitverhalten der drei Protokolle am Beispiel eines Policy-basierten Zugriffskontrollsystems. Die Umsetzung mit DPWS weist dabei eine ca. 30% größere Antwortzeit auf als die nahezu gleichwertigen Umsetzungen mit CoAP und MQTT. Ähnliches zeigt sich auch beim Speicherbedarf von DPWS, der gegenüber CoAP um 16% höher ausfällt. Die Messergebnisse für die CPU-Auslastung müssen mit Blick auf die Versuchsanordnung von Fysarakis et al. interpretiert werden. In ihrer Testanwendung wird eine neue Anfrage gesendet, sobald die Antwort auf die letzte Anfrage eingetroffen ist. Dies führt dazu, dass DPWS in der Untersuchung die geringste CPU-Auslastung aufweist. Aufgrund der längeren Antwortzeit werden mit DPWS deutlich weniger Anfragen je Zeiteinheit versandt bzw. Antworten verarbeitet. CoAP und MQTT zeigen sich in etwa gleichwertig, wobei MQTT einen leichten Geschwindigkeitsvorteil aufweist. Jedoch besitzt MQTT mit dem Broker den Nachteil eines Single Point of Failure (SPoF). Zudem erzeugt MQTT eine deutlich höhere Netzwerklast, da alle Nachrichten über den Broker geleitet werden müssen.

Unter Berücksichtigung aller Aspekte zeigt sich, dass CoAP auf Grund seiner Leichtgewichtigkeit, seiner Flexibilität und der Verwendung von UDP als Transportprotokoll besser für den Einsatz in Echtzeitsystemen geeignet ist als DPWS und MQTT. Im weiteren Verlauf dieser Arbeit soll daher das Constrained Application Protocol im Vordergrund stehen.

3 Echtzeit-Kommunikation für Geräte

3.1 Grundlagen

3.1.1 Echtzeit - Begriffsdefinition und Kategorisierung

Ein echtzeitfähiges System bezeichnet nach Young jedes System, dass innerhalb einer endlichen und in ihrer Länge festgelegten Zeitspanne auf extern generierte Eingabestimuli reagiert [113]. Eine ähnliche Definition findet sich auch in der Norm DIN 44300. Hier heißt es wörtlich:

„Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlichen zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.“

DIN 44300

Aus diesen Begriffsdefinitionen lassen sich die Anforderungen an ein Echtzeitsystem ableiten. Ein wesentliches Kriterium ist die Rechtzeitigkeit der Reaktion auf äußere Eingaben. Hierbei ist es zunächst unerheblich, wie viel Zeit die Verarbeitung der Eingabe erfordert, solange die Antwortzeit eine festgelegte Zeitspanne nicht überschreitet. Diese Zeitspanne wird auch als Deadline bezeichnet und ihre Länge hängt einzig vom jeweiligen Anwendungsfall ab. Implizit ergibt sich hieraus eine zweite wesentliche Anforderung an Echtzeitsysteme. Um die Einhaltung von Deadlines zu garantieren, müssen sich Echtzeitsysteme zeitlich deterministisch verhalten. Ein System ist zeitlich deterministisch, wenn seine Reaktionszeit unter Berücksichtigung der Rahmenbedingungen, wie den Eingabedaten, stets vorhersehbar ist.

Kategorisierung von Echtzeitsystemen

Echtzeitsysteme lassen sich nach verschiedenen Gesichtspunkten einteilen. Die in der Literatur vorherrschende Kategorisierung basiert dabei auf den Konsequenzen, die sich aus der Überschreitung von Deadlines ergeben [88]. Es wird hier zwischen drei Kategorien unterschieden. In hartechtzeitfähigen Systemen führt jede Überschreitung einer Deadline zu einem Systemfehler. Dies kann zum Beispiel bei vollautomatisierten Fertigungsstraßen der Fall sein. Reagiert ein Teil dieser Kette zu spät, kann dies sehr ernste Folgen haben. Tritt nicht sofort ein Systemfehler auf, so spricht man entweder von strenger oder weicher Echtzeit. Diese beiden Formen der Echtzeit tolerieren jeweils sporadische Deadline-Überschreitungen. Sie unterscheiden sich lediglich in dem Nutzen von Informationen, die nach der Deadline eintreffen. Bei strenger Echtzeit, ist der Nutzen verspätet eintreffender Informationen Null. Dies ist beispielsweise bei Streaming-Anwendungen der Fall, die ein Live-Bild eines

Prozesses innerhalb einer Fabrik übertragen. Hier sind verspätet erhaltene Bildinformationen wertlos, da sie nicht den aktuellen, sondern einen in der Vergangenheit liegenden Zustand zeigen. Bei weicher Echtzeit hingegen nimmt der Nutzen der Informationen mit der Dauer der Deadline-Überschreitung ab. Solche Systeme kommen häufig bei der Überwachung sich sehr langsam ändernder Prozesse, z.B. in der Medizintechnik oder der Chemie-Industrie, zum Einsatz. Abbildung 23 stellt die Unterschiede zwischen den drei Kategorien graphisch dar.

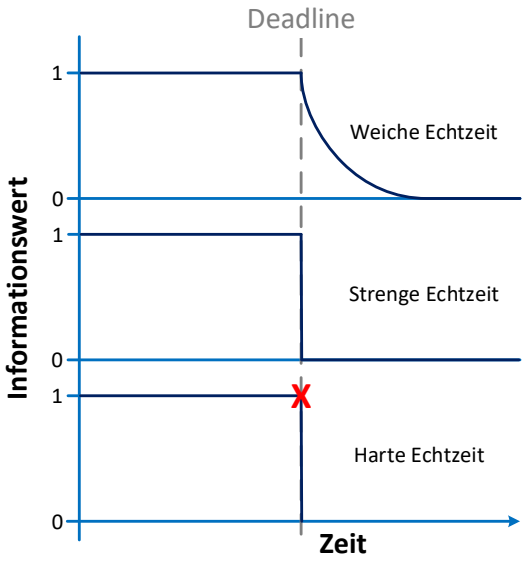

Abbildung 23: Gegenüberstellung von harter, strenger und weicher Echtzeit.

Andere Arbeiten nehmen eine Einordnung von Echtzeitsystemen anhand ihrer maximalen Antwortzeit vor [37, 38]. So definieren Felser und Fettweis drei Klassen von Echtzeit anhand der maximalen Zykluszeit. Die Zykluszeit beschreibt dabei die zeitliche Periode, in der jedes Gerät im Netzwerk mindestens einmal auf das Netzwerk zugreifen kann. Die Zykluszeit lässt sich mit der maximalen Antwortzeit gleichsetzen, da das angefragte Gerät in diesem Zeitraum mindestens einmal den Zugriff auf das Netzwerk erhält und auf die Anfrage antworten kann. Jede der drei Klassen wird dabei einem bestimmten Anwendungsbereich zugeordnet. Tabelle 4 gibt eine Übersicht zu den Echtzeitklassen.

Klasse	**Zykluszeit**	**Anwendungsgebiet**
1	100 ms	Mensch-Maschine-Interaktion
2	10 ms	Prozessmonitoring
3	1 ms	Robotik, Industrie, Automotive

Tabelle 4: Die von Felser und Fettweis definierten Echtzeitklassen und ihre vorwiegenden Anwendungsfelder [37, 38].

Danielis et al. greifen die drei Echtzeitklassen auf und belegen sie wie auch [78] mit den Begriffen weiche (Klasse 1), harte (Klasse 2) und isochrone (Klasse 3) Echtzeit [23]. Da dies jedoch zu einer

Doppelbelegung von Begriffen führt, wird in dieser Arbeit aus Gründen der Klarheit die numerische Klassenbezeichnung verwendet.

Eine weitere Kategorisierung von Echtzeitkommunikationssystemen nehmen Danielis et al. nach der verwendeten Hardware und den genutzten Protokollen vor [23]. Hierbei unterscheiden sie, ob Standard-Hardware (Commercial Off The Shelf Hardware - COTS Hardware) und Standard-Protokolle zum Einsatz kommen oder durch proprietäre Lösungen ersetzt werden. Daraus ergeben sich die in Abbildung 24 dargestellten Kategorien.

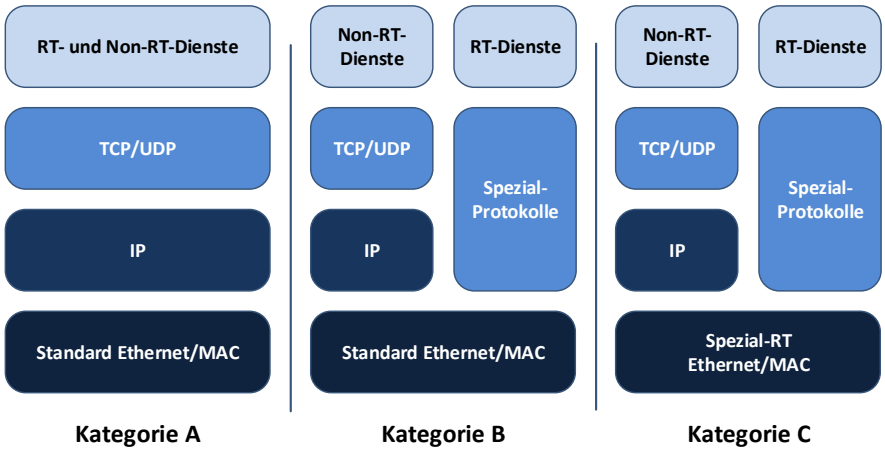

Abbildung 24: Kategorisierung von Echtzeitkommunikationssystemen nach der verwendeten Hard-/Software [23].

Danielis et al. heben besonders die erheblichen Vorteile von Systemen der Kategorie A hervor. So erhöht der Einsatz von Standard-Hardware die Interoperabilität und die Integrierbarkeit in bereits bestehende Netzwerke. Darüber hinaus sind die Anschaffungskosten von Standard-Hardware auf Grund des großen Marktes deutlich geringer als von Spezial-Hardware. Zudem begünstigt der Einsatz spezialisierter Hardware und proprietärer Protokollanpassungen den Vendor-Lock-in.

Im weiteren Verlauf dieser Arbeit wird für die Einteilung nach harter, strenger und weicher Echtzeit die klassische Definition aus der Literatur zugrunde gelegt. Im Übrigen wird die Klassifizierung von Echtzeitsystemen anhand ihrer zeitlichen Anforderungen von Felser und Fettweis verwendet. Weiterhin erfolgt eine Kategorisierung der Hardware- und Softwareanforderungen nach Danielis et al. Als besonders erstrebenswert wird dabei ein hart echtzeitfähiges Klasse-3-System der Kategorie A betrachtet, da es die höchsten zeitlichen Anforderungen erfüllt und dabei einen hohen Grad an Interoperabilität gewährleistet.

3.1.2 Übertragungsmedien und Zugriffsverfahren

Die Datenübertragung zwischen zwei Systemen kann über verschiedene Übertragungsmedien erfolgen. Bei der drahtgebundenen Kommunikation werden die Daten beispielsweise mit Hilfe elektrischer Signale über ein elektrisch leitfähiges Kabel übertragen. Im Fall von drahtloser Kommunikation dient hingegen die Luft als Kommunikationsmedium. Hier werden die Daten in Form von elektromagnetischen Wellen übertragen. Darüber hinaus kann die Kommunikation zwischen zwei Geräten auch optisch stattfinden. Dabei werden Lichtimpulse über einen Lichtwellenleiter an den Empfänger gesendet. Die Kommunikation mittels Lichtsignalen kann auch über die Luft erfolgen (vgl. Infrarot). Hierzu ist allerdings eine direkte und unblockierte Sichtlinie nötig, um die Übertragung nicht zu stören.

Beim Senden mehrerer Netzwerkteilnehmer über ein gemeinsam genutztes Kommunikationsmedium können je nach Beschaffenheit des Mediums und Art des Zugriffs Kollisionen entstehen. Eine Kollision bezeichnet das zeitgleiche Senden von zwei oder mehr Netzwerkteilnehmern über ein gemeinsames Medium, in dessen Folge sich die Kommunikationsvorgänge gegenseitig so beeinträchtigen, dass die Nachrichten vom jeweiligen Empfänger nicht mehr gelesen werden können. Kollisionen können die Echtzeiteigenschaften eines Systems nachhaltig beeinflussen, da sie unvorhergesehen auftreten. Um den konkurrierenden Zugriff auf das Medium zu steuern, können verschiedene Techniken zum Einsatz kommen und auch mit einander kombiniert werden. Die Umsetzung der jeweiligen Verfahren kann dabei zentralisiert oder verteilt erfolgen. Bei zentralisierten Verfahren werden von einer zentralen Instanz Zugriffspläne für alle Netzwerkteilnehmer erarbeitet und im Netzwerk verteilt. Bei verteilten Verfahren kann diese Aufgabe entweder von mehreren Instanzen im Netzwerk wahrgenommen werden oder die Netzwerkteilnehmer handeln den Medienzugriff untereinander aus. Die Verfahren lassen sich zumeist in die vier nachfolgenden Kategorien unterteilen.

Time Division Multiple Access

Beim Time Division Multiple Access (TDMA) handelt es sich um einen Zugriffsmodus, bei dem die verschiedenen Netzwerkteilnehmer niemals zeitgleich auf das Übertragungsmedium zugreifen. Hierzu wird die Zeitachse in einzelne Zeitschlitze (engl.: Time Slots) eingeteilt. Dabei kann die Einteilung entweder kontinuierlich oder zyklisch erfolgen, sodass sich die Abfolge der Zeitschlitze stets wiederholt. Die Periode eines Zeitzyklus wird auch als Cycle Time bezeichnet. Jedem Netzwerkteilnehmer wird einer dieser Zeitschlitze zugeteilt, in dem er dann exklusiv auf das Medium zugreifen kann. Abbildung 25 zeigt dies beispielhaft für vier Netzwerkteilnehmer N1 bis N4 und eine zyklische Einteilung der Zeitschlitze T1 bis T4.

Um die zeitlich korrekte Abfolge der Netzwerkzugriffe zu ermöglichen und einen zeitgleichen Zugriff mehrerer Teilnehmer auszuschließen, ist eine gemeinsame Zeitbasis für alle Teilnehmer erforderlich. Dies kann mit Hilfe von unterschiedlichen verteilten und zentralisierten Verfahren zur Zeit-

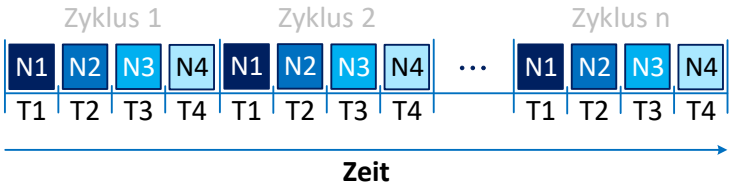

Abbildung 25: Zeitlich wechselnde Medienzugriffe bei der Verwendung eines TDMA-Verfahrens.

synchronisation erreicht werden. Einen Sonderfall des TDMA-basierten Zugriffs bilden sogenannte Token-basierte Verfahren, die ohne eine gemeinsame Zeitbasis auskommen. Dabei wird eine Marke beziehungsweise ein Token von einem Netzwerkteilnehmer an den nächsten übertragen. Der Besitzer des Tokens darf jeweils uneingeschränkt auf das Netzwerk zugreifen. Aufgrund ihrer inhärenten Kollisionsfreiheit bilden TDMA-basierte Verfahren den klassischen Ansatz für die Echtzeitkommunikation.

Frequency Division Multiple Access

Frequency Division Multiple Access (FDMA) basierte Verfahren werden zumeist in der drahtlosen Kommunikation eingesetzt. Dabei wird das zur Übertragung genutzte Frequenzsprektrum in mehrere Bereiche eingeteilt, welche auch als Kanäle bezeichnet werden. Zwischen den einzelnen Kanälen werden dabei Puffer-Frequenzen definiert, die keinem der Kanäle zugeordnet sind. Bei der richtigen Wahl dieser Puffer werden Interferenzen zwischen den Kanälen verhindert. Dies führt dazu, dass mehrere Kommunikationsvorgänge zeitgleich auf unterschiedlichen Kanälen stattfinden können, ohne sich gegenseitig zu beeinflussen. Zwei Kanäle, für die diese Bedingung erfüllt ist, werden auch als orthogonal bezeichnet. Abbildung 26 verdeutlicht das Prinzip des FDMA.

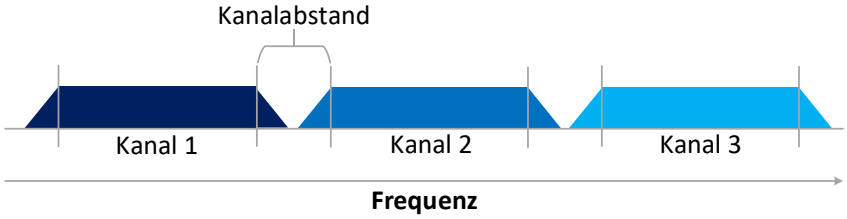

Abbildung 26: Aufteilung des Frequenzraumes in einem FDMA-Verfahren.

In der Praxis können FDMA-Verfahren jedoch nur in Kombination mit anderen Ansätzen, wie z.B. TMDA, zur Kollisionsvermeidung verwendet werden. Andernfalls müsste jedem Netzwerkteilnehmer exklusiv ein Kanal zugeordnet werden. Dies hätte zur Folge, dass jeder Knoten in einem Netzwerk mit n Teilnehmern auch über n Netzwerkschnittstellen verfügen muss, um auf jedem Kanal lauschen zu können.

Space Disivion Multiple Access

Auch beim Space Division Multiple Access (SDMA) kann der Zugriff auf das Medium zeitgleich erfolgen. Hier werden die Kommunikationsvorgänge räumlich so getrennt, dass sie sich nicht gegenseitig beeinflussen können. Dies kann beispielsweise bei der drahtgebundenen Kommunikation durch die Nutzung unterschiedlicher Verbindungskabel erfolgen. Bei der drahtlosen Kommunikation stellen sich auf Grund der begrenzten Sendereichweite der Geräte automatisch SDMA-Effekte ein. Gezielt können diese jedoch nur mittels Richtfunktechnik ausgenutzt werden.

Code Division Multiple Access

Anders als beim FDMA werden beim Code Division Multiple Access (CDMA) mehrere Nachrichten zeitlich überlagert im gesamten zur Verfügung stehenden Frequenzspektrum versendet, auf die Einteilung fester Kanäle wird verzichtet. Dabei kommen unterschiedliche Spreizcodes zur Kodierung der Nachrichtensignale zum Einsatz, sodass durch Anwendung der entsprechenden Dekodierungsverfahren vom Empfänger alle Nachrichten wieder aus dem überlagerten Signal rekonstruiert werden können. Die Spreizkodes müssen dabei bestimmte Eigenschaften erfüllen (z.B. Orthogonalität, siehe hierzu auch [106]).

3.1.3 Ethernet

Bei Ethernet handelt es sich um eine drahtgebundene Netzwerktechnologie, welche die Bitübertragungsschicht und die Sicherungsschicht definiert. Ethernet wurde 1983 erstmals durch das IEEE als IEEE 802.3 standardisiert. Seither wurde der Standard vielfach weiterentwickelt und erweitert, um höhere Bandbreiten, andere Medienzugriffsverfahren und Management-Funktionen zu unterstützen. Bei Ethernet werden die zu sendenden Daten in mehrere Blöcke, sogenannte Frames, aufgespalten und nacheinander versendet. Jedem Frame wird dabei der Ethernet-Header vorangestellt. Hinter den Daten wird jeweils noch eine Prüfsumme zur Kontrolle der Datenintegrität angehängt. Abbildung 27 zeigt die Struktur eines Ethernet-Frames.

Der Ethernet-Header enthält eine Präambel, die sich aus alternierenden Nullen und Einsen zusammensetzt und der Synchronisation zwischen den einzelnen Netzwerkteilnehmern dient. Weiterhin enthält der Header einen Start-Delimiter, der den Beginn eines Frames markiert, sowie die Ziel- und Quell-MAC-Adresse. Das Typen-Feld kann auf zwei Arten interpretiert werden. Zum einen kann es die Länge des Frames in Bytes angeben (alle Werte < 1500). Zum anderen kann es dazu verwendet werden, das übergeordnete und in dem Frame gekapselte Protokoll zu spezifizieren (alle Werte > 1536). Auf das Typen-Feld folgen die Nutzdaten und eine Prüfsumme zur Kontrolle der Datenintegrität.

Um den Zugriff auf ein geteiltes Kommunikationsmedium zu regulieren, unterstützt Ethernet mehrere Verfahren. Diese sollen jedoch hier nicht näher erläutert werden, da Kollisionen in modernen Ethernet-Netzwerken nur noch eine untergeordnete Rolle spielen. Durch den Einsatz von Switches

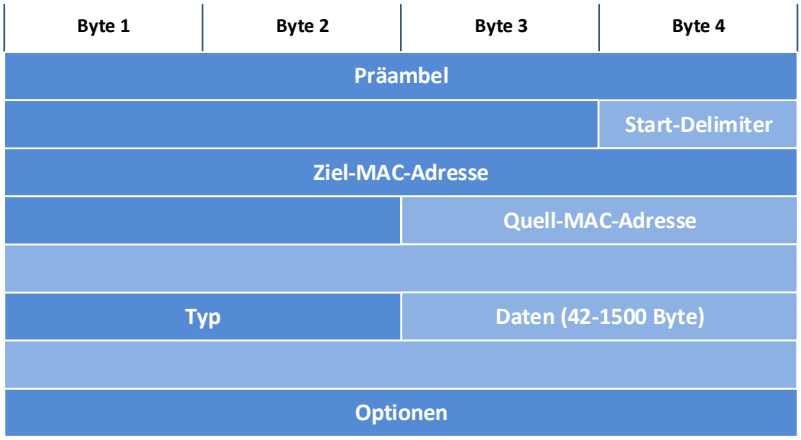

Abbildung 27: Struktur eines Ethernet-Frames [4].

wird aus dem geteilten Kommunikationsmedium eine Aneinanderreihung von voll-duplex-fähigen Punkt-zu-Punkt-Verbindungen, bei der die Nachrichten zunächst die jeweiligen Switches durchlaufen müssen. In den Switches werden die Nachrichten zwischengepuffert und nacheinander an den jeweiligen Netzwerkteilnehmer weitergeleitet. Auf diese Weise ist ein Auftreten von Kollisionen nahezu unmöglich. Dennoch ist das Zeitverhalten von Ethernet nur schwer vorhersagbar. Dies liegt zum einen an der variablen Verarbeitungszeit der Frames in den Switches. Zum anderen hängt die tatsächliche Zeit bis zur Weiterleitung der Frames durch den Switch von der Anzahl der bereits zwischengepufferten Nachrichten ab. Daraus folgt, dass die für eine Ende-zu-Ende-Übertragung benötigte Zeit direkt vom Nachrichtenaufkommen im Netzwerk insgesamt beeinflusst wird.

3.1.4 Wireless LAN

Unter Wireless LAN versteht man eine Reihe von Technologien zum Aufbau drahtloser lokaler Netzwerke, die von dem IEEE in der Standardfamilie IEEE 802.11 zusammengefasst wurden. Wie auch bei Ethernet beziehen sich die Standarddefinitionen auf die unteren beiden Schichten des ISO/OSI-Referenzmodells. Die erste Version des Standards wurde bereits 1997 veröffentlicht. Später folgten weitere Standards, die zu einer stetigen Erhöhung der Bandbreite der Kommunikation führten und die Nutzung von Mehrantennentechniken (MIMO) ermöglichten. Tabelle 5 gibt einen Überblick über die einzelnen Teilstandards und ihre Besonderheiten.

WLAN-Netzwerke können in drei verschiedenen Modi betrieben werden. Im Ad-hoc-Modus lassen sich Punkt-zu-Punkt-Verbindungen zwischen zwei einzelnen Geräten aufbauen. Der Infrastruktur-Modus ermöglicht den Aufbau von Netzwerken zwischen mehreren Geräten über ein zentrales Infrastrukturelement, den Access Point. Einzelne Netzwerke im Infrastruktur-Modus können zudem über ein drahtgebundenes Backbone-Netzwerk, meist via Ethernet, miteinander verbunden werden. Der

WLAN Standard	Frequenz-Band	Max. Datenrate (Brutto)	MIMO-Unterstützung
802.11	2,4 GHz	2 MBit/s	-
802.11b	2,4 GHz	11 MBit/s	-
802.11a	5 GHz	54 MBit/s	-
802.11g	2,4 GHz	54 MBit/s	-
802.11n	2,4 GHz oder 5 GHz	600 MBit/s	X
802.11ac	5 GHz	6,9 GBit/s	X

Tabelle 5: Übersicht der aktuell verfügbaren Teilstandards der 802.11-Standard-Familie für die Bitübertragungsschicht [52].

dritte Modus erlaubt es, sogenannte Mesh-Netzwerke aufzubauen, in denen die einzelnen WLAN-Geräte zeitgleich als Endpunkte und Weiterleitungsknoten mit Routing-Funktionalität agieren. Bei der Kommunikation über WLAN werden die Daten, wie auch bei Ethernet, in Frames eingeteilt, wobei vor jeden Frame der WLAN-Header gesetzt wird.

Bei der Datenübertragung können verschiedene Kanäle genutzt werden. Die Anzahl der verfügbaren Kanäle hängt dabei vom verwendeten WLAN-Standard und dem Frequenzband ab. WLAN erlaubt nach der neusten Standardversion die Nutzung des 2,4-GHz- und des 5-GHz-Bandes. Im 2,4-GHz-Band stehen insgesamt 14 Kanäle mit einer Kanalbreite von jeweils 20 MHz und einem Abstand von jeweils 5 MHz zur Verfügung, von denen in Europa ausschließlich die ersten 13 Kanäle zur Nutzung zugelassen sind. Durch den geringen Kanalabstand überlappen sich die Frequenzbereiche benachbarter Kanäle, sodass von den 13 Kanälen bei Einhaltung des Mindestabstands von 20 MHz nur maximal 3 Kanäle gleichzeitig zueinander orthogonal sind. Im 5-GHz-Frequenzband sind in Europa 19 Kanäle für die nicht zweckgebundene Nutzung freigegeben. Der Kanalabstand im 5 GHz-Band beträgt 20 MHz, bei einer Kanalbreite von üblicherweise 20 MHz. Die Kanalbreite kann jedoch auch bis zu 160 MHz betragen, wobei sich die Anzahl der Kanäle entsprechend verringert. Aus der Aufteilung des Frequenzbandes ergibt sich im 5-GHz-Band eine deutlich höhere Anzahl orthogonaler Kanäle, was die Störanfälligkeit der Kommunikation verringert. Bei der WLAN-Kommunikation können jeweils Kollisionen zwischen zwei gesendeten Frames auf ein und demselben Kanal, aber auch auf zwei zueinander nicht-orthogonalen Kanälen entstehen. Um das Kollisionsrisiko möglichst gering zu halten, wird für die Kontrolle des Medienzugriffs das CSMA-Verfahren (Carrier Sense Multiple Access) genutzt. Bei CSMA lauscht ein Netzwerkteilnehmer vor dem Versenden eines Frames auf dem Kommunikationsmedium, ob bereits ein Sendevorgang eines anderen Netzwerkteilnehmers über das gemeinsam genutzte Medium stattfindet. Ist das Medium bereits belegt, wartet der Netzwerkteilnehmer mit dem Senden seines Frames bis das Medium wieder frei ist. So wird verhindert, dass sich die versendeten Frames zweier Netzwerkteilnehmer auf dem Medium überlagern und so für den jeweiligen Empfänger nicht mehr interpretierbar sind. Starten zwei Netzwerkteilnehmer ihren Sende-

vorgang jedoch gleichzeitig, erscheint das Kommunikationsmedium für beide als frei. In diesem Fall kann es weiterhin zu einer Kollision kommen. Um die Wahrscheinlichkeit von Kollisionen weiter zu verringern wird CSMA bei WLAN um die sogenannte Collision Avoidance (CA) erweitert. Hierbei lauscht ein Netzwerkteilnehmer vor dem Senden auf dem Kommunikationsmedium. Ist das Medium bereits durch einen anderen Kommunikationsteilnehmer besetzt, wartet der Netzwerkteilnehmer für einen zufällig gewählten Zeitraum. Nach Ablauf der Wartezeit wird wieder überprüft, ob das Kommunikationsmedium belegt ist. Für jeden Kommunikationsversuch, bei dem das Medium bereits belegt ist, wird die Wartezeit für den nächsten Kommunikationsversuch exponentiell erhöht. Dieses Vorgehen beim Festlegen der Wartezeit wird auch als *Exponential Back-Off* bezeichnet. Bei der drahtlosen Kommunikation ergeben sich jedoch noch andere Probleme (vgl. Hidden Station Problem).

Aus der Verwendung von CSMA/CA und der damit verbundenen exponentiell ansteigenden Back-Off-Zeit entstehen jedoch erhebliche zeitliche Unsicherheiten bei der Kommunikation. Darüber hinaus ist die drahtlose Kommunikation deutlich anfälliger gegenüber externen Interferenzen und anderen Umgebungseinflüßen. So wird das 2,4-GHz-Band auch von anderen Technologien, wie beispielsweise Bluetooth, genutzt. Dies kann zu einer nicht vorhersehbaren Störung der Kommunikation führen und eine zeitaufwendige Neuübertragung der Daten notwendig machen. Ähnliche Probleme ergeben sich auch im 5-GHz-Band. Hier werden vom Gesetzgeber sogar Ausweichmechanismen wie Dynamic Frequency Selection (DFS, dynamische Frequenzauswahl) und Transmission Power Control (TPC, Sendeleistungskontrolle) verpflichtend vorgeschrieben, um den Betrieb von Wetter- und Flugradaranlagen nicht zu stören. Die Störanfälligkeit von drahtlosen Kommunikationstechnologien und das damit einhergehende nicht-deterministische Zeitverhalten machen eine Verwendung von WLAN in Umgebungen mit harten Echtzeitanforderungen problematisch und eröffnen ein eigenständiges Forschungsfeld (vgl. RT-WiFi [69, 108] und LiT MAC [42]).

3.2 Stand der Forschung

3.2.1 Feldbussysteme und ihre Schwächen

Feldbussysteme sind die erste Generation von Technologien zur echtzeitfähigen Vernetzung verteilter Steuerungssysteme (engl. Distributed Control Systems, DCS), welche flächendeckend im Bereich der Industrieautomation eingesetzt wurden. Die Entwicklung von Feldbussystemen nahm besonders im Verlauf der 1980er-Jahre an Fahrt auf. Im Zuge der steigenden Anzahl von Sensoren, Aktoren und Steuerungseinheiten in industriellen Netzen erwies sich die bis dahin etablierte parallele und vollvermaschte Vernetzung der einzelnen Elemente als zu kosten- und planungsintensiv [111]. Bei Feldbussystemen werden hingegen alle Geräte über eine Bus-Interface-Karte mit einem zentralen Kommunikationsbus verbunden. Auf diese Weise werden der Planungsaufwand und die Kosten für die Verkabelung drastisch reduziert. Das Nachrichtenformat und das verwendete Zugriffsverfahren

hängen dabei von dem jeweils verwendeten Feldbus/Feldbusprotokoll ab. Im Verlauf der 80er Jahre wurde von verschiedenen Firmen der Automatisierungsbranche eine Vielzahl von Feldbusprotokollen entwickelt, welche häufig proprietär waren. Im Verlauf des von den Kunden getriebenen Selektionsprozesses verschwand ein Großteil dieser Protokolle jedoch wieder vom Markt. Seit dem Jahr 1999 werden Feldbussysteme von der International Electrotechnical Commission (IEC) in dem internationalen Standard IEC 61158 genormt [96]. Tabelle 6 gibt eine Übersicht der bis heute weit verbreiteten, echtzeitfähigen Feldbussysteme und ihrer Einsatzbereiche.

Feldbus	**Entwickler**	**Einführungs-jahr**	**Anwendungsbereich**
Profibus PA/DP	Siemens	PA: 1995 DP: 1994	Inter-PLC-Kommunikation
LONWorks	Echolon	1991	Gebäudeautomatisierung
CANopen	CAN in Automation	1995	Automotive, Sensor-Aktor-Ebene
Interbus-S	Phoenix Contact	1987	Sensor-Aktor-Ebene, Prozessüberwachung
Foundation Fieldbus H1	Fieldbus Foundation	1995	Sensor-Aktor-Ebene
ARINC 629	ARINC	1989	Luftfahrt

Tabelle 6: Übersicht verbreiteter Feldbussysteme und ihrer Anwendungsbereiche [13, 29, 34, 107].

Im Zuge der „Industrie 4.0“-Bewegung verlieren Feldbussysteme jedoch immer weiter an Bedeutung. Die steigende Anzahl von Netzwerkteilnehmern in Industrienetzen und die Integration von Geräten auf der Feldebene in übergeordnete Geschäftsprozesse (vertikale Integration) stellen neue Anforderungen an die Skalierbarkeit und die verfügbare Datenrate [1]. Diesen Anforderungen können Feldbussysteme nicht gerecht werden. PROFIBUS lässt beispielsweise lediglich eine maximale Anzahl von 124 Slave-Knoten pro Kommunkationsbus zu. Zwar lassen sich mehrere Einzel-Busse miteinander verbinden, jedoch steigt hier der Planungsaufwand ungemein. Darüber hinaus ermöglicht PROFIBUS nur Datenraten von maximal 31,25 kbit/s (PROFIBUS PA) bzw. 12,1 Mbit/s (PROFIBUS DP). Für den CAN Bus wird im ISO-Standard 11898 eine Datenrate von 1 Mbit/s angegeben, jedoch dürfen hierfür eine Leitungslänge von maximal 40 m und die Anzahl von 30 Geräten nicht überschritten werden. Ähnliche Kennwerte ergeben sich auch für den Foundation Fieldbus. Hier können pro Bussegment maximal 32 Netzwerkteilnehmer angebunden werden. Die Datenrate beträgt, wie auch bei PROFIBUS PA, 31,25 kbit/s. Darüber hinaus lassen sich Feldbusse nur über Gateways in andere Netzwerke einbinden. Dies führt zu einem Flaschenhals und einem SPoF bei der vertikalen Intergation.

3.2.2 Industrial Ethernet

Auf Grund der Schwächen von Feldbussystemen, werden diese nach und nach durch echtzeitfähige Netzwerke auf der Basis von Ethernet ersetzt. Diese Technologien werden auch als Industrial Ether-

net (IE) bezeichnet. Die Entwicklung solcher Netzwerke hat bereits in den späten 80er-Jahren begonnen. Auf Grund der langen Einsatzzyklen von Technologien im Industrieumfeld (20 Jahre und länger) und dem fehlenden Druck zum Einsatz neuer Technologien, erfolgte der tatsächliche Durchbruch von Industrial Ethernet erst nach der Jahrtausendwende. Die treibende Kraft für den Wandel war dabei die „Industrie 4.0"-Bewegung und die daraus entstehenden neuen Anforderungen an die Kommunikationsnetze. Die Verwendung von Ethernet-basierten Technologien ermöglicht die Nutzung von übergeordneten Protokollen wie IP, TCP und UDP auch auf der Feldebene. Dies führt nicht nur zu einer deutlich höheren Anzahl möglicher Netzwerkteilnehmer, sondern vereinfacht auch die Anbindung an andere IP-basierte Netzwerke und somit die vertikale Intregration. Im Laufe der Jahre entstand eine Vielzahl von Ethernet-basierten Netzwerktechnologien zur Echtzeitkommunikation. Danielis et al. geben in [23] eine Übersicht über die wichtigsten Industrial-Ethernet-Lösungen. Dabei nehmen sie zudem eine Einordnung der einzelnen Technologien in die jeweilige Echtzeitkategorie vor und gehen auf die jeweiligen Hard- und Softwareanforderungen ein. Tabelle 7 gibt diese Übersicht wieder.

IE-Lösung	**Klasse**	**Kategorie**	**Enthält einen SPoF?**	**Benötigt Spezialhardware?**
Modbus-TCP	1	A	Nein	Nein
Ethernet Powerlink	3	C	Ja	Optional
EtherCAT	3	C	Ja	Ja
TCNet	2	C	keine Angabe	Ja
TTEthernet	2	C	Ja	Ja
CC-Link IE Field	2	C	Ja	Optional
Profinet	3	C	Ja	Ja
EtherNet/IP	3	C	Nein	Ja
SERCOS III	3	C	Ja	Optional
DRTP	3	B	Ja	Nein
DARIEP	3	C	Ja	Ja
HaRTKad	2-3	A	Nein	Nein

Tabelle 7: Übersicht verbreiteter Industrial-Ethernet-Lösungen [23].

Der überwiegende Anteil der heutzutage verwendeten IE-Lösungen, wie beispielsweise PROFINET IO/IRT, EtherCAT oder SERCOS III, beruht dabei auf dem Master-Slave-Prinzip [20, 57, 83]. Hierbei wird der Netzwerkzugriff über eine zentrale Instanz, den Master, über einen TDMA-Ansatz so koordiniert, dass die Netzwerkgeräte (Slaves) sich nicht gegenseitig in ihrer Kommunikation stören. Auf diese Weise soll eine hart echtzeitfähige Kommunikation gewährleistet werden. Die Nutzung einer zentralen Instanz wirkt sich jedoch nachteilig auf die Robustheit des Netzwerkes aus (SPoF-Problem). Zudem setzen viele der etablierten IE-Lösungen die Verwendung von proprietärer und meist sehr kostenintensiver Spezialhardware oder propritärer Protokollanpassungen voraus. Dies reduziert den Spielraum bei der Planung und Bereitstellung der Netzwerke erheblich und erhöht zudem die Kosten.

Für die Nutzung von ProfiNet IO/IRT werden beispielsweise spezielle Switches benötigt, da für jedes Datenpaket eine feste Route durch das Netzwerk in Abhängigkeit des Sendezeitpunktes geplant wird. Bei EtherCAT werden spezielle Hardware-Controller auf der Seite der Slaves benötigt (EtherCAT Slave Controller, ESC), welche die On-the-Fly-Verarbeitung der EtherCAT-Frames ermöglichen [30]. EtherNet/IP kommt zwar ohne eine zentrale Instanz aus, benötigt jedoch Hardware-Unterstützung zur Zeitsynchronisation, um die höchste Echtzeitkategorie zu erreichen [79, 91]. Außerdem werden Broad- und Multicast-Funktionen in den Routern vorausgesetzt, deren Konfiguration bislang nicht standardisiert ist.

3.2.3 Aktuelle Forschung

In diesem Abschnitt werden aktuelle Forschungsarbeiten im Bereich der Echtzeitkommunikation für das IIoT beleuchtet.

In [93] stellen Schlesinger et al. mit *VABS* eine echtzeitfähige Ethernet-Lösung auf Master-Slave-Basis vor, bei der jeder Netzwerkteilnehmer die Rolle des Masters einnehmen kann. Um ein deterministisches Zeitverhalten der Kommunikation zu gewährleisten, ersetzen Schlesinger et al. die Sicherungsschicht durch ein selbst entwickeltes Protokoll. Dies hat jedoch zur Folge, dass angepasste Hardware benötigt wird. Hierdurch geht die Kompatibilität mit Standard-Ethernet verloren, sodass die vertikale Integration erschwert wird.
Hu et al. beschreiben ebenfalls einen Ansatz zur echtzeitfähigen Kommunikation auf Basis von Ethernet [50]. Wie auch viele bereits etablierte IE-Lösungen setzen Hu et al. dabei auf das Master-Slave-Prinzip. Allerdings stellt hier der Master, im Gegensatz zu der Lösung von Schlesinger et al., einen SPoF dar. Dies wirkt sich negativ auf die Robustheit des Netzwerkes aus. Zudem müssen die Slaves über spezielle Hardware verfügen, was zu höheren Anschaffungskosten führt und Probleme in der Interoperabilität nach sich zieht.
Santos et al. stellen ein Konzept Namens *Hard Real-Time Ethernet Switching Architecture (HaRTES)* vor. Hierbei wird eine hierarchische Anordnung der Switches genutzt, um eine dynamische Bandbreitenallokation für bestimmte Datenströme zu ermöglichen [92]. Santos et al. erstellen eine „Proof-of-Concept“-Umsetzung in einem Switch als Hardware-Software-Co-Design. Ashjaei et al. erweitern *HaRTES* um die Möglichkeit der Multi-Hop-Kommunikation über mehrere HaRTES-Switches hinweg [9]. Ashjaei et al. untersuchen zudem zwei unterschiedliche Ansätze bei der Paketweiterleitung zwischen HaRTES-Switches, das Distributed Global Scheduling (DGS) und das Reduced Buffering Scheme (RBS) [10]. Bei DGS werden Pakete zwischen den Switches gepuffert und nach einem von dem jeweiligen Switch festgelegten Zeitplan weitergeleitet. Mit RBS werden Pakete hingegen unverzüglich weitergeleitet. Ashjaei et al. kommen zu dem Schluss, dass RBS auf Grund der kürzeren Antwortzeiten gegenüber DGS zu bevorzugen ist. Unabhängig davon bleibt jedoch die Abhängigkeit von Spezialhardware bestehen.

Schmidt et al. präsentieren mit *DRTP* einen Ansatz zur echtzeitfähigen Ethernet-basierten Kommunikation ohne den Nachteil einer zentralen Instanz [95]. Dabei wird der Netzwerkzugriff über zwei proprietäre Protokollschichten gesteuert, welche direkt oberhalb von Ethernet eingefügt werden. Als Medienzugriffsverfahren wird ein TDMA-Ansatz mit festgelegten Zeitschlitzen verwendet. Die fehlende Unterstützung von TCP/UDP und IP verringert allerdings Interoperabilität und verhindert die nahtlose Integration in bereits existierende Netzwerke. Dies wirkt sich negativ auf die vertikale Integration aus, was eines der Hauptziele der Industrie 4.0-Bewegung und des IIoT ist.
Qian et al. präsentieren mit *XpressEth* einen Ansatz, der die Koexistenz von zeitkritischer und nichtzeitkritischer Kommunikation in Ethernet-basierten Netzwerken ermöglichen soll [86]. Ihr Ansatz beruht auf der Einführung von sogenannter Frame-Preemption sowohl an den Eingangs- als auch den Ausgangspuffern der Switches (Dual Preemption). Bei der Frame-Preemption wird die Übertragung von nicht-zeitkritischen Frames zugunsten zeitkritischer Frames unterbrochen. Auf diese Weise soll die Ende-zu-Ende-Latenz für Anwendungen mit Echtzeitanforderungen minimiert werden. Mit Hilfe eines Schedulers werden alle zeitkritischen Kommunikationsflüsse im Netzwerk so geplant, dass keine Ressourcenkonflikte entstehen. Das Scheduling muss jedoch zentralisiert erfolgen, was einen SPoF nach sich zieht. Zudem erfordert die Dual Preemption Veränderungen an den Switches, sodass keine Standardkomponenten verwendet werden können.
Grüner et al. untersuchen die Anwendbarkeit und den Nutzen von REST-Architekturen im IIoT [46]. Hierzu präsentieren sie eine modifizierte Version von OPC-UA (Open Plattform Communications - Unified Archirecture). Für die Untersuchung des Zeitverhaltens werden drei verschiedene Plattformen mit unterschiedlicher Leistungsfähigkeit herangezogen. Die einzelnen Geräte wurden über Standard Fast Ethernet miteinander verbunden. Die Messungen zeigen ein nicht-deterministisches Zeitverhalten der Kommunikation. Zur Lösung des Problems verweisen die Autoren auf bereits bestehende echtzeitfähige Lösungsansätze auf den unteren Netzwerkschichten.
In der IEEE 802.1 Time-Sensitive Networking (TSN) Task Force wird derzeit an einem Ethernet-Standard für den Einsatz in Umgebungen mit Echtzeitanforderungen gearbeitet [77, 111]. Dabei baut TSN auf dem Standard IEEE 802.1 Audio-/Video-Bridging (AVB) auf, der obere Zeitschranken für die Übertragung von Audio- und Video-Streams garantiert. Unter dem Begriff TSN lassen sich insgesamt 60 IEEE Standards zusammenfassen, die neben der zeitlichen Ebene der Kommunikation auch Sicherheitsaspekte betrachten [111]. TSN stellt jedoch keinen Gegensatz zu der in dieser Arbeit vorgestellten Lösung zur Echtzeitkommunikation dar. Vielmehr lässt TSN sich in Verbindung mit Software Defined Networking (SDN) als sinnvolle Ergänzung verstehen, die eine deterministische Planung der Kommunikation im Verbindungsnetz ermöglicht, während die in dieser Arbeit vorgestellte CoAP-Erweiterung eine interoperable und zeitlich-deterministische Kommunikation auf Anwendungsebene erlaubt.
Skodzik et al. präsentieren mit *Hard Real-Time Kademlia (HaRTKad)* einen anderen rein Software-basierten Ansatz zur echtzeitfähigen Kommunikation über Standard-Ethernet [104]. Die Grundlage hierfür bildet das aus dem Bereich des File-Sharing bekannte Peer-to-Peer-(P2P)-Protokoll Ka-

demlia. HaRTKad stellt aufgrund seiner dezentralen Arbeitsweise und den fehlenden Hardware-Abhängigkeiten einen sehr vielversprechenden Ansatz für die Echtzeitkommunikation dar und wird im nachfolgenden Abschnitt näher erläutert.

3.3 Das HaRTKad-Framework

Das HaRTKad-Framework beschreibt eine hart-echtzeitfähige Modifikation von Kad, einer Implementierungsvariante des P2P-Protokolls Kademlia, welche in dem weit verbreiteten File-Sharing-Netzwerk eMule verwendet wird. HaRTKad setzt zur Erreichung der Echtzeitfähigkeit auf ein rein Software-basiertes TDMA-Verfahren, sodass es auch mit Standard-Ethernet verwendet werden kann. Hierbei nutzt HaRTKad die Eigenschaften und Funktionen von Kad, um eine einheitliche Zeitbasis im Netzwerk zu schaffen und Zeitschlitze an die einzelnen Geräte zu verteilen. Durch das TDMA-Verfahren soll eine kollisionsfreie Kommunikation erreicht und eine Überlastsituation (Congestion) auf den Infrastruktur-Geräten, wie z.B. Switches, verhindert werden. Dies führt zu einem zeitlich deterministischen Kommunikationsverhalten. Im weiteren Verlauf dieses Kapitels werden zunächst Kademlia und die in Kad umgesetzten Erweiterungen erläutert. Danach wird das von HaRTKad genutzte TDMA-Verfahren und die Zeitsynchronisation näher beschrieben. Abschließend werden Limitierungen von HaRTKad aufgezeigt.

3.3.1 Das Kademlia-Protokoll & Kad

Bei Kademlia handelt es sich um ein vollständig dezentrales P2P-Protokoll, dass sich besonders gut für die Suche von Daten oder Netzwerkteilnehmern eignet [73]. Hierzu wird jedem Netzwerk-Knoten und jedem Datenobjekt eine eindeutige ID, ein sogenannter Hash-Wert, zugewiesen. Dieser Hash-Wert berechnet sich mit Hilfe einer Hash-Funktion wie Message Digest 5 (MD5) beispielsweise aus dem Namen einer Datei oder der MAC-Adresse eines Netzwerk-Knotens. In Kademlia wird der Hash-Wert zur logischen Adressierung von Knoten und Daten verwendet. Dabei teilen sich die Netzwerkteilnehmer und Daten denselben Adressraum, welcher sich wie in Abbildung 28 als Strahl darstellen lässt.

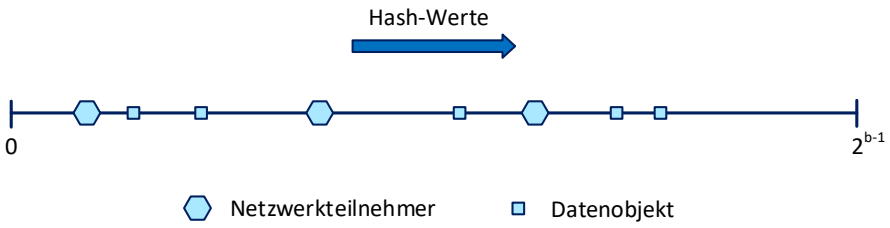

Abbildung 28: Darstellung des gemeinsamen b-Bit Adressraumes für Netzwerkteilnehmer und Daten in Kademlia als Strahl.

Um eine dezentrale Suche im Netzwerk zu ermöglichen, verwaltet jeder Knoten eine eigene Routing-Tabelle. Die Routing-Tabelle wird hierbei in mehrere, auch als k-Buckets (engl. für Eimer) bezeichnete Bereiche unterteilt. Das k steht dabei für die maximale Anzahl von Kontakten, die sich in einem Bucket befinden können. Welchem Bucket ein Kontakt zugeordnet wird, bestimmt sich aus seinem Hash-Wert und dem Hash-Wert des Knotens, der die Routing-Tabelle verwaltet. Hierbei wird mit Hilfe der XOR-Funktion die logische Distanz zwischen den Hash-Werten der beiden Knoten ermittelt. Die Auswahl des Buckets erfolgt nun mittels eines binären Baums. Dabei wird die Distanz vom höchstwertigen Bit hin zum Niederwertigsten untersucht. Ist der Wert der jeweiligen Stelle 1, wird der Kontakt dem Bucket hinter dem rechten Ast zugeordnet. Ist der Wert hingegen 0, wandert der Kontakt weiter entlang des linken Astes und die nächste Bitstelle wird betrachtet. Dieses Vorgehen wird solange wiederholt, bis der Knoten einem Bucket zugeordnet ist. Man spricht hier auch von Prefix Matching. Abbildung 29 stellt die Zuordnung von Kontakten zu den Buckets für die ersten vier Bitstellen der Distanz graphisch dar. Aus der Anordnung der Buckets und der maximalen Kontaktanzahl von k pro Bucket ergibt sich, dass ein Knoten viele Kontakte in kurzer Distanz und nur wenige weit entfernte Kontakte in seiner Routing-Tabelle speichert.

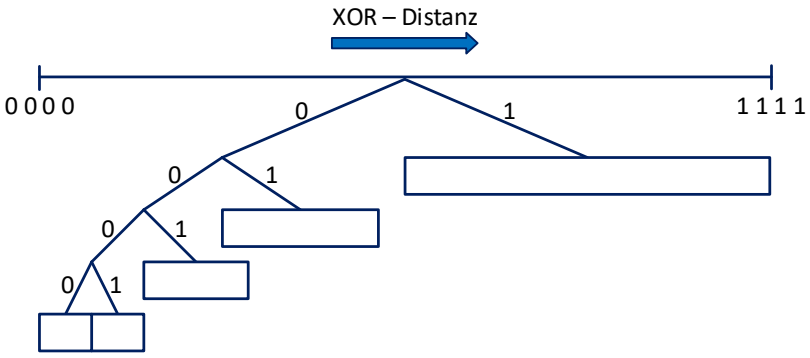

Abbildung 29: Aufbau der Routingtabelle in Kademlia. Die Boxen symbolisieren jeweils die k-Buckets, in denen die Kontaktadressen der Knoten gespeichert werden [73].

Möchte ein Knoten nun ein Datenobjekt oder einen anderen Knoten suchen, sendet er eine Anfrage mit dem Hash-Wert des gesuchten Objekts an die k Knoten aus seiner Routing-Tabelle mit der kürzesten Distanz zu diesem Hash-Wert. Diese antworten entweder mit den gewünschten Daten, wenn sie diese gespeichert haben, oder mit einer Liste von k Knoten, die sich näher an dem gewünschten Ziel befinden. Im nächsten Schritt sendet der suchende Knoten eine Anfrage an die noch nicht kontaktierten der nun bekannten k nächstgelegenen Knoten. Dies wird solange wiederholt, bis der gewünschte Knoten oder das Datenobjekt gefunden, eine maximale Anzahl von Suchschritten durchgeführt oder ein Timout erreicht wurde. Auf Grund dieses Vorgehens und der besonderen Struktur der Routing-Tabelle, werden Suchen in einem Kademlia-Netzwerk im Durchschnitt mit einer zeitlichen Komplexität von $O(n) = log_2\, n$ durchgeführt. In [105] werden weitere Optimierungsmöglichkeiten für Suchoperationen in Kademlia präsentiert. Hierzu zählt beispielsweise das parallele Versenden von

Suchanfragen an mehrere Knoten. Neben Suchoperationen für Daten und andere Netzwerkknoten stehen auch noch weitere Operationen zur Verfügung, die für diese Arbeit jedoch nicht von Bedeutung sind. In [105] werden noch weitere Optimierungsmöglichkeiten für Suchoperationen in Kademlia präsentiert. Hierzu zählt beispielsweise das parallele Versenden von Suchanfragen an mehrere Knoten.

Die aus dem File-Sharing-Protokoll eMule bekannte Umsetzung von Kademlia, Kad, nimmt drei wesentliche Änderungen an Kademlia vor. Zunächst führt sie neue Nachrichten-Typen ein, um auch Metadaten für Dateien austauschen zu können. Darüber hinaus wird die Struktur der Routing-Tabelle leicht abgewandelt, sodass die Suchkomplexität weiter optimiert wird. Die weitaus wichtigste Änderung von Kad gegenüber Kademlia ist die Einführung einer Suchtoleranz. Die Suchtoleranz (ST) legt fest, welche Knoten für die Speicherung bestimmter Inhalte verantwortlich sind. Im Vergleich dazu sind beim herkömmlichen Kademlia immer die k nächstgelegenen Knoten für ein Datum zuständig. Ist die Distanz zwischen dem Hash-Wert eines Knotens und dem eines zu speichernden Datenobjekts kleiner als die Suchtoleranz, so muss der Knoten das Datenobjekt speichern (vgl. Speicherbedingung in Formel (1)).

$$0 \leq Hash_{Daten} \oplus Hash_{Knoten} < ST \tag{1}$$

Mit der ST soll das Maß der redundanten Datenhaltung im Netzwerk von der Kademlia-Kenngröße k getrennt werden. Dies hat den Vorteil, dass das angestrebte Ausmaß der Datenredundanz keinen Einfluss auf die Struktur der Routing-Tabelle hat. Entsprechend ändert sich auch der Ablauf einer Suchoperation nach Daten in Kad. Wie auch bei Kademlia werden zunächst Suchanfragen an die k Knoten aus der Routing-Tabelle versendet, welche die kleinste Distanz zu dem Hash-Wert der gesuchten Daten aufweisen. Die angefragten Knoten antworten entweder mit den gesuchten Daten oder mit k Knoten, deren Distanz zu dem Hash-Wert der gesuchten Daten noch geringer ist. Anschließend kann der Initiator der Suche eine Suchanfrage an diese Knoten senden. Dies wird nun solange wiederholt, bis ein Knoten gefunden wurde, dessen Distanz zu den gesuchten Daten kleiner ist als die Suchtoleranz oder bis eine vorher festgelegte Anzahl an Suchschritten ausgeführt wurde beziehungsweise ein Timeout eintritt. Nachfolgend wird erläutert, wie die Suchtoleranz genutzt werden kann, um einen TDMA-basierten Netzwerkzugriff der einzelnen Netzwerkteilnehmer zu realisieren.

3.3.2 TDMA und Zeitsynchronisation in HaRTKad

Wie eingangs erläutert, setzt HaRTKad auf ein rein Software-basiertes TDMA-Verfahren, um ein zeitlich deterministisches Kommunikationsverhalten herbeizuführen. Im Gegensatz zu anderen Industrial-Ethernet-Ansätzen verzichtet HaRTKad jedoch auf eine zentrale Instanz zur Koordination des Netzwerkzugriffs. Vielmehr nutzt es die Eigenschaften und Funktionen von Kad, um jedem Netz-

werkteilnehmer die eigenständige Berechnung seiner Zugriffszeit zu ermöglichen. Dabei dient der Hash-Wert des Knotens als Basis zur Bestimmung des Zeitschlitzes. Diese Korrelation ist möglich, da sowohl der Hash-Wert als auch der Zeitschlitz im Netzwerk einzigartig sind. Um die Berechnung des Zeitschlitzes aus dem Hash-Wert zu ermöglichen, nutzt HaRTKad die Suchtoleranz ST. Ursprünglich verwendet Kad einen statischen Wert für ST, was jedoch für diesen Anwendungsfall nicht hinreichend ist. Daher nutzt HaRTKad eine modifiziert Form der in [22] von Danielis et al. beschriebenen dynamischen Suchtoleranz-Berechnung (DST - Dynamic Search Tolerance). Der DST-Algorithmus hat das Ziel, die Speicherung aller möglichen Daten zu gewährleisten. Hierzu wird ST so bestimmt, dass sich in jedem durch ST definierten Verantwortungsbereich mindestens ein Knoten befindet. Skodzik et al. wandeln den DST-Algorithmus so ab, dass sich in jedem Verantwortungsbereich **maximal ein** Knoten befindet. Diese Modifkation wird von Skodzik et al. als inversdynamische Suchtoleranz-Berechnung (IDST) bezeichnet [104]. Abbildung 30 stellt die sich aus beiden Algorithmen ergebenden Suchtoleranzen für ein Netzwerk mit drei Knoten und einer Hash-Wertbreite von 4 Bit gegenüber.

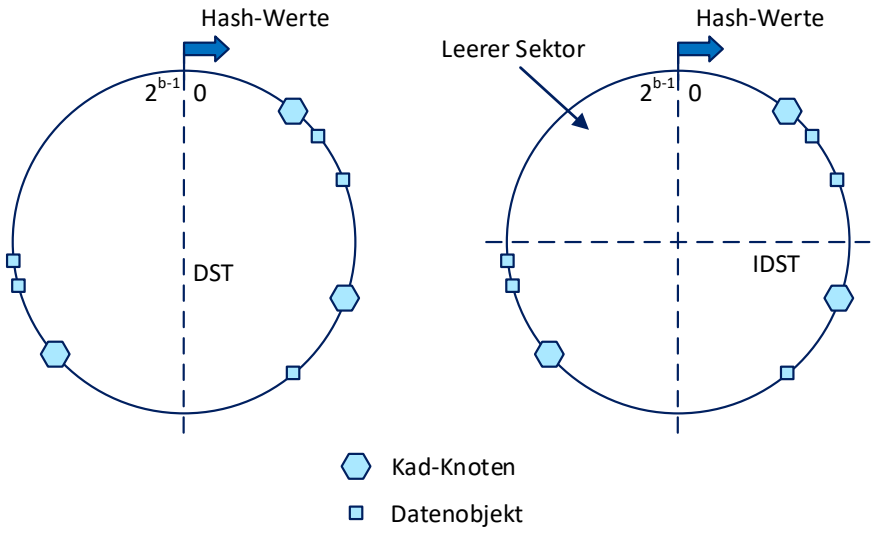

Abbildung 30: Gegenüberstellung der mit DST (links) und IDST (rechts) berechneten Suchtoleranz [22, 104].

Für die Berechnung der Suchtoleranz müssen allerdings die Anzahl der Netzwerkteilnehmer und deren Hash-Werte bekannt sein. Um diese zu bestimmen, wird zunächst ein initialer Master-Knoten unter allen Netzwerkteilnehmern ausgewählt. Dieser kann entweder durch den Anwender initial festgelegt oder durch eine Wettlaufstrategie bestimmt werden. Bei Letzterer wird derjenige Knoten zum Master, der sich zuerst gegenüber den anderen Netzwerkteilnehmern als Master zu erkennen gibt. Die Funktion des Masters kann von jedem beliebigen Knoten übernommen werden. Der Master führt dann im ersten Schritt einen Discovery-Vorgang durch, um die Anzahl der Netzwerkteilnehmer und deren Hash-Werte zu bestimmen. Hierzu greift HaRTKad auf den in [100] von Skodzik et al. beschriebenen Algorithmus namens KaDisSy (Kademlia Discovery and Synchronization) zurück. Der

KaDisSy-Algorithmus basiert auf der Annahme, dass die Hash-Werte von Knoten aus einem bekannten Muster heraus generiert werden, wie beispielsweise einer festgelegten Zeichenkette (z.B. „Node“) mit nachfolgender aufsteigender Nummerierung. Zum Finden der Netzwerkknoten geht der Master dabei wie in Abbildung 31 dargestellt vor.

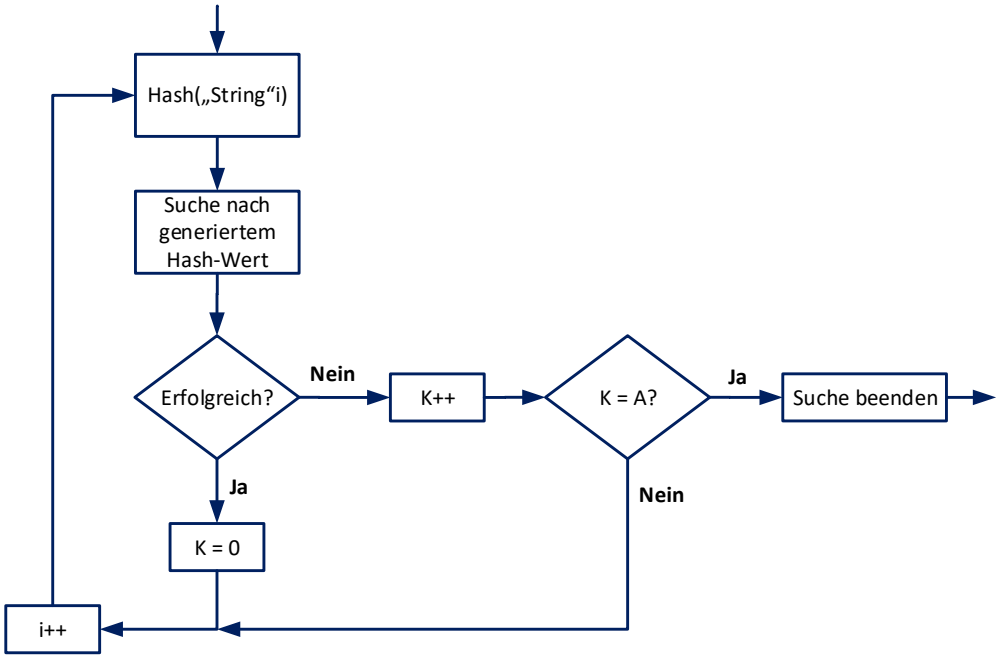

Abbildung 31: Ablaufdiagramm des KaDis-Algorithmus [102].

Im ersten Schritt wird ein dem Muster entsprechender Hash-Wert generiert. Anschließend wird eine Suche nach einem Knoten mit diesem Hash-Wert gestartet. Fällt die Suche positiv aus, so wird K, die Anzahl der konsekutiv fehlgeschlagenen Suchoperationen, auf null gesetzt. Andernfalls wird K um eins inkrementiert. Danach wird der nächste Hash-Wert generiert und im Netzwerk gesucht. Hat K einen bestimmten Schwellwert A erreicht, wird der Algorithmus beendet und der Discovery-Prozess als abgeschlossen angenommen. Ob dabei tatsächlich alle Knoten gefunden werden, hängt maßgeblich von der Wahl des Schwellwertes A ab. A gibt die Toleranz des Algorithmus gegenüber nicht besetzten Stellen im Adressraum an. Nachdem der Master die Anzahl der Knoten und deren Hash-Werte bestimmt hat, kann er den IDST-Algorithmus ausführen. Die so bestimmte Suchtoleranz wird dann durch den Master an die anderen Netzwerkteilnehmer verteilt. Anschließend kann jeder Knoten wie in Formel (2) beschrieben selbstständig seinen Zeitschlitz berechnen.

$$Slot_{Node} = \frac{Hash_{Node}}{ST_{IDST}} \tag{2}$$

Die Gesamtanzahl der Zeitschlitze N_{Slots} hängt dabei direkt von der Suchtoleranz und der Bitbreite der Hash-Werte ab und berechnet sich wie in Formel (3).

$$N_{Slots} = \frac{2^b}{ST_{IDST}} \tag{3}$$

Auf Grundlage von N_{Slots} und der Länge eines Zeitschlitzes t_{Slot} lässt sich die Zyklus-Zeit T_{Cycle} wie in Formel (4) bestimmen.

$$T_{Cycle} = t_{Slot} * N_{Slots} \tag{4}$$

Mit Hilfe der Zyklus-Zeit kann dann auch die relative Zeit innerhalb eines Zyklus t_{Ring} mittels Formel (5) berechnet werden. Hierbei bezeichnet t_{now} die aktuelle Systemzeit.

$$t_{Ring} = t_{now} \mod T_{cycle} \tag{5}$$

Abschließend kann jeder Knoten selbst über die in Formel (6) dargestellte Bedingung prüfen, ob er auf das Kommunikationsmedium zugreifen darf. Dabei muss t_{Ring} deutlich kleiner sein als das Ende des Zeitschlitzes, sodass ein begonnener Kommunikationsvorgang noch im eigenen Zeitschlitz abgeschlossen werden kann.

$$(t_{Ring} > Slot_{Node} * t_{Slot}) \wedge (t_{Ring} << Slot_{Node} * t_{Slot} + t_{Slot}) \tag{6}$$

Damit auf diese Weise ein kollisionsfreier Netzwerkzugriff erreicht werden kann, müssen alle Knoten im Netzwerk über eine einheitliche Zeitbasis verfügen. Dies wird in HaRTKad über das in [103] beschriebene verteilte Synchronisationsverfahren erreicht. Hierbei dient der Masterknoten als Referenz-Zeitgeber. Die Synchronisation der Systemzeiten mit dem Master erfolgt baumartig über Hilfsknoten. Dabei lässt sich über die Anzahl der neuen Hilfsknoten je Baum-Ebene die Dauer, die ein Synchronisationsvorgang benötigt, bestimmen. Die Zeitsynchronisation in HaRTKad wird periodisch in einem Wartungszyklus ausgeführt. Der Häufigkeit der Re-Synchronisation kann dabei in Abhängigkeit der Clock Drift zwischen den einzelnen Netzwerkteilnehmern gewählt werden.

3.3.3 Limitierungen von HaRTKad

HaRTKad ist einer der wenigen rein Software-basierten Ansätze, mit dem sich harte Echtzeitanforderungen erfüllen lassen. Dabei ist es vollkommen dezentralisiert und vermeidet einen SPoF. Darüber

hinaus erlaubt es die Nutzung von Standard-Technologien wie Ethernet und IP auf den unteren Schichten und ermöglicht so eine einfachere horizontale sowie vertikale Integration.

Im Rahmen dieser Arbeit wurde HaRTKad auf mögliche Schwächen untersucht. Dabei konnten drei wesentliche Einschränkungen identifiziert werden, zu denen auch potentielle Lösungen erarbeitet wurden. Jedoch wurden diese bisher nur theoretisch betrachtet, sodass keine abschließende Aussage über deren Wirksamkeit getroffen werden kann. Die Ergebnisse dieser Untersuchungen wurden bereits in [62] veröffentlicht.

Die erste Einschränkung ist die potenziell sehr niedrige Auslastung des Kommunikationsmediums. Dieses Problem tritt auf, wenn Gruppen von mehreren Knoten mit einem sehr niedrigen logischen Abstand der Hash-Werte existieren. In diesem Fall ergibt sich aus dem IDST-Algorithmus ein sehr niedriger Wert für ST. Hieraus resultiert, wie in Abbildung 32 dargestellt, eine sehr große Anzahl von Zeitschlitzen, von denen nur ein geringer Teil tatsächlich genutzt wird. Das Kommunikationsmedium bleibt in der Folge über längere Zeit ungenutzt. Die hohe Anzahl von Zeitschlitzen wirkt sich zudem nicht nur auf die Medienauslastung aus, vielmehr führt sie auch zu einer entsprechend langen Zykluszeit, sodass Echtzeitanforderungen nur sehr begrenzt entsprochen werden kann. Dem könnte mit der Anwendung von Verteilungsalgorithmen entgegengewirkt werden, die eine gleichmäßige Verteilung der Knoten im logischen Adressraum herbeiführen. [36] fasst einige infrage kommende Algorithmen zusammen.

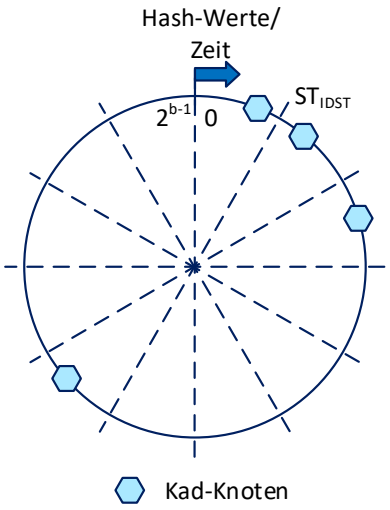

Abbildung 32: Missverhältniss zwischen verfügbaren Zeitschlitzen und Netzwerkteilnehmern [62].

Ein weiteres Problem, welches in HaRTKad bisher weitgehend unberücksichtigt bleibt, ist das Auftreten von Hash-Kollisionen. Als Hash-Kollision bezeichnet man das Auftreten von zwei Netzwerkteilnehmern, die für sich den gleichen Hash-Wert als logische Adresse berechnet haben. Die Konsequenz einer Hash-Kollision ist, dass die beiden betroffenen Knoten im Netzwerk nicht eindeutig adressierbar sind. Darüber hinaus sind die beiden Knoten nicht in der Lage, sich gegenseitig zu finden. In HaRT-

Kad wirken sich Hash-Kollisionen allerdings noch drastischer aus, da die Hash-Werte der Knoten die Grundlage für die Berechnung des eigenen Zeitschlitzes bilden. Infolge einer Hash-Kollision nutzen dabei zwei Netzwerkteilnehmer denselben Zeitschlitz und korrumpieren so das Echtzeitverhalten der Kommunikation. Hash-Kollisionen können vermieden werden, indem die einzelnen Knoten vor ihrem Beitritt in das Netzwerk nach dem eigenen Hash-Wert suchen. Erhalten sie eine positive Antwort, ist der Hash-Wert bereits belegt, sodass der beitretende Knoten einen neuen Hash-Wert generieren muss.

Die dritte Einschränkung ist die fehlende Möglichkeit, bestimmte Kommunikationsflüsse im Netzwerk zu priorisieren. Typischerweise setzen sich Automatisierungsumgebungen aus vielen heterogenen Geräten mit verschiedenen Aufgaben zusammen. Dabei stellen die einzelnen Geräte entsprechend ihrer Aufgaben auch unterschiedliche zeitliche Anforderungen an die Kommunikation. In HaRTKad finden diese Unterschiede bisher keine Berücksichtigung. Hier wird jedem Knoten die gleiche Zugriffszeit auf das Medium gewährt, unabhängig von seinen tatsächlichen Anforderungen. Dabei dienen immer die jeweils höchsten Anforderungen als Grundlage. Dies kann eine potenziell niedrige Medienauslastung mit sich bringen, da nicht jedes Gerät seinen Zeitschlitz in jedem Zyklus zur Kommunikation nutzt. Außerdem wird so die maximale Anzahl der Netzwerkteilnehmer eingeschränkt, da zwar noch Kommunikationskapazitäten im Netzwerk vorhanden, aber alle Zeitschlitze bereits belegt sind. Zur Lösung dieses Problems wird in [62] eine Einteilung der Netzwerkteilnehmer in die von Felser et al. und Fettweis et al. definierten Echtzeitklassen vorgeschlagen. Anschließend kann der logische Adressraum in drei Bereiche eingeteilt werden, die jeweils exklusiv für Geräte einer Echtzeitklasse reserviert sind. Die Einteilung der Zeitschlitze kann dann so erfolgen, dass sich Knoten mit niedrigeren Echtzeitanforderungen einen Zeitschlitz teilen, den sie dann in abwechselnden Zyklen nutzen (vgl. Abbildung 33).

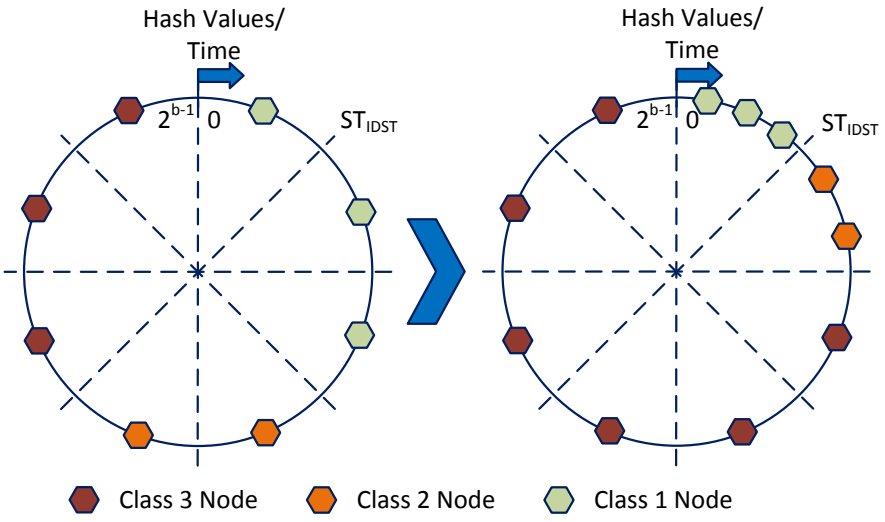

Abbildung 33: Clustering der Knoten mit niedrigen Echtzeitanforderungen und modifizierte Berechnung der Suchtoleranz [62].

Für eine praktische Umsetzung muss zunächst der KaDisSy-Algorithmus so angepasst werden, dass Informationen über die Echtzeitklasse der jeweiligen Geräte gesammelt werden. Weiterhin muss ein Mechanismus zur Verteilung der für die verschiedenen Echtzeitklassen reservierten Adressbereiche eingeführt werden, damit jeder Knoten seinen Hash-Wert entsprechend anpassen kann. Darüber hinaus muss die Berechnung von ST_{IDST} so angepasst werden, dass nur Knoten mit den höchsten Echtzeitanforderungen berücksichtigt werden.

Neben diesen in [62] betrachteten Problemen, unterliegt HaRTKad weiteren Einschränkungen, die einer Nutzung im IIoT entgegenstehen. Zum einen existiert für HaRTKad bisher nur eine plattformabhängige Umsetzung unter Verwendung des Echtzeitbetriebssystems FreeRTOS. Dies wirkt sich negativ auf die Verbreitung und Akzeptanz von HaRTKad aus, da eine Portierung auf unterschiedliche Geräte sehr aufwändig ist. Zum anderen kann HaRTKad allein den Anforderungen des IIoT nicht gerecht werden, da es selbst keine Schnittstellen zur interoperablen Kommunikation und Interaktion der Geräte bereitstellt, sondern nur den Nachrichtenaustausch selbst steuert. Daher muss HaRTKad immer in Kombination mit einem anderen IoT-Protokoll verwendet werden, welches über die nötigen Schnittstellen und Mechanismen, wie zum Beispiel Eventing, verfügt. In [101] wurde von Skodzik et al. bereits eine Kombination von HaRTKad und CoAP namens CoHaRT untersucht. Hierbei wurde jedoch nur der blockweise Transfer von CoAP umgesetzt, mit dem Ziel auch größere Datenmengen mit HaRTKad übertragen zu können. In dem Versuchsaufbau musste jeder Netzwerkteilnehmer gleichzeitig als CoAP-Client und CoAP-Server agieren. Dies ist in der Praxis jedoch nicht wünschenswert. Da CoAP speziell für ressourcenbeschränkte Umgebungen entwickelt wurde, sollte eine zusätzliche Belastung aller Geräte vermieden werden. Weiterhin werden die Antworten des CoAP-Servers in CoHaRT immer sofort versendet, sodass die Übertragung der Antworten auch in dem Zeitschlitz eines dritten, nicht an der Transaktion beteiligten Gerätes stattfinden kann. Skodzik et al. stellen jedoch selbst fest, dass sich dies negativ auf das Echtzeitverhalten der Kommunikation auswirken kann. Darüber hinaus wurde von Skodzik et al. in ihren Versuchen die für die Suche eines Kommunikationspartners benötigte Zeit nicht berücksichtigt. Zudem wurde nicht beschrieben, wie in dieser Protokollkombination wichtige CoAP-Mechanismen, wie die Geräte-Discovery oder CoAP Observe, umgesetzt werden können.

Am Beispiel von CoHaRT wird deutlich, dass eine kombinierte Nutzung von HaRTKad und etablierten IoT Protokollen nicht unproblematisch ist. In dieser Arbeit wird daher eine Lösung erarbeitet, die ausschließlich auf CoAP-Standard-Mechanismen zurückgreift. Um außerdem der Geräteheterogenität im IIoT gerecht zu werden, wird diese Lösung als plattformunabhängiger Prototyp umgesetzt. Auf diese Weise entsteht eine rein Software-basierte Lösung zur Echtzeitkommunikation, die den Anforderungen des IIoT in Hinblick auf Interoperabilität und Integrierbarkeit gerecht wird.

4 Das jCoAP-Framework

Bei dem jCoAP-Framework handelt es sich um eine leichtgewichtige Implementierung von CoAP, die für den Einsatz im IoT- und Automatisierungsumfeld konzipiert wurde. Die Ursprünge von jCoAP gehen auf die Arbeiten von Lerche et al. zurück [70]. Diese legten den Fokus ihrer Arbeit zunächst auf die standardkonforme Implementierung. Die Entwicklung von jCoAP kam jedoch auf dem Stand der Internet Draft Version 11 von CoAP[1] zum Erliegen. Im Rahmen dieser Arbeit wurde jCoAP zunächst an die finale Version des CoAP-Standards angepasst. Zudem wurden weitere Substandards, wie z.B. Blockwise Transfer, eingearbeitet. In diesem Kapitel wird das Konzept hinter jCoAp erörtert. Weiterhin wird beschrieben, wie die Standardkonformität von jCoAP im Rahmen dieser Arbeit nachgewiesen wurde. Abschließend wird jCoAP auf sein Zeitverhalten hin untersucht. Die in diesem Kapitel vorgestellten Ergebnisse wurden in Teilen bereits veröffentlicht [61].

4.1 Architektur des Frameworks

Um der in IoT-Szenarien üblichen Geräteheterogenität gerecht zu werden, basiert jCoAP auf der plattformunabhängigen Java-Technologie. jCoAP kann sowohl zur Umsetzung eines CoAP-Clients als auch eines CoAP-Servers genutzt werden. Abbildung 34 gibt eine Übersicht über die Architektur von jCoAP.

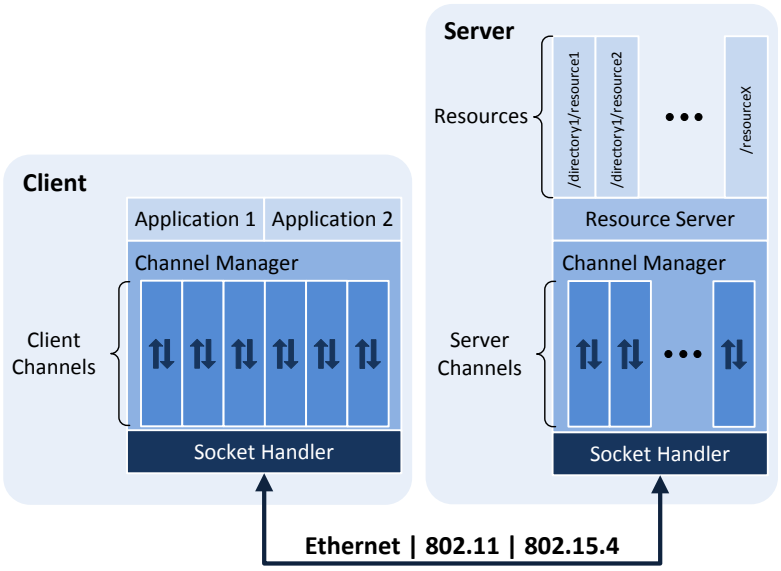

Abbildung 34: Aufbau eines Client-Server-Systems mit jCoAP [61].

[1] Der letzte Internet Draft vor der finalen Standardisierung war die Version 18

Auf einem jCoAP-Client können, wie in Abbildung 34 dargestellt, ein oder mehrere Client-Anwendungen laufen. Dabei kann jede Anwendung eine Verbindung zu einem oder mehreren entfernten Servern aufbauen. Um die jeweiligen Datenströme den entsprechenden Client-Anwendungen zuzuordnen, wird für jede Verbindung ein ClientChannel erstellt. Jeder dieser Kanäle wird mit Hilfe eines ChannelKeys (CK) eindeutig identifiziert. Der ChannelKey berechnet sich dabei wie in Formel (7) beschrieben.

$$CK = P * (P + Hash_{ServerIP}) + Port_{Server} \tag{7}$$

Hier steht P für einen beliebigen und zufällig gewählten Faktor. P verhindert die Doppelung des ChannelKeys, wenn mehrere Client-Anwendungen mit dem gleichen Server interagieren. $Hash_{Server}$ bezeichnet den Hash-Wert der IP-Adresse des Servers. Dieser Hash-Wert kann mit Hilfe einer beliebigen Hash-Funktion berechnet werden. In jCoAP kommt hier standardmäßig die in die InetAdress-Klasse integrierte Hash-Funktion zum Einsatz. Sendet eine Client-Anwendung eine Nachricht an einen Server, wird diese über den Channel an den SocketHandler weitergeleitet. Der SocketHandler verwaltet alle Ports und den eingehenden beziehungsweise ausgehenden Datenverkehr. Alle ausgehenden Nachrichten werden in einem Sendepuffer zwischengespeichert und in der Reihenfolge ihres Eintreffens im SocketHandler über den festgelegten Port versandt. Empfängt der Server die Nachricht an seinem SocketHandler, wird zunächst geprüft, ob bereits ein Channel zu dem entsprechenden Client existiert. Der Abgleich erfolgt dabei anhand des ChannelKeys, der sich aus der IP-Adresse des Senders und dessen Port berechnen lässt. Kann der Nachricht kein Channel zugeordnet werden, so wird ein neuer ServerChannel erstellt. Über den Channel wird die Nachricht an den ResourceServer weitergeleitet. Der ResourceServer stellt die Hauptkomponente eines jCoAP-Servers dar. Hier wird die Nachricht in Hinblick auf die angefragte Operation (GET, POST PUT, DELETE) untersucht und weitere Header-Optionen ausgelesen. Anschließend wird die in der Nachricht spezifizierte Ressource ausgewählt und überprüft, ob die gewünschte Operation auf diese angewendet werden kann. Ist dies der Fall, wird die jeweilige Operation auf der Ressource ausgeführt, je nach Operation und Ergebnis die entsprechende Antwort-Nachricht generiert und über den ServerChannel und SocketHandler an den Client übermittelt. Der Client empfängt diese Anwort wiederum an seinem SocketHandler. Von dort wird sie über den zugehörigen ClientChannel an die korrespondierende CoAP-Anwendung weitergeleitet.

jCoAP stellt dabei nicht nur die in der Basisspezifikation von CoAP beschriebenen Funktionalitäten bereit, sondern unterstützt ebenso den blockweisen Transfer großer Datenmengen sowie den in CoAP Observe spezifizierten Publish/Subscribe-Mechanismus.

jCoAP Observe

Der ResourceServer in jCoAP verwaltet eine Liste aller durch den Server bereitgestellten Ressourcen. Jede Ressource verfügt dabei über eine eigene Sammlung von ServerChannels. Sie entsprechen den jeweiligen Clients, welche die Ressource beobachten und über Änderungen informiert werden wollen. Empfängt ein Server einen GET-Request, welcher die Observe-Option enthält, fügt er den Channel zur Beobachter-Liste der jeweiligen Ressource hinzu. Tritt eine Änderung an der Ressource auf, wird für jeden Channel in der Liste eine entsprechende Notification generiert und an den hinter dem Channel stehenden Client versandt. Ein Client kann sein Abonnement auf zwei Arten kündigen. Zum einen kann er einen weiteren GET-Request mit einer Observe-Option für dieselbe Ressource an den Server senden. Zum anderen kann er auf eine Notification mit einer RST-Nachricht antworten. In beiden Fällen wird der Channel aus der Liste der Beobachter der jeweiligen Ressource entfernt.

jCoAP Blockwise Transfer

Neben CoAP Observe unterstützt jCoAP auch den blockweisen Datentransfer. Abbildung 35 gibt dabei einen Überblick über den Ablauf eines solchen Transfers. Nachfolgend wird der Verlauf sowohl eines blockweisen GET- als auch eines blockweisen PUT-Requests kurz erläutert.

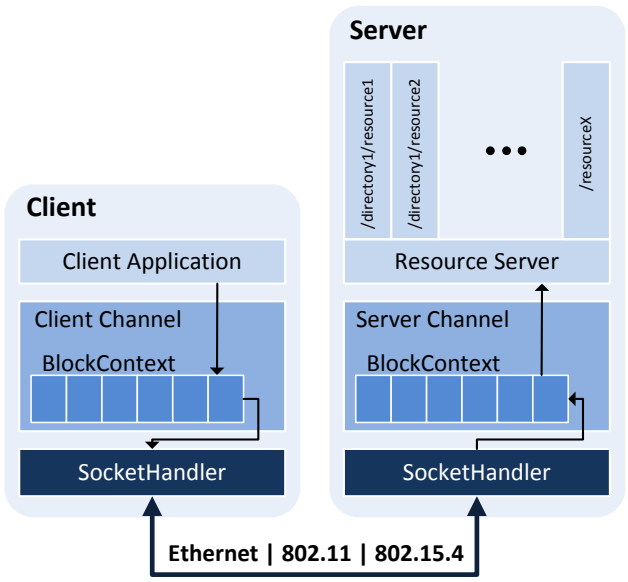

Abbildung 35: Funktionsweise des Blockwise Transfer mit jCoAP [61].

Will ein Client Daten mittels eines GET-Requests blockweise von einem Server abrufen, so setzt er in seinem Request die Block2-Option. Auf der Server-Seite wird jede eingehende Nachricht im ServerChannel auf das Vorhandensein einer Block-Option überprüft. Ist, wie in diesem Fall, eine Block2-Option vorhanden, wird zunächst überprüft, ob der Request einer bereits laufenden blockweisen Transaktion zuzuordnen ist. Handelt es sich bei der Nachricht um den ersten Request einer

blockweisen Transaktion, so wird sie an den ResourceServer weitergeleitet. Dieser ruft zunächst die Daten von der angefragten Ressource ab. Diese werden anschließend genutzt, um einen BlockContext in dem jeweiligen Channel anzulegen. Bei dem BlockContext handelt es sich im Wesentlichen um einen Datenpuffer. Für alle nachfolgenden Requests, die zu dieser Transaktion gehören, wird bereits im Channel automatisch der entsprechende Datenblock aus dem BlockContext ausgelesen und als Antwort an den Client gesendet. Will ein Client mittels eines PUT-Requests Daten blockweise an den Server senden, fügt er dem Request eine Block1-Option hinzu. Auf der Server-Seite wird zunächst wieder geprüft, ob der Request zu einer bereits laufenden Transaktion gehört. Ist dies nicht der Fall, so legt der Channel selbst einen BlockContext an und fügt diesem die in dem PUT-Request enthaltenen Daten hinzu. Für alle nachfolgenden PUT-Requests der Transaktion generiert der Channel automatisch die entsprechenden Antworten und fügt die enthaltenen Daten dem BlockContext hinzu. Am Ende der Transaktion liest der Channel alle Daten aus dem BlockContext aus und leitet diese in Form eines einzigen Nachrichtenobjektes an den ResourceServer weiter. Dieser führt dann entsprechend die PUT-Operation mit den vollständig empfangenen Daten auf der angefragten Ressource aus.

4.2 Evaluation der Interoperabilität

Um die Standard-Konformität von jCoAP nachzuweisen, wurden diverse Funktionstests zur Interaktion mit anderen CoAP-Implementierungen durchgeführt. Die Grundlage für diese Untersuchungen bildete ein vom European Telecommunications Standards Institute (ETSI) in Zusammenarbeit mit der IPSO Alliance herausgegebener Interoperabilitätstest. Dieser Test wurde für das IoT CoAP Plugtest-Event 2012 in Paris erarbeitet und sollte bereits während der Standardisierung von CoAP einen einheitlichen Rahmen zur Bewertung der Interoperabilität verschiedener Implementierungen schaffen [31]. Trotz einiger Veränderungen, die CoAP während der Standardisierung erfahren hat, bleibt die Bedeutung dieses Interoperabilitätstests als Standard zur Überprüfung der Interaktionsfähigkeit verschiedener CoAP-Implementierungen erhalten. In der Testbeschreibung wird dabei zwischen obligatorischen und fakultativen Tests unterschieden. Tabelle 8 gibt einen Überblick über die in dem Test enthaltenen Operationen. Darüber hinaus gibt die Testbeschreibung Auskunft über die zu verwendende Ressourcen-Struktur auf den CoAP-Servern und legt fest, welche der Ressourcen in den einzelnen Testszenarien zu verwenden sind.

Tabelle 8: Übersicht der durchzuführenden Operationen in dem vom ETSI und der IPSO Alliance spezifizierten Interoperabilitätstest [31].

#	Subspezifikation	Beschreibung
		Obligatorische Tests
1	CoAP Core	Durchführen einer GET-TransaKtion (CON-Modus)
2	CoAP Core	Durchführen einer POST-Transaktion (CON-Modus)
3	CoAP Core	Durchführen einer PUT-Transaktion (CON-Modus)
4	CoAP Core	Durchführen einer DELETE-Transaktion (CON-Modus)
5	CoAP Core	Durchführen einer GET-TransaKtion (NON-Modus)
6	CoAP Core	Durchführen einer POST-Transaktion (NON-Modus)
7	CoAP Core	Durchführen einer PUT-Transaktion (NON-Modus)
8	CoAP Core	Durchführen einer DELETE-Transaktion (NON-Modus)
9	CoAP Core	Durchführen einer GET-Transaktion im CON-Modus ohne Piggybacked Response (Separate Response)
10	CoAP Core	Verarbeitung eines Requests mit der Token Header Option
11	CoAP Core	Verarbeitung eines Requests ohne die Token Header Option
12	CoAP Core	Verarbeitung eines Requests mit mehrfach vorhandener URI-Path Option
13	CoAP Core	Verarbeitung eines Requests mit mehrfach vorhandener URI-Query Option
14	CoAP Core	Interaktion im CON-Modus mit Piggybacked Response
15	CoAP Core	Interaktion im CON-Modus mit verzögerter Response
16	CoAP Core	Interaktion im NON-Modus mit verzögerter Response
		Fakultative Tests
17	CoRE Link Format	Discovery über das Well-Known-Interface
18	CoRE Link Format	Nutzung von Filtered Requests zur Eingrenzung der Suchergebnisse
19	CoAP Blockwise Transfer	Durchführen einer blockweisen GET-Transaktion mit Early Negotiation.
20	CoAP Blockwise Transfer	Durchführen einer blockweisen GET-Transaktion mit Late Negotiation.
21	CoAP Blockwise Transfer	Durchführen einer blockweisen PUT-Transaktion
22	CoAP Blockwise Transfer	Durchführen einer blockweisen POST-Transaktion
23	CoAP Observe	Beobachtung einer Ressource
24	CoAP Observe	Beenden der Beobachtung einer Ressource
25	CoAP Observe	Clientseitige Erkennung der Deregistrierung (Max-Age)
26	CoAP Observe	Serverseitige Erkennung der Deregistrierung (Client OFF)
27	CoAP Observe	Serverseite Erkennung der Deregistrierung durch eine RST-Nachricht

Für die Umsetzung des zweiten Kommunikationspartners zur Durchführung der Interoperabilitätstests kamen zwei unterschiedliche CoAP-Implementierungen zum Einsatz. Zum einen wurde die Java-Implementierung Californium aus dem Eclipse Projekt verwendet, da es sich hierbei um die Referenzimplementierung aus dem Eclipse Projekt handelt. Zum anderen wurde das Firefox Browser PlugIn Copper genutzt, da hier die Testverfahren für den CoAP-Server schon vorimplementiert wurden. Da es sich bei Copper um einen reinen CoAP-Client handelt, konnte es nur in Verbindung mit einem jCoAP-Server eingesetzt werden. Die Evaluation des jCoAP-Clients erfolgte daher ausschließlich mit Californium.

In Rahmen der Interoperabilitätsprüfung wurden sämtliche obligatorischen sowie fakultativen Tests mit Ausnahme von Test Nr. 25 und 26 durchgeführt. Diesen stand entgegen, dass die Funktionen, welche in diesen Szenarien geprüft werden sollen, zum Testzeitpunkt noch nicht in jCoAP integriert waren. Die Tests wurden sowohl für den jCoAP-Server in Kombination mit einem Copper- und einem Californium-Client als auch mit einem jCoAP-Client und einem Californium-Server durchgeführt. Da bei dieser Untersuchung nur die funktionale Ebene der Protokoll-Implementierung im Vordergrund stand, wurden der Client und der Server jeweils auf demselben Gerät ausgeführt.

Die Interoperabilitätstests verliefen größtenteils erfolgreich. Einzelne reproduzierbare Fehler konnten einzig im Testfall Nr. 20 bei der Verwendung eines jCoAP-Servers in Verbindung mit Copper festgestellt werden. Bei der Verwendung des blockweisen Transfers in Verbindung mit Late Negotiation kam es in Einzelfällen zu einer fehlerhaften Darstellung des letzten Datenblockes in Copper. Das Auftreten des Fehlers war dabei abhängig von der verwendeten Blockgröße und der Gesamtgröße der abgefragten Ressource. Auf Grund der Tatsache, dass dieser Fehler bei der Nutzung eines Californium-Clients nicht auftrat, kann geschlussfolgert werden, dass es sich hierbei um einen Copper-spezifischen Fehler handelt. Copper wurde für diese Untersuchung in der Version 0.18.4 verwendet. Eine Prüfung, ob der Fehler in einer späteren Version von Copper behoben wurde, konnte nicht durchgeführt werden, da Copper nicht mehr mit der aktuellen PlugIn-Schnittstelle des Firefox-Browsers kompatibel ist. Die Entwicklung von Copper wird aktuell nicht mehr fortgeführt.

4.3 Performance-Analyse

Zur Analyse des Zeitverhaltens von jCoAP wurde eine beispielhafte Client-Server-Anwendung erstellt. Im Rahmen dieser Testanwendung bietet ein jCoAP-Server eine Echo-Ressource an, die aus einem einfachen Byte Array besteht. Der Client generiert zunächst pseudozufällige Nutzdaten mit einer Länge von 16 Byte. Diese werden mittels eines zuverlässigen PUT-Requests an die Echo-Ressource des Servers übersendet, um sie dieser als Wert zuzuweisen. Nachdem der Client eine ACK-Nachricht zu diesem Request erhalten hat, beginnt er damit, den Wert der Echo-Ressource mittels eines GET-Requests im CON-Modus abzurufen. Das Abrufen des Wertes wird dabei in einer Schleife 100 mal durchgeführt. Weiterhin wird in dem Experiment der blockweise Transfer mit einer Blockgröße von

16 Byte verwendet. Nach jedem Durchlauf der gesamten Schleife werden neue pseudozufällige Nutzdaten mit einer um 16 Byte erhöhten Länge generiert und der Echo-Ressource per PUT-Request zugewiesen. Anschließend wird die Abfrageschleife erneut ausgeführt. Dieses Vorgehen wird solange wiederholt, bis eine Nutzdatenlänge von 160 Byte erreicht ist. Während des gesamten Versuchablaufs misst der Client die Zeitspanne, die jede Transaktion in Anspruch nimmt. Als Beginn einer Transaktion wird dabei das Aufrufen der Send-Methode in der Client-Anwendung angenommen. Das Ende einer Transaktion ist erreicht, wenn die abgerufenen Daten vollständig in der Client-Anwendung zur Verfügung stehen. Die Messung der Zeitstempel erfolgte dabei mit der Java-eigenen System.TimeNano()-Funktion. Demzufolge wurde lediglich die Wall Clock Time gemessen, deren Genauigkeit systemlastabhängigen Schwankungen unterliegt. Um dennoch vergleichbare Ergebnisse zu erzielen, wurden alle Untersuchungen unter gleichen Lastbedingungen durchgeführt. Die Testanwendung wurde jeweils mit jCoAP und dem weit verbreiteten Californium umgesetzt. Abbildung 36 zeigt einen schematischen Aufbau der für die Untersuchung genutzten Testumgebung.

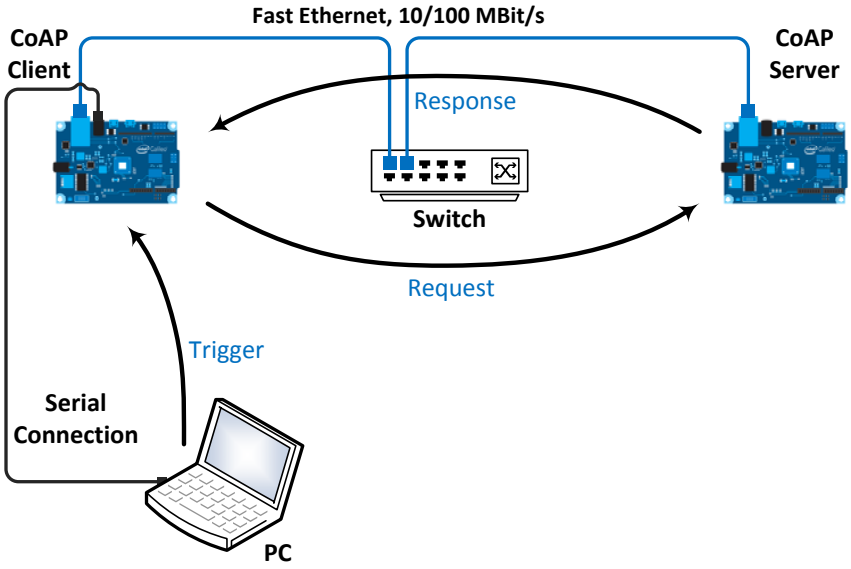

Abbildung 36: Aufbau der für die Performance-Messungen verwendeten Testumgebung [61].

Sowohl der Client als auch der Server wurden auf einem Intel Galileo Board der ersten Generation ausgeführt. Das Galileo Board verfügt über einen Quark X1000 SoC mit einer Taktfrequenz von 400 MHz und einem Rechenkern als Prozessor. Darüber hinaus ist es mit einem 256 MB großen Arbeitsspeicher, einer 10/100 MBit/s Fast-Ethernet-Schnittstelle und einem MicroSD-Kartenleser ausgestattet. Als Betriebssystem wurde ein angepasstes Embedded Linux mit einem Standard Linux Kernel in der Version 3.8 verwendet, welches mit Hilfe der Yocto Build Tools erstellt wurde. Als Java Virtual Machine kam die Java HotSpot Client Virtual Machine von Oracle in der Version 25.11-b03 mit ihrer Standardkonfiguration zum Einsatz. Die beiden Boards wurden mittels Fast Ethernet und einem Switch miteinander verbunden. Der PC war über die serielle Schnittstelle des Boards mit dem

Client verbunden und löste den Start der Testanwendung aus. Abbildung 37 zeigt die jeweils mit jCoAP und Californium erzielten Ergebnisse. Auf der x-Achse ist die jeweilige Größe der Nutzdaten abgebildet, während die y-Achse die Transaktionszeit in Millisekunden angibt. Die schwarzen Linien markieren dabei den Median der Transaktionszeiten über die jeweils 100 Messungen, während die eingefärbten Flächen die Verteilungsdichte der gemessenen Transaktionszeiten darstellen. Hierbei sind die hellgrauen Kurven jCoAP und die dunkelgrauen Kurven Californium zuzuordnen. Aufgrund der hohen Differenzen in den gemessenen Transaktionszeiten wird für die y-Achse eine logarithmische Einteilung verwendet, um die Darstellbarkeit zu verbessern.

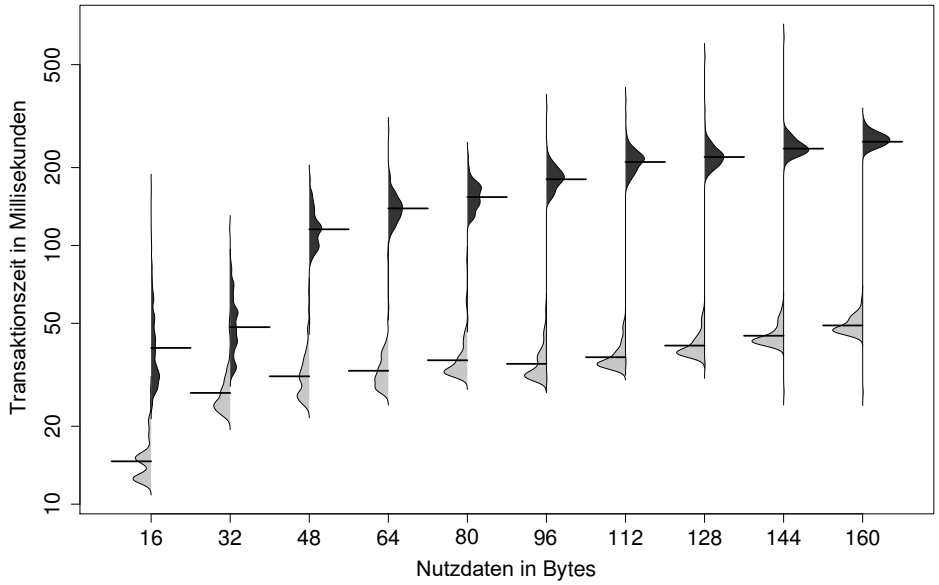

Abbildung 37: Gegenüberstellung der Transaktionszeiten von jCoAP und Californium [61].

Die Ergebnisse zeigen, dass die mit jCoAP umgesetzte Version der Testanwendung ein deutlich besseres Zeitverhalten aufweist als die auf Californium basierende Testanwendung. Die durchschnittliche Transaktionszeit bei einer Nutzdatengröße von 16 Byte in einem einzigen Paket ist mit jCoAP um 86,79% geringer als mit Californium. Dies lässt sich mit dem unterschiedlichen Aufbau der beiden Frameworks begründen. Californium stützt sich auf eine streng in Schichten aufgeteilte Struktur, wobei jede Schicht einen festgelegten Teil der CoAP-Spezifikation umsetzt. Jede Nachricht durchläuft dabei sämtliche Schichten, bevor sie schließlich von dem Server oder Client verarbeitet wird. Dies geschieht unabhängig davon, ob die jeweilige Funktion in der Kommunikation zum Einsatz kommt. Beim Durchlaufen der Ebenen entstehen zahlreiche Speicherkopien derselben Nachricht. Dies führt zum einen zu einer höheren Verzögerung durch die zusätzlichen Speicheroperationen. Zum anderen muss die Garbage Collection von Java deutlich häufiger ausgeführt werden, um den Speicher nicht mehr benötigter Objekte wieder freizugeben. Mit dem kanalbasierten Konzept von jCoAP können diese Effekte deutlich reduziert werden. Dies hat zur Folge, dass eingehende Nachrichten deutlich schneller in der Client-/Server-Anwendung verfügbar sind.

Allerdings zeigt sich in der Verteilungsdichte der Transaktionszeiten, dass beide Versionen der Testanwendung hohe Schwankungen in der Transaktionsdauer aufweisen. Die Standardabweichung beträgt bei einer Nutzdatengröße von 16 Byte für jCoAP 6,893 ms und für Californium 52,539 ms. Zudem lässt sich feststellen, dass die Standardabweichung für kleinere Nutzdatengrößen höher ist und mit steigender Größe der Nutzdaten abnimmt. Dies kann mit der Nutzung des blockweisen Transfers begründet werden. Für große Datenmengen werden hier in einer einzigen Transaktion mehrere Nachrichten gesendet, die jeweils in die Transaktionszeit eingehen. Je größer die Datenmenge ist, desto mehr Nachrichten müssen in einer Transaktion gesendet werden und desto höher ist auch die Wahrscheinlichkeit, dass einzelne Ausreißer wieder neutralisiert werden. Bei sehr kleinen Datenmengen wird hingegen nur eine einzige Nachricht gesendet, sodass Ausreißer hier sofort sichtbar werden. Unabhängig davon stellen die beobachteten zeitlichen Schwankungen die Eignung von jCoAP für den Einsatz in Echtzeit-Szenarien infrage. Daher werden im nächsten Kapitel die Ursachen für das nicht-deterministische Verhalten von jCoAP untersucht und Lösungsansätze beschrieben.

5 Deterministische Protokollverarbeitung in jCoAP

Wie sich in den vorangegangen Untersuchungen bestätigt hat, zeigen sowohl jCoAP als auch Californium ein nichtdeterministisches Ausführungsverhalten. In diesem Kapitel werden die Gründe für die Schwankungen in der Ausführungszeit untersucht und mögliche Lösungsmöglichkeiten aufgezeigt. Die in diesem Kapitel vorgestellten Ergebnisse wurden bereits veröffentlicht [61].

5.1 Preemptive Linux als Echtzeitbetriebssystem

Einen ersten Anhaltspunkt zur Klärung der Ursachen des beobachteten Zeitverhaltens bietet eine nährere Betrachtung des verwendeten Betriebssystems. In den Untersuchungen in Kapitel 4 wurde ein Standard Linux Kernel in der Version 3.8 verwendet. Hier werden die ausgeführten Anwendungen generell in zwei Kategorien eingeteilt, Kernel und User Space Tasks. Dabei stellen Kernel Tasks Systemprozesse, wie beispielsweise das Speichermanagement oder den Netzwerk-Stack des Betriebsystems, dar. Sie besitzen eine höhere Priorität bei der Abarbeitung als User-Space-Programme und können durch diese auch nicht unterbrochen werden. User Space Tasks sind hingegen alle Anwendungen, die von einem Benutzer des Betriebssystems gestartet wurden. Zwar kann ihnen auch eine Priorität zugewiesen werden, jedoch wird diese nur gegenüber anderen User Space Tasks berücksichtigt. Darüber hinaus können User Space Tasks jederzeit von Kernel Tasks unterbrochen werden, um die Abarbeitung des Kernel Tasks zu beschleunigen.

Auf diese Weise ergeben sich viele mögliche Task-Konstellationen, die zu einem zeitlich nicht vorhersagbaren Verhalten von User Tasks, wie etwa dem CoAP-Client bzw. -Server, führen können. Hierzu zählt unter anderem das Aufrufen von Kernel Tasks durch einen Interrupt, wie es beispielsweise beim Eintreffen eines Netzwerkpakets der Fall ist.

Diesem Problem kann auf zwei Arten begegnet werden. Zum einen könnte der CoAP-Client bzw. -Server als Kernel Task gestartet werden. Hierzu müsste die Anwendung als eigenes Kernel-Modul entwickelt werden. Dies würde jedoch erhebliche Eingriffe in das Betriebssystem erfordern. Zudem müsste das Betriebssystem immer wieder neu auf die aktuelle Anwendung angepasst werden [15]. Einen universelleren Ansatz bietet der Preemptive Patch für den Linux Kernel, dessen Entwicklung von Ingo Molnar und Thomas Gleixner gestartet wurde [44]. Dieser Patch bringt mehrere Veränderungen des Linux Kernels und seines Schedulers mit sich, von denen zwei von besonderer Bedeutung sind. Zunächst werden neue Prioritätsstufen, die einen Task als Echtzeit-Prozess kennzeichnen, und Scheduling Policies hinzugefügt.[2] Weiterhin werden in alle Kernel Tasks sogenannte Preemption Points eingefügt. Sie erlauben es dem Scheduler, wie zuvor sonst nur bei User Space

[2] Die Prioritätsstufen und Scheduling Policies sind zu großen Teilen auch schon in den Main Line Kernel eingeflossen, sodass die hauptsächliche Änderung des Preemptive Patch in der Einführung der Preemption Points liegt [58]

Tasks möglich, die Ausführung eines Kernel Tasks an dieser Stelle zu unterbrechen und einem anderen Prozess die Kontrolle über die Systemressourcen zu geben [35, 44]. Dabei werden vom Scheduler nicht nur andere Kernel Tasks, sondern auch diejenigen User-Space-Prozesse mit einer Echtzeitpriorität berücksichtigt. Bei der Verwendung eines präemptiven Linux Kernels können Kernel Tasks also auch von User Space Tasks unterbrochen werden. Weiterhin werden auch (Soft) Interrupt Handler als unterbrechbare Kernel Tasks implementiert [44]. Dies führt zu einer deutlich gesteigerten Reaktionsfähigkeit der User-Space-Prozesse und somit zu einem besser vorhersagbaren Zeitverhalten.

Um diese theoretischen Vorteile auf ihre Praxistauglichkeit hin zu untersuchen, wurde das in Kapitel 4.3 beschriebene Experiment wiederholt. Allerdings wurde diesmal ein präemptiver Linux Kernel verwendet. Zudem wurden der CoAP-Client und der Server jeweils mit der höchsten Echtzeitpriorität gestartet. Abbildung 38 zeigt die in diesem Versuch erzielten Ergebnisse. Wie auch in Kapitel 4.4 zeigt die x-Achse die jeweilige Nachrichtengröße, während die y-Achse die Transaktionsdauer in Millisekunden darstellt. Die schwarzen Linien kennzeichnen den Median der gemessenen Transaktionszeiten. Die Kurven zeigen die Verteilungsdichte der gemessenen Zeiten. Auf Grund des großen Wertebereichs der gemessenen Transaktionszeiten wurde für die Zeitachse eine logarithmische Skala gewählt.

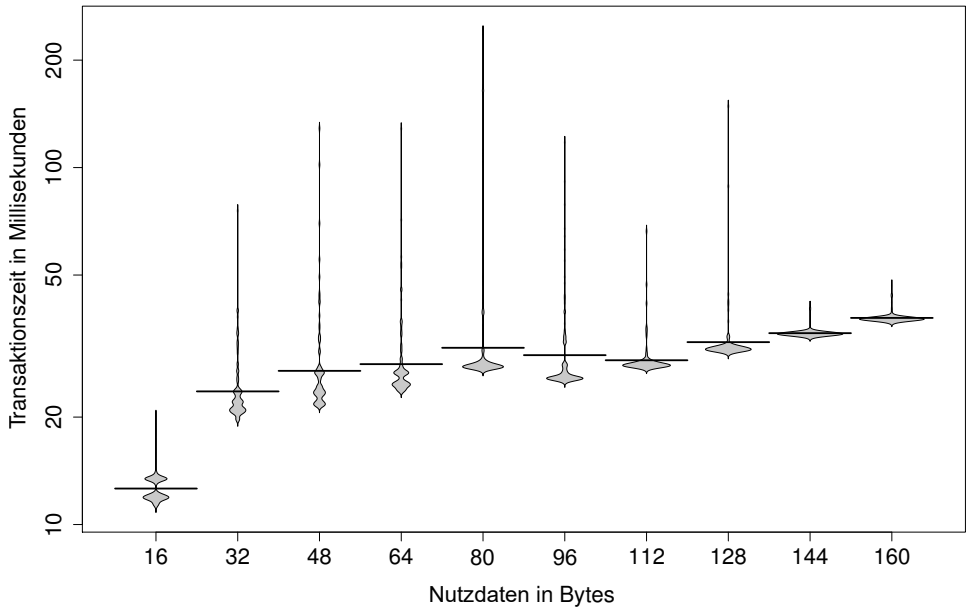

Abbildung 38: Transaktionszeiten von jCoAP unter Verwendung der Oracle JVM und eines vollständig präemptiven Linux Kernels [61].

Es zeigt sich, dass die Streuungen in den Transaktionszeiten mit Hilfe eines präemptiven Linux Kernels deutlich reduziert werden können. Allerdings treten weiterhin einige erhebliche Ausreißer auf, welche sich zum einen negativ auf die durchschnittliche Transaktionszeit auswirken. Zum anderen sind so starke Ausreißer in Anwendungsfällen mit harten Echtzeitanforderungen nicht hinnehmbar,

zumal das Auftreten dieser Ausreißer nicht vorhergesagt werden kann. Daher werden in Kapitel 5.2 die Ursachen für das beobachtete Verhalten geklärt und Lösungsmöglichkeiten untersucht.

5.2 Echtzeitfähige Java Virtual Machines

Im vorangegangenen Kapitel konnte gezeigt werden, dass ein präemptiver Linux Kernel einen positiven Einfluss auf das Zeitverhalten von User-Space-Prozessen hat. Allerdings traten bei dem praktischen Versuch weiterhin erhebliche, nicht vorhersagbare Ausreißer in den Transaktionszeiten auf. Diese Ausreißer lassen sich auf die Java Virtual Machine (JVM) zurückführen. Java ist eine plattformunabhängige Programmiersprache. Die Plattformunabhängigkeit erreicht sie durch die Abstraktion der Laufzeitumgebung in einer JVM. Im Ergebnis muss jeweils nur die JVM an ein Zielsystem angepasst werden, um sämtliche Java-Anwendungen auf diesem System auszuführen. Dabei übernimmt die JVM wichtige Aufgaben, wie das Planen der Java-internen Prozesse und die Speicherverwaltung. Insbesondere die Speicherverwaltung kann einen erheblichen Einfluss auf das Zeitverhalten einer Java-Anwendung haben. Die JVM verwaltet typischerweise einen Stack- und einen Heap-Speicher. In beiden Speicherbereichen wird von Java-Anwendungen während ihrer Lebenszeit Speicher allokiert. Allerdings bietet Java keine sprachlichen Mittel und Systemfunktionen, um den allokierten Speicher zu festgelegten Zeiten wieder freizugeben. Hierfür kommt in Java die sogenannte Garbage Collection (GC) zum Einsatz. Sie wird automatisch von der JVM gestartet, wenn eine konfigurierbare Schranke beim belegten Speicher überschritten wurde. Bei der GC wird der belegte Speicher auf in der Anwendung nicht mehr referenzierte Objekte (Garbage) hin untersucht. Der durch diese Objekte belegte Speicher wird dann durch den Garbage Collector wieder freigegeben [66, 80]. Dieses Vorgehen hat zwei entscheidende Vorteile. Auf der einen Seite kann sich der Entwickler einer Java-Anwendung voll auf die Anwendung konzentrieren ohne sich mit der Speicherverwaltung zu beschäftigen. Auf der anderen Seite reagiert die JVM automatisch flexibel auf die vom Zielsystem bereitgestellte Speichergröße. Allerdings ist weder der Zeitpunkt des Aufrufens der GC noch die Zeitspanne, welche die GC in Anspruch nimmt, vorhersagbar. Darüber hinaus handelt es sich bei der GC um ein sogenanntes „Stop the World“-Ereignis, bei dem alle Anwendungen bis zum Abschluss der GC pausiert werden [66]. Dies kann erhebliche Ausreißer in der Ausführungszeit nach sich ziehen, wie sie auch in Kapitel 5.1 beobachtet werden konnten.

Um diesem Problem zu begegnen, wurde durch die Real-Time Java Experts Group, einem internationalen Gremium aus verschiedenen Vertretern von Industrie und Forschung, die Real-Time Specification for Java (RTSJ) entworfen [12]. Sie soll die zeitlich deterministische Ausführung von Java-Anwendungen ermöglichen. Die RTSJ sieht zwei Wege vor, die GC zeitlich vorhersagbar zu gestalten. Der erste Weg ist es, die GC als eigenen Task in den Schedule einzureihen, sodass sie in regelmäßigen zeitlichen Abständen für die immer gleiche Dauer ausgeführt wird. Auf diese Weise sind sowohl der Zeitpunkt der GC als auch ihre Dauer vorhersehbar und können in dem restlichen Programmab-

lauf berücksichtigt werden. In der zweiten Variante wird bei jeder neuen Speicherallokation dieselbe Speichermenge durch den GC auch wieder freigegeben. Beide Varianten führen zu einem ähnlichen Ergebnis. Auf Grund der leichteren Umsetzbarkeit hat sich in der Praxis allerdings die erste Variante durchgesetzt.

Eine JVM, welche die RTSJ vollständig umsetzt, ist die JamaicaVM der Aicas GmbH [5]. Auch bei der JamaicaVM wird die GC als eigenständiger Task ausgeführt. Die JamaicaVM bietet zwei Möglichkeiten, eine Java-Anwendung als Echtzeit-Task auszuführen. Zum einen kann die Anwendung direkt über die JamaicaVM gestartet werden. Zum anderen kann über den JamaicaBuilder mittels Cross Compiling eine eigenständig auf dem Zielsystem ausführbare Datei erstellt werden. Hierzu wird zunächst der kritische Pfad innerhalb der Java-Anwendung mittels Profiling bestimmt. Anschließend werden alle Programmteile, welche Teil dieses Pfades sind, in C-Code umgewandelt und für die Zielarchitektur kompiliert. Der so erstellte Maschinencode wird mit der JVM und dem verbleibenden Java Byte Code in eine einzige ausführbare Datei zusammengeführt. Ziel ist es dabei, die Länge des kritischen Pfades zu minimieren und so eine schnellere Ausführung der gesamten Anwendung zu gewährleisten.

In einem dritten Experiment wurden die Auswirkungen einer echtzeitfähigen JVM auf das Zeitverhalten des jCoAP-Clients und -Servers untersucht. Hierzu wurde der Versuchsaufbau aus Kapitel 5.1 um die JamaicaVM erweitert. Der Versuch wurde mit beiden Ausführungsvarianten der JamaicaVM unter Verwendung eines vollständig präemptiven Linux Kernels durchgeführt. Abbildung 39 zeigt die erzielten Ergebnisse sowohl für die reine Java-Version als auch für die mit dem JamaicaBuilder vorkompilierte Version. Hierbei zeigt die x-Achse die Datenmenge in Byte und die y-Achse die erzielten Transaktionszeiten. Die logarithmische Zeitskala soll hierbei die Darstellbarkeit verbessern. Die schwarzen Linien kennzeichnen den Median der gemessenen Zeiten, während die Kurven deren Verteilungsdichte darstellen. Dabei gehören die Kurven auf der jeweils linken Seite zu den mit der JamaicaVM erzielten Ergebnissen. Die Kurven auf der jeweils rechten Seite visualiseren die mit dem JamaicaBuilder gewonnenen Messwerte.

Die Ergebnisse zeigen, dass die Nutzung einer JVM mit echtzeitfähiger GC einen bedeutenden Effekt auf den zeitlichen Determinismus der Anwendung hat. So konnte die Streuung bei Transaktionen mit einer Nutzlast von 16 Byte um 95% reduziert werden. Somit kann die These, dass die beobachteten Streuungen auf die nicht-deterministische GC der Standard-JVM zurückzuführen sind, bestätigt werden. Weiterhin wird deutlich, dass der Geschwindigkeitsgewinn durch die Vorkompilierung mit dem JamaicaBuilder mit 7,3% bei einer Datengröße von 16 Byte nur minimal ausfällt. Zudem kann ein linearer Anstieg der Transaktionszeiten mit steigender Datengröße beobachtet werden. Dieser ergibt sich aus der Nutzung des blockweisen Datentransfers. Mit steigender Datengröße müssen hier mehrere Request-Response-Paare für eine einzelne Transaktion versendet werden. Dies führt bei einer Blockgröße von 16 Byte und eine um jeweils 16 Byte ansteigende Datengröße zu einem linearen Anstieg. Allerdings wird auch deutlich, dass sich die Transaktionszeit

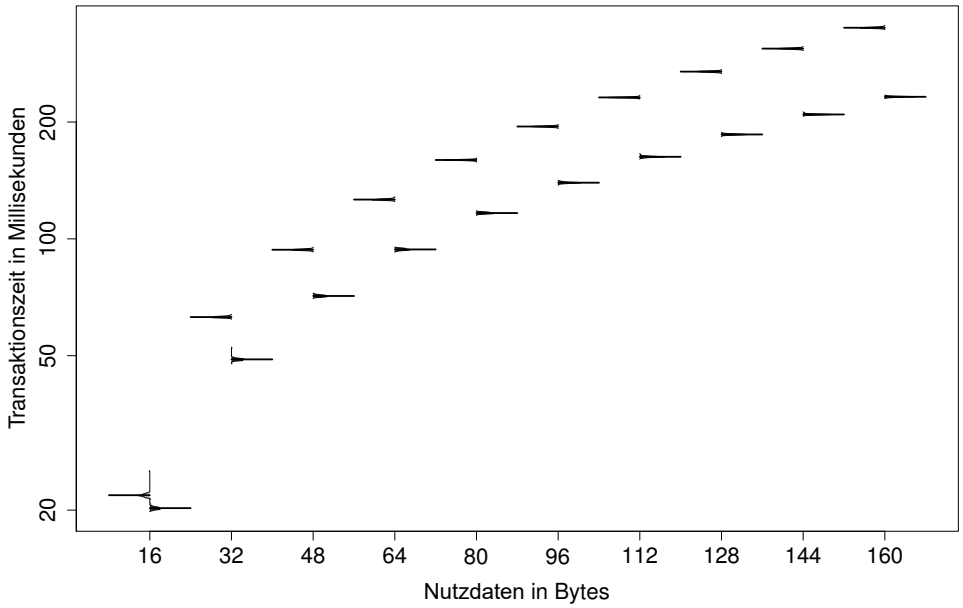

Abbildung 39: Transaktionszeiten von jCoAP unter Verwendung der JamaicaVM (die jeweils linke Kurve) bzw. des JamaicaBuilders (die jeweils rechte Kurve) und eines vollständig präemptiven Linux Kernels [61].

gegenüber den vorherigen Messungen deutlich erhöht hat. Dies lässt sich ebenfalls mit der Verwendung eines zeitlich deterministischen GC begründen. Um sicherzustellen, dass in den fest eingeplanten Zeitschlitzen, in denen die GC ausgeführt wird, hinreichend viel Speicher wieder freigegeben werden kann, müssen diese Zeitschlitze ausreichend groß gewählt werden. Da diese Zeitschlitze aber auch immer gleich groß sein müssen, um zeitlich exakt vorhersagbar zu sein, orientiert sich ihre Größe am Worst-Case. Dies führt zu erheblichen Zeitpuffern, die nur in den seltensten Fällen tatsächlich benötigt werden und die Ausführungsgeschwindigkeit der gesamten Anwendung negativ beeinflussen. Bei einer nicht-deterministischen GC orientiert sich die Dauer der GC hingegen nur an dem im aktuellen Durchlauf freizugebenden Speicher. Für die Echtzeitfähigkeit ist allerdings die zeitliche Vorhersagbarkeit des Programmablaufs von größter Bedeutung, sodass die reduzierte Ausführungsgeschwindigkeit zunächst in den Hintergrund tritt. Darüber hinaus genügen die erzielten Transaktionszeiten zumindest den Anforderungen in der Heim- und Gebäudeautomation mit Antwortzeiten von bis zu 100 ms [37]. Die RTSJ erweitert Java zudem um Werkzeuge, mit denen sich die für die GC vorgehaltenen Zeitpuffer deutlich reduzieren lassen [48]. Dabei handelt es sich allerdings um reine Programmiertechniken ohne eigene wissenschaftliche Bedeutung, sodass an dieser Stelle nicht weiter darauf eingegangen wird. Im Kapitel 9.3 - Ausblick werden diese Werkzeuge dennoch kurz vorgestellt. Weiterhin ist zu erwarten, dass der negative Einfluss der echtzeitfähigen GC auf die Ausführungsgeschwindigkeit bei der Verwendung leistungsstärkerer Hardware weiter in den Hintergrund tritt. In [55] wurden die Ausführungszeiten eines Java-basierten Kommunikations-Stacks aus der Medizintechnik auf verschiedenen Hardware-Plattformen untersucht. Dabei konnte

gezeigt werden, dass sich mit einem RaspberryPi 3[3] im Vergleich zu einem Intel Galileo Board ein Geschwindigkeitszuwachs bis zu einem Faktor von 3,4 erreichen lässt.

5.3 Drahtlose Kommunikation

Zur Evaluation von jCoAP in drahtlosen Netzwerken wurden die verwendeten Intel Galileo Boards über ihre USB-Schnittstelle mit einem WLAN-Adapter verbunden. In dem verwendeten WLAN-Adapter kommt das RT 2870 USB Chipset der Firma Ralink zum Einsatz. Für den Versuchsaufbau wurden der Client und der Server direkt über ein WLAN-Ad-Hoc-Netzwerk miteinander verbunden. Als Funkstandard wurde IEEE 802.11g mit einer maximalen Bruttodatenrate von 54 MBit/s verwendet, wobei unter Linux die Wahl der Senderate standardmäßig durch einen Ratenkontrollalgorithmus in Abhängigkeit der Kanalbedingungen erfolgt. Für den Versuch wurde die rein Java-basierte Testanwendung aus den vorherigen Versuchen mit der JamaicaVM unter Verwendung eines präemptiven Linux-Kernels ausgeführt. Bei der Versuchsdurchführung konnten keine anderen WLAN-Netzwerke im 2,4-GHz-Band in der Umgebung festgestellt werden. Abbildung 40 zeigt die erzielten Ergebnisse. Auf der y-Achse wird die gemessene Transaktionszeit in Millisekunden abgebildet. Für eine bessere Sichtbarkeit der Effekte wurde eine logarithmische Skala für die Transaktionszeit gewählt. Auf der x-Achse wird die jeweilige Nutzdatenmenge angegeben. Die Kurven zeigen die Verteilung der Transaktionszeiten, wobei die jeweiligen schwarzen Striche den Median über die gemessenen Transaktionszeiten darstellen.

Die Ergebnisse zeigen, dass sich der Median der Transaktionszeiten im Vergleich zu den Versuchen mit drahtgebundenem Ethernet nur minimal nach oben verschiebt. Jedoch zeigen die WLAN-Messungen auch in diesem einfachen Szenario mit nur zwei Netzwerkteilnehmern und ohne Störeinflüße durch andere WLAN-Netzwerke deutliche Ausreißer mit Transaktionszeiten von mehreren Sekunden. Diese sporadisch auftretenden hohen Transaktionszeiten lassen sich auf die Störanfälligkeit von drahtloser Kommunikation zurückführen. Die Störungen in der Kommunikation können dabei entweder durch andere Netzwerkteilnehmer (konkurrierender Medienzugriff) oder andere externe Sender im gleichen Frequenzband (z.B. Bluetooth) verursacht werden. Um Kollisionen auf dem Kommunikationsmedium zu vermeiden, kommt bei WLAN das in Kapitel 3.1.4 bereits erläuterte CSMA/CA-Verfahren zum Einsatz. Daraus ergibt sich für aufeinanderfolgende Sendeversuche, in denen das Kommunikationsmedium belegt ist, eine exponentiell steigende Wartezeit vor dem jeweils nächsten Sendeversuch. Dies führt zu unvorhersagbaren Transaktionszeiten und somit zu einem nicht-deterministischen Kommunikationsverhalten. Techniken, die gleichzeitige Sendeversuche mehrerer Netzwerkteilnehmer ausschließen sollen, werden in den nachfolgenden Kapiteln betrachtet. Diese beziehen sich jedoch nur auf den auf Anwendungsebene von CoAP generierten Datenverkehr und haben keinen Einfluss auf den bei WLAN auf den tieferen Schichten generierten Management-

[3] ARM Cortex-A53, 64Bit ARMv8 Architektur mit 1200 MHz CPU-Takt und 1024 MByte RAM.

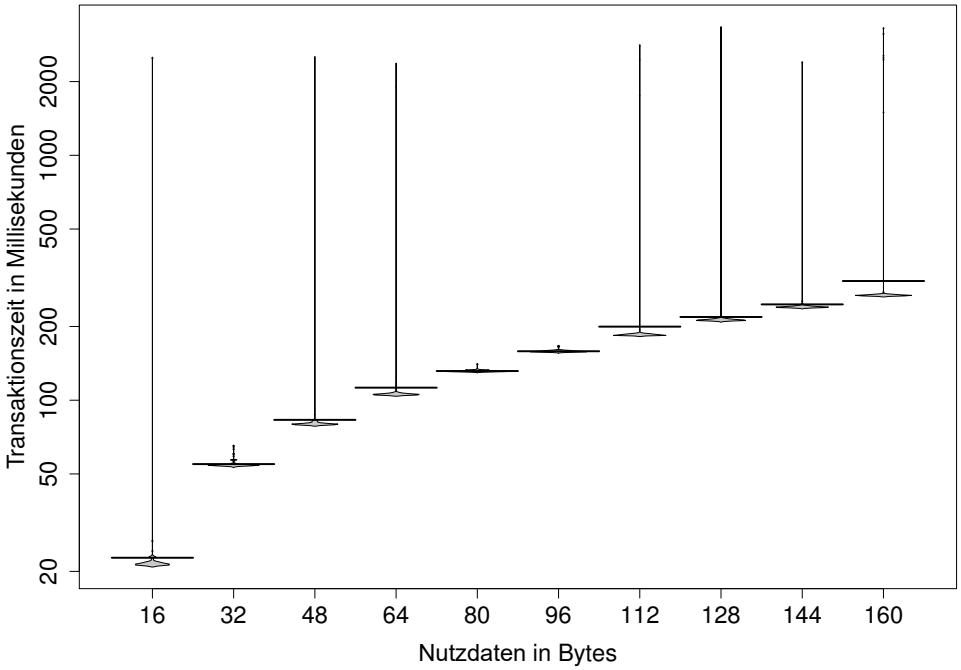

Abbildung 40: Transaktionszeiten von jCoAP unter Verwendung der JamaicaVM und eines vollständig präemptiven Linux Kernels über WLAN (IEEE 802.11g) [61].

Datenverkehr. Weiterhin ist es bei drahtloser Kommunikation nahezu unmöglich, Störungen zu unterbinden, die ihren Ursprung außerhalb des kontrollierten Netzwerkes haben. Daraus ergibt sich, dass drahtlose Kommunikationstechnologien kaum für den Einsatz in Umgebungen mit harten oder strengen Echtzeitanforderungen geeignet sind. Da die Anzahl der Ausreißer für das hier untersuchte, stark vereinfachte Szenario schon verhältnismäßig hoch ist, kann geschlussfolgert werden, dass WLAN-Technologien auch nicht ohne weiteres in Szenarien mit weicher Echtzeit eingesetzt werden können. Jedoch existieren mit RT-WiFi und LiT MAC vielversprechende Forschungsansätze, um auch mit WLAN weiche Echtzeitanforderungen erfüllen zu können [42, 69, 108].

5.4 Blockweiser oder nicht-blockweiser Datentransfer?

In den Experimenten der vorangegangen Abschnitte wurde der blockweise Datentransfer von CoAP für den Datenaustausch zwischen Client und Server genutzt. Dabei werden größere Datenmengen in mehrere Blöcke aufgeteilt und in separaten Request-Response-Paaren einzeln übertragen. Der blockweise Datentransfer hat den großen Vorteil, dass sich die Übertragung der Daten an beliebiger Stelle ohne großen Aufwand unterbrechen und später fortsetzen lässt. Diese Eigenschaft ist sehr vorteilhaft, wenn im Netzwerk beispielsweise ein TDMA-Verfahren für den Medienzugriff angewendet wird und jeder Netzwerkteilnehmer nur über einen begrenzten Zeitraum auf das Kommunikationsmedium zugreifen kann. Allerdings führt der blockweise Datentransfer durch die große Zahl der Einzelnachrich-

ten zu einem erhöhten Kommunikationsaufwand und längeren Transaktionszeiten. Daraus ergibt sich die Frage nach der richtigen Wahl der Blockgröße, sodass einerseits der zusätzliche Kommunikationsaufwand minimiert wird und gleichzeitig die Fähigkeit, den Datentransfer zu pausieren, erhalten bleibt. Um dieser Frage nachzugehen, wurde ein viertes Experiment durchgeführt. Hierbei wurde die Testanwendung aus den vorangegangenen Kapiteln so verändert, dass die in CoAP definierte maximale Blockgröße von 1024 Byte statt von bisher 16 Byte verwendet wird. Die Auswahl der Blockgröße erfolgte dabei in Hinblick auf die maximale Größe eines Ethernet-Frames von 1514 Byte, sodass sich eine einzelne CoAP-Nachricht, unabhängig von der Gesamtgröße der zu übertragenden Daten, in einem einzigen Frame transportieren lässt. Weiterhin wurde der Anstieg der Nutzdaten pro Schleifendurchlauf auf 128 Byte erhöht. Abbildung 41 zeigt die mit diesen Abwandlungen erzielten Transaktionszeiten. Dabei zeigt die y-Achse die Transaktionszeit in Millisekunden und die x-Achse die Größe der übertragenen Nutzdaten. Die Kurven stellen die Verteilungsdichte der gemessenen Zeiten dar, während die schwarzen Linien den Median der gemessenen Zeiten zeigen.

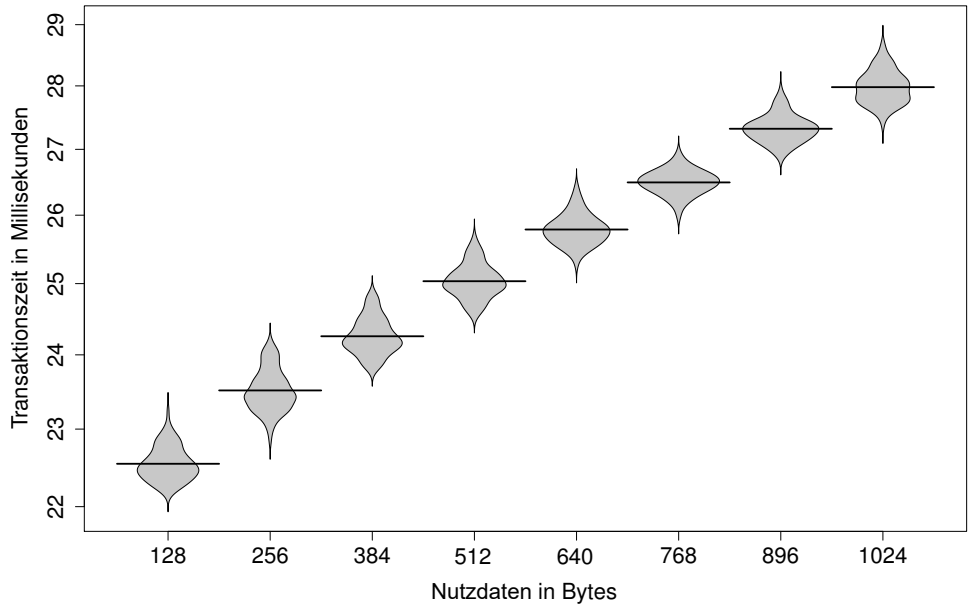

Abbildung 41: Transaktionszeiten von jCoAP mit steigender Nutzdatengröße in einem Paket unter Verwendung der JamaicaVM und eines vollständig präemptiven Linux Kernels [61].

In den Ergebnissen zeigt sich, dass die Transaktionszeit nur leicht ansteigt, auch wenn die Nutzdatengröße in einer CoAP-Nachricht maximiert wird. Dieser Anstieg lässt sich auf die leicht erhöhte Verarbeitungszeit auf der Client- und Server-Seite, bevor die Daten in der jeweiligen Anwendung zur Verfügung stehen, zurückführen. Die vorangegangenen Experimente haben bereits gezeigt, dass der Zeitbedarf beim Versenden mehrerer Datenblöcke anstelle eines größeren Pakets deutlich höher ausfällt. Allerdings würde sich die Transaktionszeit ebenfalls deutlich erhöhen, wenn eine einzelne Nachricht die Größe der Maximum Transmission Unit (MTU), also der maximal mit einem Frame

übertragbaren Datenmenge, überschreiten würde, da nun mehrere Frames versendet werden müssten. Zudem würde die Fragmentierung auf den unteren Schichten dazu führen, dass die Kommunikation auf den oberen Schichten nicht jederzeit unterbrochen werden könnte. Es lässt sich also schlussfolgern, dass die Blockgröße jeweils in Hinblick auf die gewählte Kommunikationstechnologie gewählt werden muss. Auf diese Weise lässt sich eine Fragmentierung der Daten und die daraus resultierende mangelnde Kontrolle der Anwendung über den Kommunikationsablauf unterbinden. Bei 6LoWPAN beträgt die MTU beispielsweise nur 127 Byte, die teilweise auch von dem IPv6- und dem UDP-Header belegt werden, sodass für Nutzdaten eine deutlich geringere Größe pro Frame zur Verfügung steht[4] [47].

5.5 Zwischenfazit

Bisher konnte gezeigt werden, dass es sich bei jCoAP um eine sehr effiziente Umsetzung des Constrained Application Protocol handelt. Ein Vergleich mit anderen bereits etablierten Implementierungen auf Java-Basis hat gezeigt, dass jCoAP deutlich bessere Performance bei einem ähnlichen Funktionsumfang bietet. Somit ist es deutlich besser sowohl für den Einsatz auf ressourcenbeschränkten Geräten als auch in Umgebungen mit Echtzeitanforderungen geeignet. Weiterhin konnte gezeigt werden, dass sich mit Hilfe eines vollständig präemptiven Linux Kernels und einer echtzeitfähigen JVM ein zeitlich deterministisches Kommunikationsverhalten erzielen lässt. Zudem wurde untersucht, wie der blockweise Datentransfer von CoAP in Echtzeitszenarien eingesetzt werden kann und welche Vorteile sich daraus ergeben. Allerdings wurden in den durchgeführten Experimenten immer nur jeweils zwei Geräte verwendet, sodass zumindest bei der drahtgebundenen Kommunikation kein konkurrierender Netzwerkzugriff stattgefunden hat. Zudem ist durch die niedrige Anzahl von Geräten die Auslastung des Netzwerks und der Infrastrukturgeräte sehr gering, wodurch es nicht zu Ressourcenengpässen im Netzwerk kommt. In realen Szenarien hingegen ist mit einer Vielzahl von Geräten und parallelen Kommunikationsvorgängen zu rechnen. Hier treten negative Effekte, wie das probabilistische Medienzugriffsverfahren bei WLAN oder Puffer-Zeiten in Switches, deutlich in den Vordergrund, sodass sich für die einzelnen Kommunikationsvorgänge keine zeitlichen Vorhersagen treffen lassen. Mögliche Lösungsansätze für dieses Problem werden in den nachfolgenden Kapiteln untersucht.

[4] Aber immer mindestens 33 Byte.

6 Echtzeitfähiger Kanalzugriff mit jCoAP

In diesem Abschnitt wird ein Ansatz für den deterministischen Zugriff auf das Kommunikationsmedium beleuchtet, der sich nahtlos in CoAP integrieren lässt und so auf zusätzlichen Overhead durch andere Kommunikationsprotokolle verzichtet. Zunächst wird hierfür eine CoAP-basierte Möglichkeit der Zeitsynchronisation zwischen den einzelnen Netzwerkteilnehmern untersucht. Anschließend wird ein TDMA-basierter Ansatz zur Koordination des Netzwerkzugriffs auf Grundlage der gemeinsamen Zeitbasis aller Geräte beschrieben. Die dargestellten Konzepte werden prototypisch umgesetzt und in das jCoAP-Framework integriert. Anschließend wird das Verhalten des Prototypen untersucht und bewertet. Die in diesem Kapitel vorgestellten Ergebnisse wurden teilweise veröffentlicht [59].

6.1 Zeitsynchronisation in jCoAP

Um eine effiziente Zeitsynchronisation umzusetzen, können entweder zentralisierte oder dezentralisierte Ansätze zum Einsatz kommen. Dabei hängt es von dem Einsatzszenario ab, welcher Ansatz zu bevorzugen ist.

So sind zentralisierte Verfahren zur Zeitsynchronisation in der Regel weniger komplex und daher deutlich leichter umzusetzen. Die geringere Komplexität führt überdies zu einem niedrigeren Ressourcenbedarf auf den Client-Geräten. Allerdings kann sich ein zentralisierter Zeitserver schnell als Flaschenhals erweisen, wenn die Anzahl der zu synchronisierenden Clients steigt. Zudem stellt der Zeitserver bei einem zentralisierten Ansatz einen SPoF dar, da ohne ihn keine Synchronisation der Systemzeiten zwischen den einzelnen Geräten mehr möglich ist. Demzufolge mangelt es zentralisierten Ansätzen an Skalierbarkeit und Zuverlässigkeit.

Vollständig dezentrale Ansätze, wie das in HaRTKad umgesetzte Synchronisationsverfahren, verfügen über deutlich bessere Eigenschaften hinsichtlich der Skalierbarkeit, da der Synchronisationsaufwand gleichmäßig im Netzwerk verteilt wird. Zudem vermeiden sie das Problem eines SPoF, da der Ausfall eines einzelnen Gerätes nicht zu einem Verlust der Funktionalität führt. Allerdings sind verteilte Verfahren deutlich komplexer hinsichtlich ihrer Implementierung. Zudem erfordern sie in der Regel mehr Systemressourcen auf den einzelnen Geräten. Weiterhin ist die Wartung des Systems erschwert, weil sich Fehlerquellen in vollständig dezentralen System nur sehr schwer lokalisieren lassen.

Es kann also geschlussfolgert werden, dass zentralisierte Ansätze eher für Umgebungen mit wenigen Geräten geeignet sind, die starken Ressourcenbeschränkungen unterliegen. In großen Netzwerken mit vielen Teilnehmern sind hingegen verteilte Verfahren zu bevorzugen. Allerdings bleibt zu berücksichtigen, dass die Nachteile von zentralisierten System bei der Zuhilfenahme mehrerer untereinander synchronisierter Zeitserver zumindest teilweise kompensiert werden können.

Im Rahmen dieser Arbeit wird aufgrund des geringeren Ressourcenbedarfs ein zentralisiertes Verfahren untersucht, welches sich nahtlos in CoAP integrieren lässt. Dieses Verfahren wird allerdings im weiteren Verlauf dieser Arbeit so erweitert, dass es den Anforderungen moderner Automatisierungs- und IoT-Szenarien bezüglich der Skalierbarkeit und Robustheit gerecht wird.

Um die CoAP-basierte Zeitsynchronisation zu ermöglichen, wird zunächst ein Zeitserver bestimmt. Dieser kann entweder im Vorfeld vom Entwickler festgelegt oder über eine Wettlaufstrategie zur Laufzeit ermittelt werden. Weiterhin wird eine neue Ressource mit der URI *„/well-known/time“* eingeführt. Diese Ressource muss von jedem CoAP-Server bereitgestellt werden, der die Rolle eines Zeitservers übernehmen kann. Weiterhin wird eine neue CoAP-Header-Option, die SYN-Option, eingeführt. Als Optionswert enthält diese eine Sequenznummer, welche später die Zuordnung von Requests und Responses ermöglichen soll. Weiterhin enthält die SYN-Option ein Auswahlfeld für das gewünschte Synchronisationsverfahren. Auf diese Weise können zum einen mehrere Synchronisationsverfahren umgesetzt und die Menge der verfügbaren Verfahren frei erweitert werden. Zum anderen kann das Synchronisationsverfahren entsprechend des Einsatzszenarios gewählt werden, sodass der optimale Trade-Off zwischen dem Synchronisationsaufwand und der benötigten, szenarioabhängigen Synchronisationsgenauigkeit erreicht wird. Zunächst wird im Rahmen dieser Arbeit *Cristians Algorithmus* umgesetzt [21]. Hierbei handelt es sich um ein besonders einfaches Verfahren zur Zeitsynchronisation mit sehr begrenzter Genauigkeit. Allerdings eignet es sich auf Grund seiner geringen Komplexität hervorragend zur prototypischen Evaluierung des Gesamtkonzeptes. Der Ablauf von *Cristians Algorithmus* ist in Abbildung 42 graphisch dargestellt.

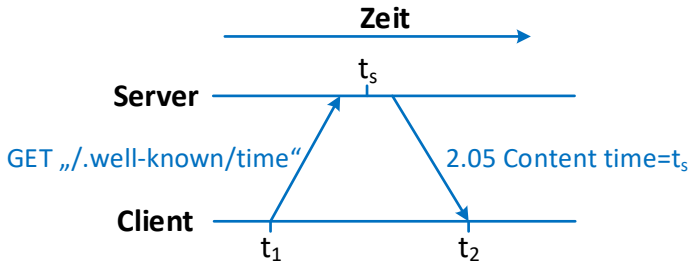

Abbildung 42: Ablauf von *Cristians Algorithmus* mit der vorgeschlagenen CoAP-Erweiterung zur Zeitsynchronisation.

Zunächst sendet der Client einen GET-Request nach der aktuellen Zeit an die *„/well-known/time“*-Ressource des Servers und misst dabei die Absendezeit t_1. Der Request muss hierbei eine SYN-Option enthalten, um ihn als Synchronisationsrequest zu kennzeichnen und den gewünschten Synchronisationsmodus festzulegen. Der Server verarbeitet den Request und versendet eine Antwort mit dem Zeitstempel t_s. Der Client empfängt die Response und misst die Empfangszeit t_2. Nun berechnet der Client die Round Trip Time (RTT), also die Umlaufzeit seiner Anfrage, nach der Formel (8).

$$RTT = t_2 - t_1 \tag{8}$$

Anschließend kann er seine neue Systemzeit T_{neu} nach der Formel (9) bestimmen. Dabei wird die Laufzeit der Response bzw. die Zeit, die seit dem Erstellen des Zeitstempels auf dem Server vergangen ist, mit Hilfe der RTT abgeschätzt.

$$T_{neu} = t_s + RTT/2 \tag{9}$$

Anhand des beschriebenen Ablaufs und der mathematischen Beschreibung lassen sich bereits die Gründe für die begrenzte Synchronisationsgenauigkeit von Cristian's Algorithmus erkennen. Zum einen wird angenommen, dass die Latenz des Requests und der Response gleich sind. Darüber hinaus wird angenommen, dass die Bearbeitungszeit auf dem Server bis zum Erstellen des Zeitstempels gleich der Zeitspanne vom Erstellen des Zeitstempels bis zum tatsächlichen Senden der Response ist. Diese Annahmen entsprechen in der Regel nicht den realen Gegebenheiten und führen so zu einer ungenauen Synchronisation der Systemzeiten. Formel (10) beschreibt den Synchronisationsfehler T_{err} mathematisch, wobei t_{min} die minimale Übertragungszeit für eine CoAP-Nachricht darstellt.

$$T_{err} = \pm(RTT/2 - t_{min}) \tag{10}$$

Allerdings lässt sich Cristian's Algorithmus leicht umsetzen und erfordert bei seiner Ausführung nur wenige Systemressourcen. Daher soll er für eine erste prototypische Untersuchung des Konzeptes genügen. Zudem lässt der Aufbau der SYN-Option ein einfaches Hinzufügen anderer Synchronisationsverfahren zu. Nachfolgend wird, aufbauend auf der beschriebenen Zeitsynchronisation, ein Konzept für den TDMA-basierten Netzwerkzugriff erläutert.

6.2 TDMA-basierter Kanalzugriff

In diesem Abschnitt wird ein CoAP-basierter TDMA-Ansatz zur Eliminierung der auf den unteren Schichten des ISO/OSI-Modells entstehenden zeitlichen Unsicherheiten diskutiert. Das vorgestellte Konzept unterteilt sich dabei in zwei Phasen, die Initialisierungs- und die Kommunikationsphase.

In der ersten Phase wird zunächst ein initialer Masterknoten bestimmt, welcher im weiteren Verlauf auch als Zeitserver dienen soll. Der Masterknoten verwaltet darüber hinaus die Zeitschlitze für den Netzwerkzugriff und deren Zuordnung zu den einzelnen Geräten im Netzwerk. Die Rolle des Masters kann prinzipiell von jedem CoAP-Server übernommen werden. In der Praxis soll er, wie in Kapitel 6.1 beschrieben, entweder im Vorfeld oder im laufenden Betrieb durch eine Wettlaufstrategie bestimmt

werden. Im zweiten Schritt findet eine Synchronisation der Systemzeiten zwischen den einzelnen Netzwerkteilnehmern und dem Master nach dem in Kapitel 6.1 beschriebenen Konzept statt.

Nachdem eine einheitliche Zeitbasis im Netzwerk hergestellt wurde, ruft jedes Gerät einen Zeitschlitz von dem Masterknoten ab. Üblicherweise wird in TDMA-basierten Ansätzen jedem Gerät ein Zeitschlitz exklusiv zugewiesen, ohne dabei die tatsächlichen Anforderungen der einzelnen Geräte zu berücksichtigen. Typische Automatisierungssysteme setzen sich aus einer Vielzahl heterogener Einzelgeräte zusammen, welche wiederum verschiedenste Anwendungen ausführen. Dabei können die zeitlichen Anforderungen der Geräte an die Kommunikation je nach Anwendung stark variieren. So müssen einige Geräte beispielsweise alle 50 ms auf das Netzwerk zugreifen, während andere nur alle 250 ms Daten versenden. So große Unterschiede in den Kommunikationsanforderungen sind z.B. im Umfeld der Medizintechnik keine Seltenheit [82]. Die Anforderung, in einem festgelegten Zeitintervall mindestens einmal auf das Netzwerk zugreifen zu dürfen, wird im weiteren Verlauf als Kommunikationsperiode bezeichnet. Die Cycle Time[5] bestimmt sich dabei immer aus der kleinsten Kommunikationsperiode, also dem Gerät mit den höchsten zeitlichen Anforderungen. In der Konsequenz lässt sich in TDMA-basierten Systemen häufig eine niedrige Netzwerkauslastung beobachten. Um ein Brachliegen wertvoller Netzwerkressourcen zu verhindern, soll das im Folgenden beschriebene Konzept eine geteilte Nutzung eines Zeitschlitzes durch mehrere Geräte mit den gleichen oder ähnlichen Anforderungen ermöglichen. Dabei nutzen die Geräte denselben Zeitschlitz jeweils in jedem Zyklus abwechselnd. So können sich n Netzwerkteilnehmer einen einzigen Zeitschlitz teilen, wenn die Bedingung aus Formel (11) erfüllt ist. Dabei bezeichnet t_{period} die von den Geräten geforderte Kommunikationsperiode und t_{cycle} die Cycle Time.

$$t_{period} \geq n * t_{cycle} \tag{11}$$

Die alternierende Nutzung von Zeitschlitzen durch eine Teilnehmergruppe kann dabei zum einen die Netzwerkauslastung in den einzelnen Zeitschlitzen erhöhen. Zum anderen kann somit auch die maximale Anzahl von Geräten im Netzwerk gesteigert werden. Abbildung 43 verdeutlicht die sich ergebenden Vorteile an einem einfachen Beispielszenario für ein Netzwerk mit einer Zykluszeit von 50 ms, fünf Zeitschlitzen T1 bis T5 und fünf Netzwerkteilnehmern N1 bis N5. Dabei ist es für die Knoten N1 und N2 ausreichend, mindestens alle 100 ms auf das Netzwerk zugreifen zu können. Die Knoten N3 bis N4 benötigen hingegen alle 50 ms die Möglichkeit, auf das Netzwerk zuzugreifen. Es wird deutlich, dass sich in diesem Szenario ohne die geteilte Nutzung von Zeitschlitzen nur eine maximale Netzwerkauslastung von 80% erreichen lässt, da die Zeitschlitze T1 und T2 in jedem zweiten Zyklus ungenutzt bleiben. Erlaubt man nun die geteilte Nutzung von Zeitschlitzen, sodass N1 und N2 denselben Zeitschlitz T1 abwechselnd in jeweils aufeinander folgenden Zyklen nutzen, scheint die Netzwerkauslastung zunächst gleich zu bleiben. Allerdings bleibt T5 auf diese Weise vollständig

[5] das Zeitintervall in dem jeder Netzwerkteilnehmer mindestens einmal auf das Netzwerk zugreifen darf.

ungenutzt. Es ließe sich also ein weiteres Gerät zu dem Netzwerk hinzufügen, ohne die Zykluszeit oder die Länge der Zeitschlitze anzupassen. Dies wäre bei einer exklusiven Zuteilung der Zeitschlitze wie in Abbildung 43 a) nicht möglich.

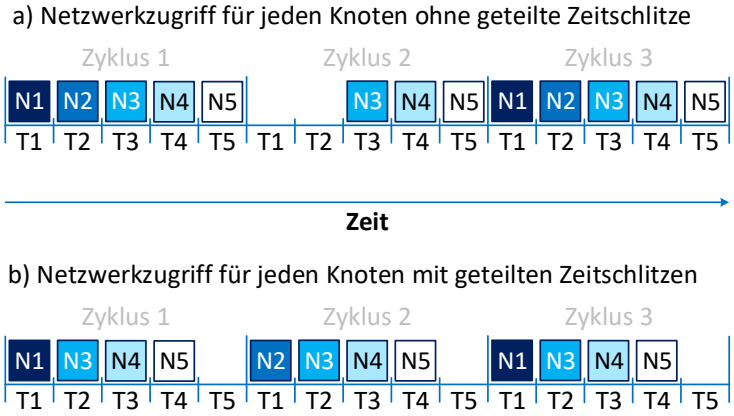

Abbildung 43: Ausnutzung der Zykluszeit a) ohne geteilte Nutzung von Zeitschlitzen und b) mit geteilter Nutzung von Zeitschlitzen.

Für das Anfordern eines Zeitschlitzes wird eine neue Ressource mit der URI *„/well-known/timeslot“* hinzugefügt. Um einen Zeitschlitz anzufragen, sendet ein Client einen POST-Request für diese Ressource an den Zeitserver. An diesen Request hängt der Client seine benötigte Kommunikationsperiode als Nutzlast an. Der Server liest diesen Wert aus und prüft, ob bereits ein Zeitschlitz mit der gleichen Kommunikationsperiode existiert. Ist dies der Fall, kann dem anfragenden CoAP-Knoten dieser Zeitschlitz zugeordnet werden, solange die Kapazität K des Zeitschlitzes noch nicht erreicht ist. K berechnet sich dabei nach Formel (12), um der Bedingung aus Formel (11) gerecht zu werden.

$$K = t_{period} / t_{cycle} \tag{12}$$

Ist die Kapazität des Zeitschlitzes noch nicht erreicht und kann der Knoten diesem zugeordnet werden, so antwortet der Server mit einer Reihe von Informationen, die es dem anfragenden Knoten erlauben, seine Netzwerkzugriffszeit selbstständig zu berechnen. Zu diesen Informationen zählen die Nummer des dem Knoten zugeordneten Zeitschlitzes n_{slot}, die Länge eines Zeitschlitzes t_{slot} und eines Zyklus t_{cycle} sowie die Kapazität K des Zeitschlitzes. Darüber hinaus enthält die Antwort den sogenannten Cycle Offset O_{cycle}. Dieser gibt an, in welchem Zyklus innerhalb der Kommunikationsperiode des Zeitschlitzes der Knoten exklusiv auf das Kommunikationsmedium zugreifen kann. Ist die Kapazität des Zeitschlitzes bereits erreicht oder existiert kein Zeitschlitz mit der gleichen Kommunikationsperiode, so wird der anfragende Knoten zu einem bisher ungenutzten Zeitschlitz hinzugefügt. In diesem Fall ist der Cycle Offset in der Antwort des Servers Null. Auf Basis der in der Serverantwort enthal-

tenen Informationen kann der anfragende CoAP-Knoten über die Formeln (13), (14), (15) und (16) die Anfangszeit seines Zeitschlitzes t_{start} berechnen.

$$C_c = t_{now} / t_{cycle} \tag{13}$$

$$O_c = C_c \mod K \tag{14}$$

$$C_{dist} = \begin{cases} O_{cycle} + K - O_c, & \text{if } O_c \geq O_{cycle} \\ O_{cycle} - O_c, & \text{else} \end{cases} \tag{15}$$

$$t_{start} = (C_c + C_{dist}) * t_{cycle} + t_{slot} * n_{slot} \tag{16}$$

Hierbei bezeichnet C_c die Nummer des aktuellen Zyklus und t_{now} die aktuelle Zeit. Darauf aufbauend gibt O_c den Versatz des aktuellen Zyklus innerhalb der Kommunikationsperiode an. Aus O_c lässt sich die Distanz C_{dist} berechnen, welche die Anzahl von Zyklen angibt, die abgewartet werden muss, bis der Zeitschlitz dem CoAP-Knoten wieder zur Verfügung steht. Mit der sich daraus ergebenden Startzeit des Zeitschlitzes t_{start} und der Zeitschlitzlänge t_{slot} ist dem Knoten der genaue Zeitraum bekannt, in dem er exklusiv auf das Netzwerk zugreifen kann. In diesem Zeitraum kann ein Netzwerkteilnehmer eine beliebige Anzahl von Nachrichten versenden. Allerdings ist hierbei zu beachten, dass der Sendevorgang bis zum Start des nächsten Zeitschlitzes abgeschlossen sein muss. Dies spielt besonders im Rahmen der praktischen Umsetzung des Prototypen eine bedeutende Rolle. Kann einem Netzwerkteilnehmer kein Zeitschlitz zugewiesen werden, so muss der Zeitserver mit einer entsprechenden Fehlermeldung antworten. In diesem Fall kann das Gerät nicht an der Kommunikation im Netzwerk teilnehmen. Mit dem Abrufen der Informationen zur Berechnung des eigenen Zeitschlitzes endet die Initialisierungsphase.

In der sich daran anschließenden Kommunikationsphase greifen alle Netzwerkteilnehmer ausschließlich in dem ihnen jeweils zugewiesenen Zeitschlitz auf das Netzwerk zu. Darüber hinaus wird eine regelmäßige Re-Synchronisation der Systemzeiten durchgeführt. Dies ist nötig, um das Auseinanderlaufen der Uhren der einzelnen Knoten zu kompensieren. Die Ursachen für diesen Effekt, der auch als Clock Drift bezeichnet wird, können vielfältig sein. Die Laufgeschwindigkeit der System Clock wird durch den Takt eines Quarzkristalls vorgegeben. Die Periode der Schwingung dieses Kristalls kann dabei je nach Fertigungsgüte über mehrere Geräte hinweg schwanken. Außerdem kann die Periode der Schwingung auf Grund von Umgebungseinflüßen, wie beispielsweise der Temperatur, innerhalb eines Gerätes schwanken [26]. Diese Effekte führen zu unterschiedlichen Laufgeschwindigkeiten der System Clock auf verschiedenen Geräten. Die regelmäßige Re-Synchronisation wirkt diesem Problem entgegen. Dabei ist die Wahl des Synchronisationsintervalls von entscheidender Bedeutung. Es sollte in Abhängigkeit der vom Hersteller angegebenen maximalen Schwankung der Schwingungsperiode des Quarzes und der maximal zulässigen Zeitabweichung zwischen den Geräten gewählt werden. Dabei ist die maximal zulässige Zeitabweichung von den zeitlichen Anforderungen in dem

jeweiligen Einsatzszenario abhängig. Das nötige Synchronisationsintervall bestimmt jedes Gerät für sich selbst unter Berücksichtigung dieser Parameter. Zur Re-Synchronisation sendet jedes Gerät eigenständig Synchronisationsnachrichten[6] innerhalb seines eigenen Zeitschlitzes an den Zeitserver. Von festgelegten Wartungszyklen zur Re-Synchronisation soll abgesehen werden, da hier der Nutzdatenaustausch im gesamten Netzwerk zeitgleich zum Zweck der Synchronisation unterbrochen wird. Weiterhin muss sich die Häufigkeit von Wartezyklen nach dem Gerät mit dem größten möglichen Clock Drift richten, sodass andere Geräte die Zeitsynchronisation häufiger als nötig vornehmen.

Die Zuordnung der Zeitschlitze gilt über die gesamte Lebensdauer eines Knotens. Die Lebensdauer eines Netzwerkteilnehmers endet, wenn er das Netzwerk, geplant oder auf Grund eines Fehlers, verlässt. In diesem Fall wird die Zuordnung aufgehoben und der Zeitschlitz ist wieder verfügbar. Verlässt ein Knoten gewollt das Netzwerk, so kann er dies dem Zeitserver signalisieren. Hierzu sendet er einen POST-Request an die *„/well-known/timeslot“* - Ressource des Zeitservers mit einem negativen Wert für die gewünschte Kommunikationsperiode. Fällt ein Knoten unerwartet aus und ist nicht mehr in der Lage, dem Zeitserver sein Verschwinden zu signalisieren, so bleibt die Zeitschlitzzuordnung zunächst bestehen. Bleiben die Synchronisationsnachrichten eines Netzwerkteilnehmers aus, so nimmt der Zeitserver an, dass der Knoten nicht länger an der Kommunikation teilnimmt. Daraufhin löst er die Zuordnung des Zeitschlitzes auf und weist diesen gegebenenfalls einem anderen Knoten zu. Der Zeitraum, nachdem die Zuordnung gelöscht wird, kann entweder global im Netzwerk festgelegt werden oder sich nach dem vergangenen Synchronisationsverhalten des jeweiligen Clients richten. Hierzu kann der Server die Zeit der letzten Synchronisation zwischenspeichern und über mehrere Synchronisationszyklen hinweg einen Mittelwert des Synchronisationsintervalls bilden. Diese Variante ist zu bevorzugen, da es sich bei der zu erwartenden Clock Drift um einen gerätespezifischen Wert handelt und in einem realen Einsatzszenario eine Vielzahl verschiedenartiger Geräte mit anderen Kennwerten zu erwarten ist. Tritt ein Knoten nach einem Ausfall wieder dem Netzwerk bei, muss er einen neuen Zeitschlitz abrufen.

6.3 Umsetzung eines Prototypen

Das in den Kapiteln 6.1 und 6.2 beschriebene Verhalten wurde prototypisch mit jCoAP umgesetzt. Ziel war es dabei, festzustellen, inwieweit diese Mechanismen dazu geeignet sind, ein zeitlich deterministisches Kommunikationsverhalten herbeizuführen. Bei der Umsetzung sind dabei einige jCoAP- und Java-spezifischen Besonderheiten zu beachten.

Wie bereits in Kapitel 4.1 erläutert, verwendet jCoAP die SocketHandler-Klasse, um die offenen Ports und den eingehenden beziehungsweise ausgehenden Datenverkehr zu verwalten. In dem SocketHandler werden hierfür zwei Threads ausgeführt, ein Sende- und ein Empfangs-Thread. Um sicherzustellen, dass der CoAP-Knoten aussschließlich in seinem Zeitschlitz auf das Kommunikationsmedium

[6] GET-Requests für die *„/well-known/time“*-Ressource.

zugreift, wird der Socket, über den die Daten gesendet werden, mit einem Semaphor blockiert. Das Semaphor wird dabei über zwei hoch priorisierte asynchrone Event Handler zu Beginn des Zeitschlitzes freigegeben und zum Ende wieder blockiert. Die Event Handler werden durch zwei periodische Timer, $T_{release}$ und T_{lock}, gesteuert. Die Timer werden beide mit der Kommunikationsperiode des Knotens t_{period} initialisiert, unterscheiden sich aber in ihrer Startzeit. Die Startzeit $t_{TRStart}$ von $T_{release}$ wird über die Formel (16) aus Kapitel 6.2 berechnet. Die Startzeit $t_{TLStart}$ von T_{lock} ergibt sich entsprechend aus Formel (17), wobei t_{slot} die Länge eines Zeitschlitzes darstellt.

$$t_{TLStart} = t_{TRStart} + t_{slot} \tag{17}$$

Bevor der Sende-Thread auf das Netzwerk zugreifen kann, muss er selbst das Semaphor blockieren. Liegt die aktuelle Systemzeit außerhalb des eigenen Zeitschlitzes, ist das Semaphor bereits blockiert. Dies hat zur Folge, dass der Sende-Thread solange blockiert wird, bis das Semaphor durch den entsprechenden Event Handler wieder freigegeben wird. In den meisten Anwendungsfällen führt dies zu dem gewünschten Verhalten, da die beiden Event Handler eine höhere Priorität haben und den Sende-Thread somit jederzeit unterbrechen können. Allerdings kann aufgrund einiger in der RTSJ getroffener Festlegungen dennoch fehlerhaftes Verhalten auftreten. So führt die RTSJ die sogenannte Priority Inheritance (engl. für Prioritätenvererbung) für Code-Bereiche ein, die jeweils nur von einem Thread zur Zeit ausgeführt werden sollen. Hierbei erbt ein Thread mit niedriger Priorität, der sich bereits in dem entsprechenden Code-Block befindet, die höchste Priorität aller Threads, die ebenfalls auf das Betreten des kritischen Bereiches warten. Auf diese Weise kann der niederpriore Thread in kritischen Code-Bereichen nicht durch hochpriore Threads unterbrochen werden. Ziel dieses Mechanismus ist es, fehlerhafte Datenzustände zu vermeiden, die sich im Falle eines Interrupts ergeben können. Weiterhin kann so auch der Effekt einer sich indirekt fortsetzenden Prioritäteninversion reduziert werden, die das Zeitverhalten der gesamten Anwendung stark negativ beeinflussen kann [24]. Auf den hier dargestellten Anwendungsfall bezogen bedeutet dies, dass der Sende-Thread nicht mehr von den Event Handlern unterbrochen werden kann, wenn er sich bereits in dem kritischen Code-Bereich befindet. Dementsprechend überprüft der Sende-Thread direkt vor dem Senden noch einmal selbstständig, ob das Ende des Zeitschlitzes bereits erreicht ist.

Eine weitere Besonderheit bei der Umsetzung des beschriebenen Konzeptes ist das Einfügen der Synchronisationsnachrichten[7] am Anfang der Sende-Queue. Auf diese Weise werden die Nachrichten zur Zeitsynchronisation immer vorrangig zu anderen Nachrichten innerhalb des nächsten Zeitschlitzes versendet. Dies erlaubt es, die Startzeit des nächsten Zeitschlitzes als die Sendezeit[8] der Nachricht anzunehmen. Daraus ergeben sich genauere Werte für die RTT-Messung, als würde der Zeitpunkt der Übergabe der Nachricht an den Sende-Thread als Sendezeitpunkt angenommen. Eine genauere

[7] GET-Requests für die *„/well-known/time“*-Ressource.
[8] Startzeit der RTT-Messung.

Messung der RTT wirkt sich entsprechend der Formel (10) aus Kapitel 6.1 positiv auf die Genauigkeit der Zeitsynchronisation aus.

6.4 Performance-Analyse

Um die Wirksamkeit der beschriebenen Mechanismen zu untersuchen, wurden drei Untersuchungen auf Basis des mit jCoAP umgesetzten Prototypen duchgeführt. Hierbei wurde der in Abbildung 44 dargestellte Testaufbau genutzt, welcher im Vergleich zu dem in Kapitel 4.3 beschriebenen Aufbau mit einem zweiten Client erweitert wurde. Als Testplattform kam weiterhin das Intel Galileo Board zum Einsatz. Der Server diente hierbei auch als Zeitserver. Alle Geräte nutzten einen voll präemptiven Linux Kernel mit der Kernel-Version 3.8. Als echtzeitfähige JVM wurde, wie auch in den vorangegangen Experimenten, die JamaicaVM der Aicas GmbH verwendet.

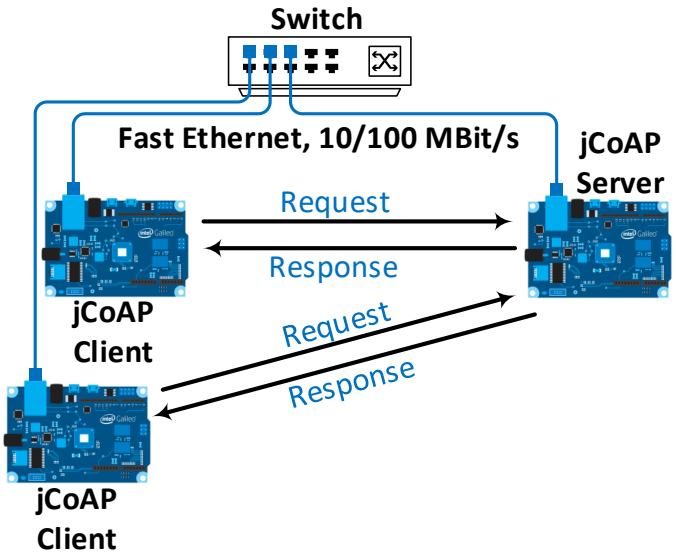

Abbildung 44: Aufbau der für die Messungen verwendeten Testumgebung.

Im ersten Testszenario wurde zunächst nur die Machbarkeit des Software-gesteuerten Netzwerkzugriffs untersucht. Hierzu kam lediglich einer der beiden Clients zum Einsatz. Der Client führte die initiale Zeitsynchronisation durch und rief einen Zeitschlitz ab. In dem Szenario wurde eine Zykluszeit von 100 ms genutzt, welche in zehn Zeitschlitze à 10 ms aufgeteilt wurde. Der Client hatte dabei eine Kommunikationsperiode von 500 ms. Diese Parameter wurden in Anlehnung an die in [82] dargestellten Anforderungen im Medizinbereich gewählt. Als Synchronisationsintervall wurden 30 s gewählt. Dies ergibt sich aus der maximalen Clock Drift des Galileo Boards und einer gewünschten Genauigkeit von 1 ms. Aus dem Datenblatt des auf dem Galileo Board verbauten Quarz-Kristalls lässt sich eine maximale Taktabweichung von ± 20 *ppm* entnehmen [53, 98]. Die Einheit ppm steht hierbei für „Parts per Million“ und gibt die Taktabweichung in Sekunden für eine Laufzeit von einer

Million Sekunden an. Nach einer Laufzeit von einer Million Sekunden muss dementsprechend mit einer Abweichung der Systemzeit von $\pm 20\ s$ gerechnet werden. Es ist zu beachten, dass sowohl der Zeitserver als auch die Clients von dem Phänomen des Clock Drifts betroffen sind. Hierbei muss berücksichtigt werden, dass die Systemzeiten der einzelnen Geräte in unterschiedliche Richtungen driften, also zu schnell oder zu langsam laufen können. Daher muss für die Berechnung des Synchronisationsintervalls der doppelte Wert für die maximale Taktabweichung zugrunde gelegt werden. Nach der Initialisierungsphase wurde die Client-Anwendung gestartet. Diese rief den 16 Byte großen Wert einer *„/echo“*-Ressource von dem Server mit einem GET-Request ab. Dieser Vorgang wurde 50.000 mal wiederholt, wobei für jeden Request die Sendezeit auf dem Client sowie die Empfangszeit auf dem Server mit einer Genauigkeit von 1 ms aufgezeichnet und in einer Datei gespeichert wurde. Abbildung 45 stellt die aufgezeichneten Zeitstempel grafisch dar. Die x-Achse gibt die Gesamtzeit des Versuchs in Minuten an. Die y-Achse zeigt hingegen die relative Zeit innerhalb eines Kommunikationszyklus in Millisekunden, zu der ein Request von dem Client gesendet (a) oder vom Server empfangen wurde (b). Die rote Linie markiert hierbei die 10-ms-Grenze des an den Client vergebenen Zeitschlitze *Slot 1* innerhalb des Zyklus. Die blauen Punkte repräsentieren die aufgezeichneten Zeitstempel.

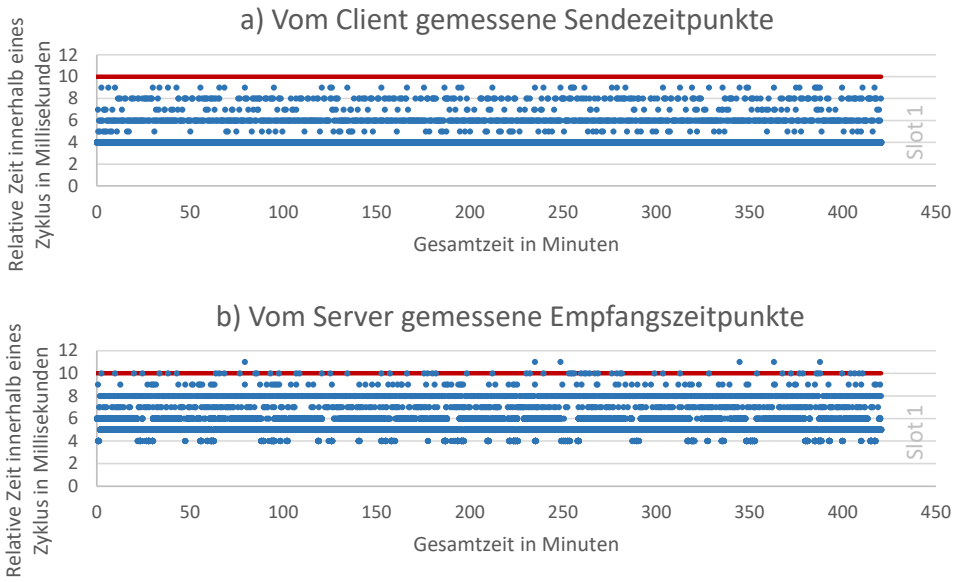

Abbildung 45: Darstellung der von dem Client und dem Server gemessenen Sende- (a) und Empfangszeitpunkte (b) der einzelnen Nachrichten.

Aus Abbildung 45 a) wird deutlich, dass der Client jeweils nur innerhalb des ihm zugewiesenen Zeitschlitzes auf das Netzwerk zugreift, da die Sendezeit der Nachrichten stets vor der Zeitschlitzgrenze liegt. Zudem lässt sich erkennen, dass der Großteil der Requests auch innerhalb des Zeitschlitzes von dem Server empfangen werden, sodass der Kommunikationsvorgang noch innerhalb des Zeitschlitzes abgeschlossen wird. Darüber hinaus verdeutlicht die Verteilung der gemessenen Zeiten innerhalb

des Zeitschlitzes, dass die vom Server gemessene Empfangszeit nach der vom Client aufgezeichneten Sendezeit liegt. Daraus lässt sich schlussfolgern, dass die Zeitsynchronisation zwischen den Geräten mit hinreichender Genauigkeit erfolgt ist. Allerdings zeigen die Ergebnisse auch, dass nicht alle Nachrichten innerhalb des Zeitschlitzes von dem Server empfangen werden. Dies ist problematisch, da so die Kommunikation der Knoten innerhalb der anderen Zeitschlitze gestört werden kann. Die Ursache für dieses Zeitverhalten liegt in der Tatsache begründet, dass der Client den vollständigen Zeitschlitz ausnutzt, um Nachrichten zu senden, ohne die Übertragungszeit zu berücksichtigen. Daher wurde in den nachfolgenden Experimenten ein Zeitpuffer von 2 ms am Ende jedes Zeitschlitzes reserviert, sodass dem Client effektiv noch 8 ms blieben, um Nachrichten zu versenden. Die Länge des Zeitpuffers ergibt sich aus der Genauigkeit der Systemzeit von 1 ms und der Übertragungszeit eines Ethernet-Frames über 100 Mbit/s Fast Ethernet von 121 μs[9].

Im zweiten Testszenario kamen zwei Clients und ein Zeitserver zum Einsatz. Der zweite Client hatte eine gewünschte Kommunikationsperiode von 200 ms. Die übrigen Kenndaten wurden aus dem ersten Testszenario übernommen. Beide Clients führten die gleiche Anwendung aus. Abbildung 46 zeigt die vom Server gemessenen Empfangszeiten für die Requests beider Clients. Auch hier zeigt die x-Achse die Gesamtlaufzeit des Experiments in Minuten an, während die y-Achse die relative Zeit innerhalb eines Kommunikationszyklus abbildet. Die roten Linien markieren die Grenzen der Zeitschlitze innerhalb des Zyklus. Die dunkelblauen Punkte stellen die Empfangszeiten der Nachrichten des ersten Clients mit einer Kommunikationsperiode von 500 ms dar. Die hellblauen Punkte zeigen die Empfangszeiten der Nachrichten des zweiten Clients mit einer Kommunikationsperiode von 200 ms.

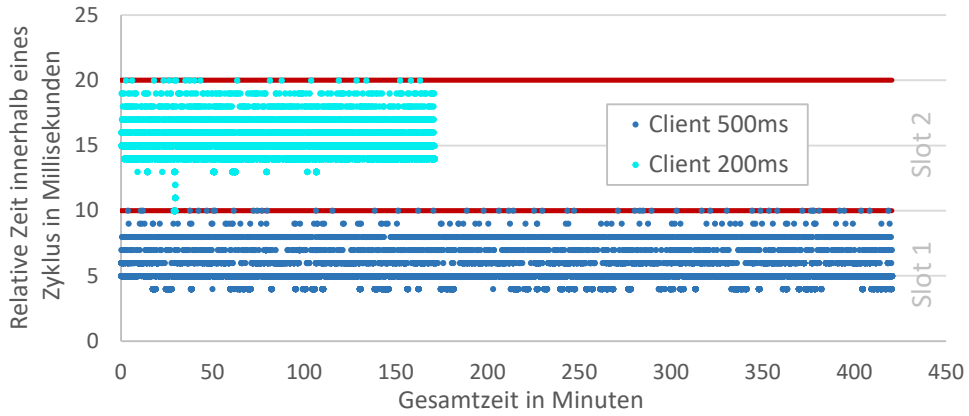

Abbildung 46: Darstellung der vom Server gemessenen Empfangszeitpunkte der einzelnen Nachrichten von zwei Clients.

In den Ergebnissen spiegelt sich das erwartete Verhalten wieder. Den beiden Clients wurden unterschiedliche Zeitschlitze zugewiesen, da sie unterschiedliche Zeitanforderungen an das Netzwerk

[9]Dieser Wert bezieht sich nur auf eine Punkt-zu-Punkt-Verbindung, ohne Switching-Delays. Bei großen Netzwerken mit vielen Infrastrukturelementen müssen gegebenenfalls größere Zeitpuffer eingeplant werden

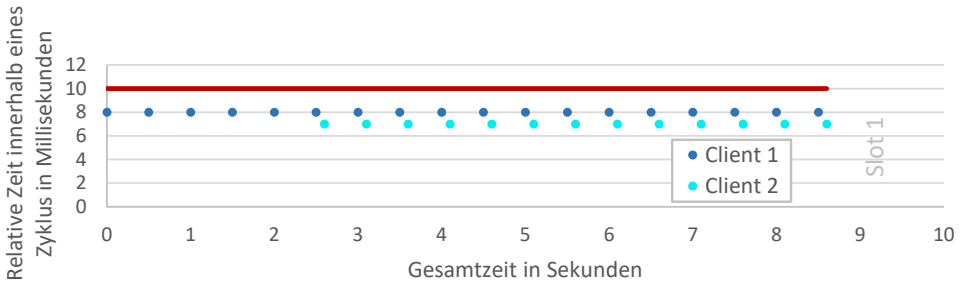

Abbildung 47: Darstellung der vom Server gemessenen Empfangszeitpunkte der einzelnen Nachrichten von zwei Clients mit gemeinsam genutztem Zeitschlitz.

stellen. Der erste Client nutzt den ersten Zeitschlitz, während der zweite Client im zweiten Zeitschlitz arbeitet. Es lässt sich außerdem erkennen, dass der Server die Nachrichten der Clients ausschließlich in den ihnen jeweils zugewiesenen Zeitschlitzen erhält. Die im ersten Testszenario beobachteten Überschreitungen der Zeitschranken bleiben, wie erwartet, aus, da die Übertragungszeit der Nachrichten durch den eingeführten Zeitpuffer mit berücksichtigt wird. Es kann zudem beobachtet werden, dass der zweite Client nach deutlich kürzerer Zeit die Kommunikation mit dem Server einstellt. Dies kann mit der kürzeren Kommunikationsperiode und der in der Clientanwendung festgelegten Nachrichtenanzahl von 50.000 begründet werden. Weiterhin ist zu sehen, dass die Nachrichten des zweiten Clients über eine Synchronisationsperiode hinweg zu einem sehr frühen Zeitpunkt innerhalb des Zeitschlitzes bei dem Server eintreffen. Dies lässt die Vermutung zu, dass die Sendezeit der Nachricht noch außerhalb des Zeitschlitzes liegt. Dies ist auf den verwendeten Synchronisationsalgorithmus zurückzuführen, da *Cristians Algorithmus* nur eine sehr begrenzte Genauigkeit besitzt. Das präsentierte Konzept erlaubt daher die Erweiterung mit genaueren Synchronisationsverfahren. Im nachfolgenden Kapitel soll dieser Punkt weiter aufgegriffen werden.

Im dritten Testszenario wurde das Verhalten bei geteilten Zeitschlitzen untersucht. Hierzu arbeiteten beide Clients mit der gleichen Kommunikationsperiode. In der Folge sollte beiden Clients der gleiche Zeitschlitz zugewiesen werden, wobei sie ihn in abwechselnden Zyklen nutzen. Wie auch in den vorangegangenen Versuchen zeichnete der Server die Empfangszeiten der Nachrichten beider Clients auf. Abbildung 47 zeigt nur einen kleinen Ausschnitt der Ergebnisse, da das Zeitverhalten auf Grund der kurzen Zykluszeit in der graphischen Darstellung aller Ergebnisse nicht sichtbar wäre. Anders als in den anderen Darstellungen, bildet die x-Achse diesmal die Gesamtdauer des Experiments in Sekunden ab. Die y-Achse gibt wieder die relative Zeit innerhalb eines Kommunikationszyklus in Millisekunden an.

Es zeigt sich, dass dem ersten Client auch der erste Zeitschlitz zugewiesen wird. Weiterhin ist sichtbar, dass der zweite Client nach ca. 2,5 s dem Netzwerk beitritt. Dabei wird auch dem zweiten Client der erste Zeitschlitz zugewiesen, allerdings mit einem Cycle Offset von 1. Die gemessenen Zeitstempel zeigen deutlich, dass die beiden Clients den ersten Zeitschlitz abwechselnd nutzen. Auf diese Weise

treten sie nicht miteinander in Konkurrenz, vielmehr kann jeder Client das Medium exklusiv nutzen. Der leichte Versatz der Empfangszeiten innerhalb des Zeitschlitzes zwischen den Requests von Client 1 und Client 2 kann mit dem verwendeten Synchronisationsverfahren begründet werden. *Cristians Algorithmus* kann zu einer Abweichung der Systemzeiten der beiden Clients führen, sodass beide jeweils zu unterschiedlichen Zeiten innerhalb des Zeitschlitzes mit dem Sendevorgang beginnen. Das beobachtete entspricht insgesamt dem erwarteten, im Konzept beschriebenen Verhalten.

6.5 Zwischenfazit

In diesem Kapitel wurde ein zentralisierter Ansatz zur Zeitsynchronisation zwischen CoAP-Geräten und Verwaltung von Zeitschlitzen beschrieben. Ziel war es, einen rein Software-gesteuerten TDMA-basierten Netzwerkzugriff zu ermöglichen. Auf diese Weise sollten die zeitlichen Unsicherheiten, die aus der Nutzung von Standard-Ethernet resultieren, eliminiert und somit eine zeitlich deterministische Kommunikation ermöglicht werden. Hierzu wurde CoAP um die SYN-Option erweitert, die einen Synchronisationsvorgang gegenüber dem Zeitserver anzeigen soll und eine flexible Aushandlung des verwendeten Synchronisationsverfahrens ermöglicht. Weiterhin wurde ein Mechanismus zur Verteilung von Zeitfenstern für den exklusiven Medienzugriff erörtert. Das vorgestellte Verfahren bezieht dabei die zeitlichen Anforderungen der Geräte an die Kommunikation mit ein. Das Wissen über das tatsächliche Kommunikationsverhalten der Geräte wird genutzt, um mehrere Geräte zu gruppieren und einem einzelnen Zeitschlitz zuzuordnen. Zeitschlitze, die einer Gruppe zugeordnet sind, werden von den Geräten in abwechselnder Reihenfolge exklusiv genutzt. Auf diese Weise lässt sich eine höhere Auslastung der zur Verfügung stehenden Netzwerkressourcen mit einer gleichzeitigen Erhöhung der maximalen Anzahl von Netzwerkteilnehmern erreichen. Die Evaluation eines mit jCoAP umgesetzten Prototypen hat gezeigt, dass die beschriebenen Mechanismen prinzipiell geeignet sind, ein zeitlich deterministisches Kommunikationsverhalten im Netzwerk zu erzielen. Es konnte weiterhin gezeigt werden, dass eine Software-basierte Kontrolle des Netzwerkzugriffs effektiv anwendbar ist. Jedoch bildet der zentralisierte Zeitserver einen SPoF. Hieraus ergeben sich gravierende Probleme hinsichtlich der Skalierbarkeit und der Robustheit des beschriebenen Ansatzes. Mit einem Ausfall des Zeitservers würde die Übersicht über die Zuordnung der Netzwerkteilnehmer zu den Zeitschlitzen verloren gehen. Eine gezielte Zuordnung neuer Netzwerkteilnehmer wäre so selbst bei der Bestimmung eines neuen Zeitservers nicht mehr möglich. Darüber hinaus würde die gemeinsame Zeitbasis im Netzwerk zerstört, da eine Synchronisation der Geräte bis zur Auswahl eines neuen Zeitservers nicht mehr möglich wäre. Weiterhin könnte eine steigende Anzahl von Netzwerkgeräten zu einer Überlastung des Zeitservers führen, sodass dieser nicht mehr in der Lage ist, die Anfragen zur Zeitsynchronisation in hinreichender Geschwindigkeit zu verarbeiten. Um diesen Problemen entgegenzuwirken, bedarf es einer Dezentralisierung der beschriebenen Ansätze. Diese Dezentralisierung ist das zentrale Thema des nachfolgenden Kapitels.

7 Dezentralisierung des CoAP-Zeitservers

Für die Dezentralisierung des im vorangegangenen Kapitel beschriebenen Zeitservers können unterschiedliche Ansätze verfolgt werden. So können zum Beispiel P2P-Technologien, wie HaRTKad, zusätzlich zu CoAP verwendet werden. Allerdings werden so Abhängigkeiten zu anderen Protokollen geschaffen. Diese erhöhen zum einen die Komplexität der Software. Zum anderen führen Abhängigkeiten zwischen zwei Technologien in der Regel dazu, dass die beiden Technologien nicht unabhängig voneinander weiterentwickelt werden können, ohne Kompatibilitätsprobleme zu erzeugen [112]. Dies erschwert die Wartung des Gesamtsystems und führt so zu einer geringeren Akzeptanz von CoAP bei der Verwendung in Echtzeitszenarien. Daher ist es vorzuziehen, auch bei der Dezentralisierung des Zeitservers nur auf CoAP-eigene Mechanismen zurück zu greifen. Nachfolgend wird zunächst das Grundkonzept zur CoAP-basierten Verteilung des Zeitservers erläutert. Dabei werden Anforderungen entwickelt, deren Erfüllung in den nachfolgenden Unterkapiteln beschrieben wird. Darauf folgen Betrachtungen zur Behandlung von auftretenden Fehlern. Abschließend werden die beschriebenen Mechanismen anhand einer prototypischen Umsetzung evaluiert. Die in diesem Kapitel vorgestellten Ergebnisse wurden bereits veröffentlicht [60].

7.1 Grundkonzept

Auch das hier vorgestellte Konzept zur Verteilung der Zeitserver-Funktion auf mehrere CoAP-Server geht zu Beginn wie die zentralisierte Variante von einem initialen Zeitserver aus. Dieser versendet zunächst eine Discovery-Nachricht im Netzwerk mit dem Ziel, weitere potentielle Zeitserver zu finden. Um eine Kommunikation zwischen den Servern zu ermöglichen, müssen die einzelnen Server allerdings auch über Client-Funktionalitäten verfügen. Nur auf diese Weise ist es ihnen möglich, Requests an andere Server zu versenden. Um dies zu realisieren, kann auf zweierlei Weise vorgegangen werden. Zum einen kann ein Client als Parallelanwendung gestartet werden. Dies würde allerdings wieder voraussetzen, dass zwischen dem Client und dem Server eine Interprozesskommunikation stattfindet. Diese Art von Kommunikation lässt sich jedoch nur schwer mit in die Protokoll-Spezifikation von CoAP aufnehmen, da hier eine ganz andere Ebene der Kommunikation betrachtet wird. Dies kann zu starken Unterschieden in der Umsetzung der hierfür nötigen Schnittstellen zwischen den einzelnen CoAP-Implementierungen führen. Ein anderer Ansatz ist es, die Client-Funktionalität direkt in den Server zu integrieren. Dieser Ansatz birgt jedoch die Gefahr, dass die strikte Trennung zwischen Client und Server aufgeweicht wird. Die Trennung von Client und Server ist allerdings ein zentraler Bestandteil in der Definition des REST-Paradigmas. Jedoch gibt es eine Möglichkeit, die Verschmelzung von Client und Server so zu gestalten, dass sie sich mit dem REST-Paradigma vereinbaren lässt. Hierzu ist es hilfreich, sich die Definition einer Ressource im REST-Kontext noch einmal vor Augen zu führen. Eine Ressource ist definiert als jedes Datenobjekt oder

jeder Dienst, der von einem Server nach außen bereitgestellt wird [39]. Ein Dienst kann in diesem Zusammenhang jede Softwarekomponente sein. Unter Berücksichtigung dieser Eigenschaften ist es möglich, die Client-Funktionalität, welche ein CoAP-Server benötigt, um Anfragen an andere CoAP-Server zu versenden, als Dienst zu begreifen. So kann ein CoAP-Client als Ressource unter dem URI *„/.well-known/synclient/“* gestartet werden. Mit Hilfe dieses Clients kann der initiale Zeitserver nun einen GET-Request für die Ressource *„/well-known/core“* an die All-CoAP-Nodes-Multicast-Adresse versenden. Weiterhin wird dem Request der Query-String „titel=/.well-known/time“ hinzugefügt. Auf diese Weise antworten nur Server, welche eine Ressource mit dem festgelegten URI bereitstellen und somit auch die Echtzeitspezifikation von CoAP unterstützen. Nachdem der initiale Zeitserver *i* Antworten erhalten hat oder ein Timeout erreicht wird, bricht er den Discovery-Prozess ab und geht dazu über, die CoAP-Clients auf den zusätzlichen Zeitservern zu starten und zu konfigurieren. Hierzu versendet er an jeden der *i* Zeitserver einen POST-Request an die Ressource *„/.well-known/synclient“*. Dieser Request enthält alle wichtigen Parameter für das Management der Zeitschlitze, wie etwa die Zykluszeit und die Anzahl der Zeitschlitze. Darüber hinaus enthält der Request eine Liste der ausgewählten Zeitserver im Netzwerk.

Bei Empfang dieses Requests starten die einzelnen Zeitserver ihren Synchronisations-Client. Dieser sendet zunächst einen GET-Request mit der Observe-Option für die Ressource *„/.well-known/timeslot“* an alle anderen Zeitserver. Die Observe-Option zeigt dabei an, dass der Client über alle Änderungen des Wertes der Ressource mittels einer Notification informiert werden will. Eine Änderung des Ressourcen-Wertes ist immer dann zu erwarten, wenn ein neues CoAP-Gerät dem Netzwerk beitritt und einen Zeitschlitz abruft oder wenn ein CoAP-Gerät das Netzwerk verlässt. Die Notifications, die der Synchronisations-Client in diesem Fall erhält, enthalten die IP-Adresse des CoAP-Gerätes, die Zeitschlitz-Nummer sowie die Kommunikationsperiode in diesem Zeitschlitz und den Offset des CoAP-Gerätes innerhalb des Zeitschlitzes. Mit diesen Daten kann der Zeitserver seine Sicht auf die Zeitschlitzverteilung aktualisieren. Auf diese Weise hat jeder der Zeitserver immer einen vollständigen Überblick über die Netzwerkteilnehmer, ihre Kommunikationsperioden und den ihnen jeweils zugeordneten Zeitschlitzen.

Im Anschluss fragen die zusätzlich akquirierten Zeitserver über ihren Synchronisations-Client einen Zeitschlitz von dem intialen Zeitserver an. Hierzu versenden sie einen POST-Request für die *„/.well-known/timeslot“*-Ressource des initialen Zeitservers. Der initiale Zeitserver beansprucht dabei immer den ersten Zeitschlitz im Zyklus für sich. Nachdem alle zusätzlichen Zeitserver einen Zeitschlitz erhalten haben, synchronisieren sie ihre Systemzeit mit dem initialen Zeitserver.

Nachdem jedem Zeitserver ein Zeitschlitz zugewiesen wurde und die Systemzeiten der einzelnen Zeitserver synchronisiert wurden, greifen die Zeitserver nur noch in ihrem Zeitschlitz auf das Netzwerk zu. Nach der Initialisierung ist jeder der *i* Zeitserver für alle CoAP-Geräte im *i*-ten Bereich des Kommunikationszyklus verantwortlich. Diese Aufteilung ist in Abbildung 48 beispielhaft für einen Zyklus mit zehn Zeitschlitzen TS1 - TS10 und drei Zeitservern dargestellt.

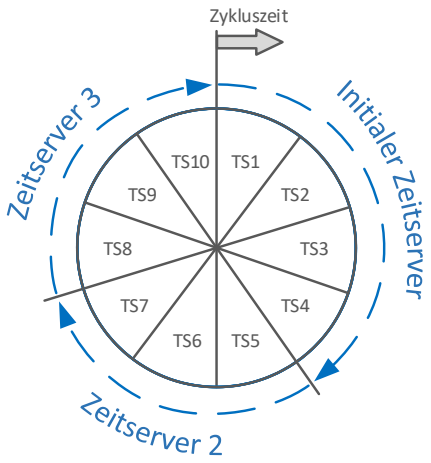

Abbildung 48: Verteilung der Verantwortungsbereiche in einem Szenario mit zehn Zeitschlitzen und drei Zeitservern.

Betritt ein neues CoAP-Gerät das Netzwerk und fordert einen Zeitschlitz bei einem der Zeitserver an, ordnet ihm dieser einen Zeitschlitz entsprechend seiner Sicht auf den Kommunikationszyklus und die zeitlichen Anforderungen des Clients zu. Die Antwort des Zeitservers enthält zusätzlich zu den in Kapitel 6 dargestellten Informationen die IP-Adresse des für das Gerät zuständigen Zeitservers. Anschließend benachrichtigt er alle anderen Zeitserver über den neuen Netzwerkteilnehmer. Alle nachfolgenden Synchronisationsanfragen des neuen Gerätes werden ausschließlich an den ihm zugeteilten Zeitserver versand. Abbildung 49 stellt die hier beschriebenen Abläufe noch einmal für ein einfaches Szenarion mit zwei Zeitservern und einem Client graphisch dar.

7.2 Verbesserte Zeitsynchronisation

Um die Nutzung eines verteilten Zeit-Servers zu ermöglichen, muss die Zeitsynchronisation zwischen den Geräten verbessert werden. In den vorangegangenen Experimenten wurde der Algorithmus von Cristian zur Zeitsynchronisation zwischen den Geräten verwendet, welcher nur eine sehr eingeschränkte Genauigkeit bei der Synchronisation zulässt. So konnten in den Experimenten bereits einige Verstöße gegen das TDMA-Schema auf Grund einer zu ungenauen Zeitsynchronisation beobachtet werden. In dem beschriebenen Konzept für eine Verteilung der Zeitserver-Funktionalität auf mehrere Zeitserver wirkt sich diese Ungenauigkeit noch drastischer aus, da sich hier der Synchronisationsfehler über mehrere Hierarchie-Ebenen fortpflanzt. Die größten Schwächen von *Cristians Algorithmus* liegen darin, dass die Verarbeitungszeit auf dem Server außer Acht gelassen wird und die beiden Kommunikationswege als symmetrisch angenommen werden. Daher wird in diesem Abschnitt eine Verfeinerung von *Cristians Algorithmus* beschrieben, welche diese Punkte aufgreift und

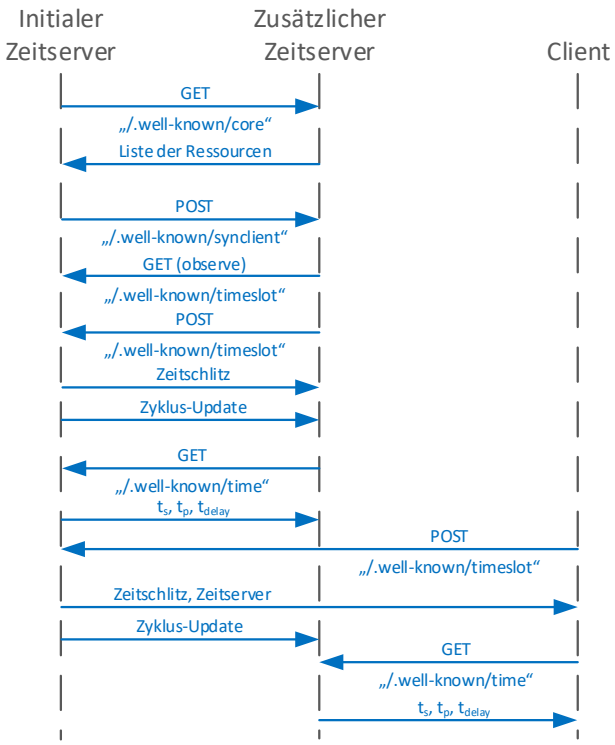

Abbildung 49: Ablauf der Akquise von zusätzlichen Zeitservern und der Verteilung der Clients unter den Zeitservern.

den Synchronisationsfehler deutlich reduziert. Der Ablauf der verfeinerten Zeitsynchronisation ist in Abbildung 50 schematisch dargestellt.

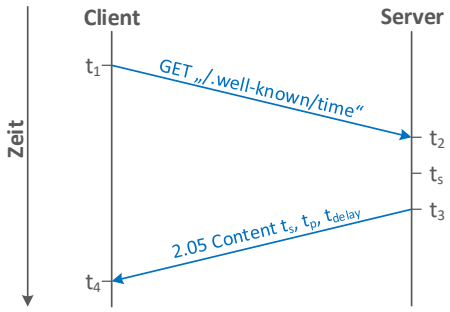

Abbildung 50: Ablauf der verbesserten Zeitsynchronisation.

Wie auch bei *Cristians Algorithmus*, versendet der Client eine Synchronisationsanfrage und speichert die Sendezeit t_1 der Anfrage. Der Server misst neben dem Zeitstempel t_s auch die Empfangszeit der Synchronisationsanfrage t_2. Außerdem wird die Sendezeit t_3 der Antwortnachricht geschätzt. Hierbei wird für die prototypische Umsetzung davon ausgegangen, dass die Antwort im nächsten Zeitschlitz des Zeitservers gesendet wird. Der tatsächliche Sendezeitpunkt kann hiervon abweichen, je nachdem wie viele Nachrichten sich bereits in der Sende-Queue des Servers befinden. Können nicht alle Ant-

worten im nächsten Zeitschlitz des Zeitservers gesendet werden, verschiebt sich der Sendezeitpunkt in den nächsten Zyklus. Die Sendezeit kann näherungsweise aus der Bandbreite der verwendeten Kommunikationstechnologie, der Zeitschlitzlänge und der Größe der Nachrichten berechnet werden. Aus den Zeiten t_2, t_s und t_3 werden die Verarbeitungszeit t_p auf dem Server und die Verzögerung zwischen dem Messen des eigentlichen Zeitstempels und dem Senden der Antwortnachricht berechnet (vgl. Formel (18), (19)).

$$t_p = t_3 - t_2 \tag{18}$$

$$t_{delay} = t_3 - t_s \tag{19}$$

In seiner Antwortnachricht sendet der Zeitserver neben t_s auch die berechneten Zeitspannen t_p und t_{delay}. Der Client speichert, wie auch bei *Cristians Algorithmus*, die Empfangszeit t_4 der Antwort. Anschließend kann der Client über die Formel (20) die neue Systemzeit t_{now} berechnen.

$$t_{now} = t_s + \frac{t_4 - t_1 - t_p}{2} + t_{delay} \tag{20}$$

So werden sowohl die Verarbeitungszeit auf dem Server (t_p) als auch die Sendeverzögerung der Antwort (t_{delay}) bei der Berechnung der neuen Systemzeit miteinbezogen, wodurch sich der Synchronisationsfehler deutlich reduziert. Zwar wird die Latenz bei der Nachrichtenübermittlung immer noch als symmetrisch angenommen, jedoch tritt der hierdurch hervorgerufene Synchronisationsfehler gegenüber der Sendeverzögerung und der Verarbeitungszeit in den Hintergrund, da diese Zeiten ein Vielfaches der Zykluszeit betragen können. Der Synchronisationsfehler t_e ergibt sich nach der Anpassung des Verfahrens aus Formel (21).

$$t_e = \pm \left(\frac{RTT - t_p}{2} - T_{min} \right), \; wobei \; RTT >> (RTT - t_p) \tag{21}$$

Eine weitere Reduktion des Synchronisationsfehlers t_e lässt sich beispielsweise durch den in [85] beschriebenen Ansatz erreichen. Hierbei wird eine Wahrscheinlichkeitsverteilung für die Latenzen zwischen den einzelnen Kommunikationspartnern im Netzwerk ermittelt. Diese Daten können anschließend genutzt werden, um die jeweiligen Latenzen für die Synchronisationsanfrage des Clients und die Antwort des Zeitservers zu gewichten.

7.3 Ausgeglichene Zeitschlitz-Verteilung

Bei der Verteilung der Zeitschlitze an die Netzwerkteilnehmer werden grundsätzlich zwei Ziele verfolgt. Zum einen soll der Kommunikationszyklus effizient ausgenutzt werden, sodass dem Netzwerk

eine möglichst hohe Anzahl von Geräten beitreten kann. Zum anderen soll die Last, welche durch die Synchronisationsvorgänge auf den einzelnen Zeitservern entsteht, möglichst gleichmäßig auf alle Zeitserver verteilt werden. Diesen beiden Zielen muss allerdings unterschiedliche Bedeutung beigemessen werden. So kann eine schlechte Ausnutzung der Zykluszeit sehr schnell zu einer Sättigung des Netzwerks führen. In der Folge können dem Netzwerk keine neuen Geräte mehr beitreten, ohne die Echtzeit-Eigenschaften der Kommunikation zu gefährden. Zwar gefährdet die Überlastung eines Zeitservers auch die Echtzeit-Eigenschaften, da sie zum Scheitern der Zeitsynchronisation führen könnte, wodurch keine einheitliche Zeitbasis im Netzwerk mehr herrschen würde. Allerdings tritt eine höhere Belastung der Zeitserver erst auf, wenn sich bereits eine Vielzahl von Geräten im Netzwerk befindet. Um diesem Umstand gerecht zu werden, wird ein zweistufiger Ansatz für die Verteilung der Zeitschlitze gewählt.

Wie bereits in Kapitel 6.2 erläutert, sendet ein neuer Netzwerk-Teilnehmern einen POST-Request für die *„/well-known/timeslot“*-Ressource an einen der Zeitserver, um einen Zeitschlitz abzurufen. An diesen Request fügt er seine gewünschte Kommunikationsperiode t_{period} an. Ist t_{period} größer als die Zykluszeit t_{cycle}, beginnt der Zeitserver mit der ersten Stufe der Zeitschlitzauswahl. Diese ist auf eine möglichst effiziente Nutzung der Zykluszeit ausgerichtet. Dabei prüft der Zeitserver, ob bereits ein Zeitschlitz existiert, dessen Kommunikationsperiode t_{comm} der Bedingung in Formel (22) entspricht und dessen Kapazität noch nicht erreicht ist.

$$t_{period} \geq t_{comm} > t_{cycle},\ mit\ \{t_{period} - t_{comm} \rightarrow 0\} \tag{22}$$

Ist die Kapazität dieses Zeitschlitzes noch nicht erreicht, so kann das neue Gerät ebenfalls diesem Zeitschlitz zugeordnet werden. Auf diese Weise soll verhindert werden, dass die verfügbaren Zeitschlitze durch Geräte mit einer hohen Kommunikationsperiode blockiert werden, sodass Geräte mit $t_{period} \approx t_{cycle}$ dem Netzwerk nicht mehr beitreten können.

Kann kein Zeitschlitz gefunden werden, der diese Bedingung erfüllt, fährt der Zeitserver mit der zweiten Stufe fort. Hier steht die Auslastung der einzelnen Zeitserver im Fokus. Wie bereits in Kapitel 7.1 beschrieben, ist jeder der Zeitserver für einen bestimmten Bereich des Zeitzyklus verantwortlich, sodass über die Auswahl des Zeitschlitzes gleichzeitig der zuständige Zeitserver bestimmt wird. Die Auslastung aller Zeitserver im Netzwerk lässt sich demnach am aktuellen Zustand des Zeitzyklus ablesen. In der zweiten Stufe wird dem Gerät daher ein Zeitschlitz in dem Zuständigkeitsbereich des Zeitservers mit der geringsten Auslastung zugewiesen. Auf diese Weise soll eine gleichmäßige Verteilung der Synchronisationslast auf die einzelnen Zeitserver erreicht werden. Abbildung 51 verdeutlicht dies beispielhaft an einem Netzwerk mit zehn Zeitschlitzen und drei Zeitservern. Dabei sind bereits zwei Geräte mit einer Kommunikationsperiode $t_{period} = t_{cycle}$ im Netzwerk aktiv und wurden den Zeitschlitzen T1 und T5 zugeordnet. Tritt nun ein drittes Gerät bei, wird ihm ein Zeitschlitz im Verantwortungsbereich des dritten Zeitservers zugeordnet, da die Kapazität der Zeitschlitze T1 und T5

auf Grund der niedrigen Kommunikationsperiode bereits erreicht ist und der dritte Zeitserver bisher die geringste Auslastung aufweist.

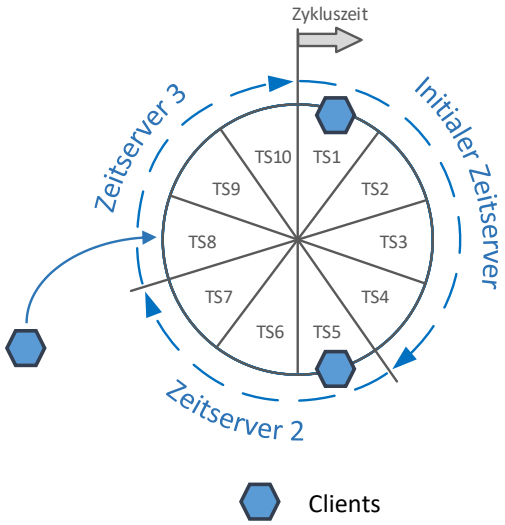

Abbildung 51: Verteilung der Clients auf die verschiedenen Zeitserver im Netzwerk zur Reduktion der Synchronisationslast pro Server.

7.4 Fehlerbehandlung

Die Verfügbarkeit der Zeitserver ist von herausragender Bedeutung. Fällt einer der Zeitserver aus, so können die Geräte innerhalb seines Verantwortungsbereichs ihre Systemzeit nicht mehr synchronisieren. Dies führt mittelfristig zu einem Auseinanderdriften der Systemzeiten, wodurch die Einhaltung des TDMA-Verfahrens nicht mehr garantiert werden kann. Daher ist es wichtig, den Ausfall eines Zeitservers festzustellen und entsprechende Gegenmaßnahmen zu treffen. Wie der Ausfall eines Zeitservers festgestellt werden kann, hängt dabei davon ab, ob es sich um den initialen Zeitserver handelt. Fällt der initiale Zeitserver aus, so kann dies von den anderen Zeitservern schnell durch das Ausbleiben der Synchronisationsantworten festgestellt werden. In diesem Fall übernimmt der erste Zeitserver, der den Ausfall feststellt, die Funktion des initialen Zeitserver. Zuvor teilt er den anderen Zeitservern den Funktionswechsel mit. Fällt ein anderer Zeitserver aus, kann dies durch den initialen Zeitserver erkannt werden, da er in regelmäßigen Abständen Synchronisations-Requests von den untergeordneten Zeitservern erhält. Bleiben diese aus, so ist mit einer Fehlfunktion des entsprechenden Zeitservers zu rechnen. Der Ausfall eines Zeitservers kann auf zwei Arten kompensiert werden. Es kann zum einen ein neuer Zeitserver im Netzwerk ernannt werden. Diesem Zeitserver wird der gleiche Bereich im Kommunikationszyklus zugewiesen. In der Folge müssen alle Knoten, die direkt vom Ausfall des Zeitservers betroffen sind, über ihren neuen Zeitserver informiert werden. Zum anderen können die Zuständigkeitsbereiche der noch verbleibenden Zeitserver so skaliert werden, dass der

Ausfall kompensiert wird. Hierbei müsste allerdings eine vollständige Umstrukturierung der Zuordnung zwischen den Geräten und ihren Zeitservern erfolgen. Dies kann zu einem erheblichen Kommunikationsaufwand führen. Daher sollte dieses Vorgehen nur in Ausnahmefällen gewählt werden, wenn beispielsweise kein weiterer Zeitserver im Netzwerk verfügbar ist.

7.5 Evaluation eines Prototypen

Um die Machbarkeit und die Eignung des Ansatzes eines verteilten Zeitservers für den Einsatz in Echtzeitszenarien zu untersuchen, wurden die in diesem Kapitel beschriebenen Mechanismen in jCoAP integriert. Weiterhin wurde das in Kapitel 6.4 verwendete Testbed, wie in Abbildung 52 dargestellt, um zwei weitere CoAP-Clients und einen CoAP-Zeitserver erweitert.

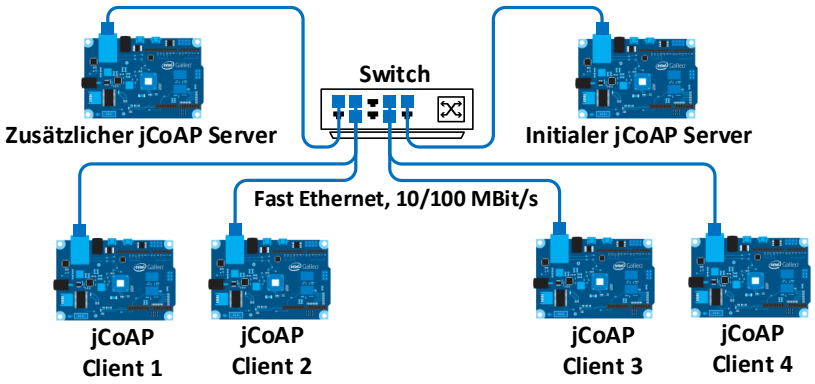

Abbildung 52: Aufbau der für die Performance-Messungen verwendeten Testumgebung [60].

Zur Analyse der Performance der beschriebenen Ansätze wurden zwei unterschiedliche Szenarien untersucht. Im ersten Testszenario wurden vier Clients und ein einziger Zeitserver verwendet. Im zweiten Testszenario wird dem Netzwerk ein weiterer Zeitserver hinzugefügt. In jedem der Experimente wurden zunächst die Zeitserver gestartet. Nach der abgeschlossenen Initialisierung der Zeitserver wurden die Clients gestartet. Die Clients rufen einen Zeitschlitz bei einem der Zeitserver ab und synchronisieren ihre Systemzeit mit dem ihnen zugewiesenen Zeitserver. Für alle Geräte wurde eine Re-Synchronisationsperiode von 30 s gewählt, um ein Auseinanderdriften der Systemzeiten zu verhindern und eine einheitliche Zeitbasis im Netzwerk sicherzustellen. Für alle Experimente wurde eine Zykluszeit von 100 ms gewählt. Der Kommunikationszyklus wurde in zehn Zeitschlitze mit einer Länge von 10 ms unterteilt. Die geforderte Kommunikationsperiode der Clients entsprach in allen Versuchen der Zykluszeit. Tabelle 9 zeigt die Zuordnung der Zeitschlitze zu den einzelnen Geräten für beide Testszenarien.

Es zeigt sich, dass der Kommunikationszyklus im ersten Szenrio mit nur einem Zeitserver nach und nach aufgefüllt wurde. Im zweiten Testszenario wurden die Geräte hingegen über den gesamten Kom-

	Szenario 1	Szenario 2
Initialer Server	1	1
Zusätzlicher Server	-	6
Client 1	2	2
Client 2	3	7
Client 3	4	3
Client 4	5	8

Tabelle 9: Zeitschlitz-Zuordnung in beiden Testszenarien.

munikationszyklus verteilt. Das beobachtete Verhalten entspricht dabei genau den in Kapitel 7.3 beschriebenen Mechanismen. Aufgrund der geringen Kommunikationsperiode der Geräte konnte in der ersten Stufe kein geeigneter Zeitschlitz gefunden werden. Daher wurden die Geräte in der zweiten Stufe gleichmäßig auf die Verwantwortungsbereiche der beiden Zeitserver verteilt.

Während der Experimente wurde auf allen Geräten die Anzahl der gesendeten Synchronisations-Requests gemessen. Demgegenüber wurde von den Zeitservern die Anzahl der empfangenen Synchronisations-Requests protokolliert. Abbildung 53 zeigt die erhaltenen Ergebnisse für beide Szenarien. Die y-Achse zeigt dabei die Anzahl der gesendeten und empfangenen Synchronisations-Requests über einen Zeitraum von 120 Minuten.

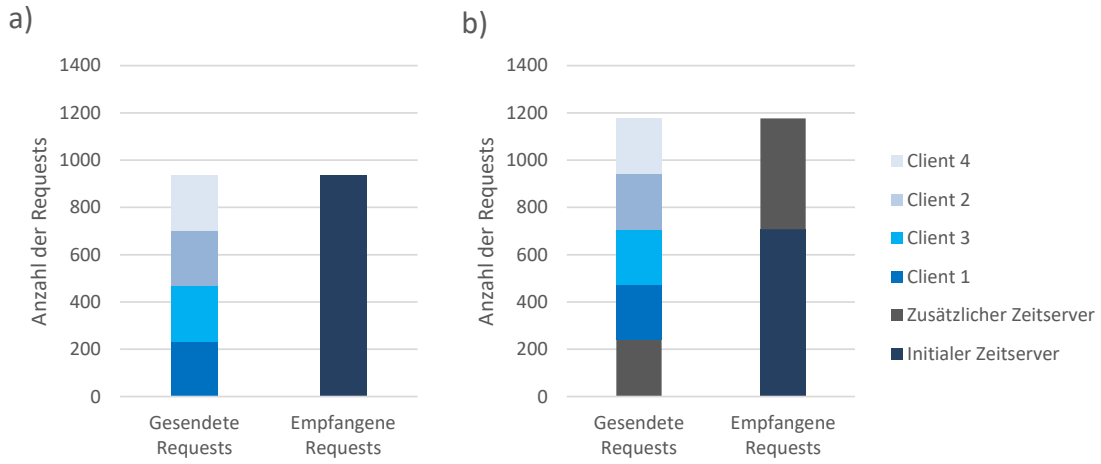

Abbildung 53: Anzahl der gesendeten und empfangenen Synchronisations-Requests in einer Testumgebung mit a) einem und b) zwei Zeitservern [60].

Es zeigt sich, dass im ersten Szenario alle Synchronisations-Requests von dem einzigen Zeitserver im Netzwerk verarbeitet wurden. Weiterhin konnte im zweiten Szenario ein Anstieg des Netzwerkverkehrs zur Zeitsynchronisation um ca. 25,5% beobachtet werden. Dies ist darauf zurückzuführen, dass sich der zusätzliche Zeitserver ebenfalls mit dem initialen Zeitserver synchronisieren muss. Allerdings konnte die Anzahl der je Zeitserver zu verarbeitenden Synchronisations-Requests um mindestens 25% verringert werden, da die Clients 2 und 4 ihre Synchronisationsnachrichten an den zweiten

Zeitserver senden. Durch das Hinzufügen eines dritten Zeitservers kann in diesem Szenario eine Reduktion der Synchronisationsnachrichten pro Zeitserver um 50% erreicht werden.[10] Hierbei würden sich nur die zusätzlichen Zeitserver mit dem initialen Zeitserver synchronisieren, während die Clients auf die zusätzlichen Zeitserver verteilt würden. Jedoch würde der Kommunikationsaufwand zur Zeitsynchronisation insgesamt ansteigen, da sich auch der dritte Zeitserver mit dem initialen Zeitserver synchronisieren muss.

Für den Einsatz in Echtzeitszenarien spielt die Genauigkeit der Zeitsynchronisation eine bedeutende Rolle. Daher wurde in einem weiteren Experiment die Qualität der Zeitsynchronisation zwischen den Zeitservern untersucht. Hierzu wurden die Sendezeiten der Synchronisations-Requests auf dem zusätzlichen Zeitserver gemessen. Darüber hinaus wurden die Empfangszeiten der Requests auf dem initialen Zeitserver gespeichert. Weiterhin wurde die Latenz der Synchronisations-Requests von dem zusätzlichen Zeitserver bestimmt. Hierzu wurde deren RTT gemessen, um die Bearbeitungszeit auf dem initialen Zeitserver bereinigt und halbiert. Die Messungen wurden über einen Zeitraum von 20 Stunden bzw. 2.400 Synchronisationsperioden durchgeführt. Eine Gegenüberstellung der Differenz aus Sende- und Empfangszeit mit der beobachteten Latenz soll Aufschluss über den Zeitversatz zwischen den Geräten und damit die Güte der Zeitsynchronisation geben. Abbildung 54 zeigt die erzielten Ergebnisse. Die x-Achse gibt die Gesamtzeit des Experiments in Stunden an, während die y-Achse die relative Zeit in Millisekunden zeigt. Die hellblauen Punkte repräsentieren die Differenz zwischen der Sendezeit auf dem zusätzlichen Zeitserver und der Empfangszeit auf dem initialen Zeitserver. Die dunkelblauen Punkte zeigen die von dem zusätzlichen Zeitserver gemessene Latenz. Die Tabelle 10 fasst zudem die mittlere Zeitdifferenz und Latenz und die jeweilige Standardabweichung zusammen.

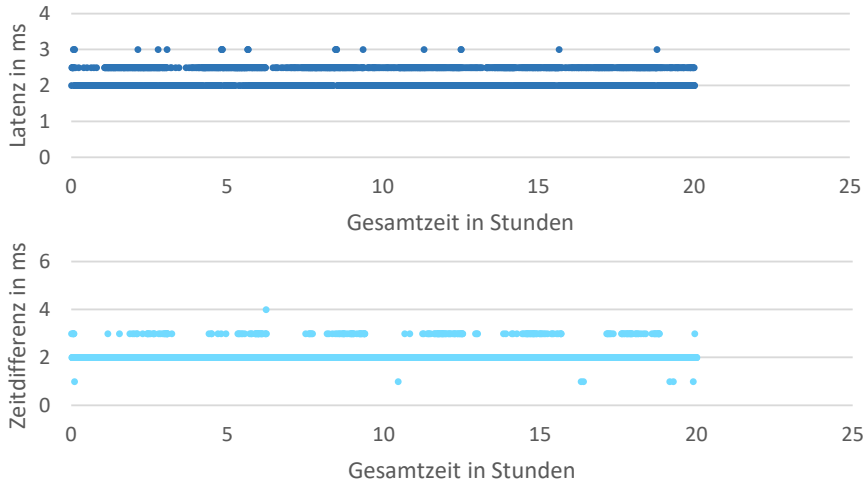

Abbildung 54: Zeitdifferenz zwischen dem Senden und Empfangen der Synchronisations-Requests und die beobachtete Netzwerklatenz [60].

[10] Im Vergleich zu Szenario 1.

	Zeitdifferenz	Latenz
Durchschn.	2.10 ms	2.23 ms
Median	2.00 ms	2.00 ms
Standardabweichung	0.30 ms	0.26 ms
Min	1.00 ms	2.00 ms
Max	4.00 ms	3.00 ms

Tabelle 10: Differenz zwischen Sende- und Empfangszeit der Synchronisations-Requests und beobachtete Latenz.

Es zeigt sich, dass die Zeitdifferenz zwischen der Sende- und Empfangszeit der Synchronisations-Requests in etwa der beobachteten Netzwerklatenz entspricht. Die geringe Standardabweichung der Latenz und der Zeitdifferenz lässt auf eine geringe Schwankung der Werte schließen. Demnach kann geschlussfolgert werden, dass die beobachtete Zeitdifferenz fast ausschließlich auf die Übertragungszeit des Requests zurückzuführen ist. Hieraus ergibt sich, dass die verfeinerte Zeitsynchronisation eine hinreichende Genauigkeit für die angestrebte maximale Zeitabweichung von 1 ms bietet, da die tatsächliche Zeitdifferenz zwischen den Knoten in jeder Synchronisationsperiode deutlich unterhalb der Grenze von einer Millisekunde liegt.

In einem letzten Experiment wurde das Gesamtsystem betrachtet. Dabei wurde untersucht, inwieweit sich ein hierarchischer Synchronisationsansatz auf das TMDA-Verfahren auswirkt. Hierzu wurden die Empfangszeiten aller Synchronisations-Requests der einzelnen Geräte auf den Zeitservern gemessen. Abbildung 55 zeigt die erhaltenen Ergebnisse. Die x-Achse repräsentiert die Gesamtzeit des Experiments in Minuten, während die y-Achse die relative Zeit innerhalb eines Kommunikationszyklus in Millisekunden darstellt. Die roten Linien markieren hierbei die Grenzen zwischen den Zeitschlitzen.

Es zeigt sich, dass alle Geräte nur innerhalb ihres Zeitschlitzes auf das Netzwerk zugreifen. Darüber hinaus wurden alle Geräte gleichmäßig in dem Kommunikationszyklus verteilt, sodass die Synchronisationslast je Zeitserver minimiert wird. Die Ergebnisse unterstreichen nicht nur die praktische Umsetzbarkeit der beschriebenen Mechanismen, sondern auch die Vorteile in Bezug auf Robustheit und Skalierbarkeit. Weiterhin verdeutlichen die Versuchsergebnisse die Eignung der dargestellten Ansätze für den Einsatz in Umgebungen mit Echtzeitanforderungen.

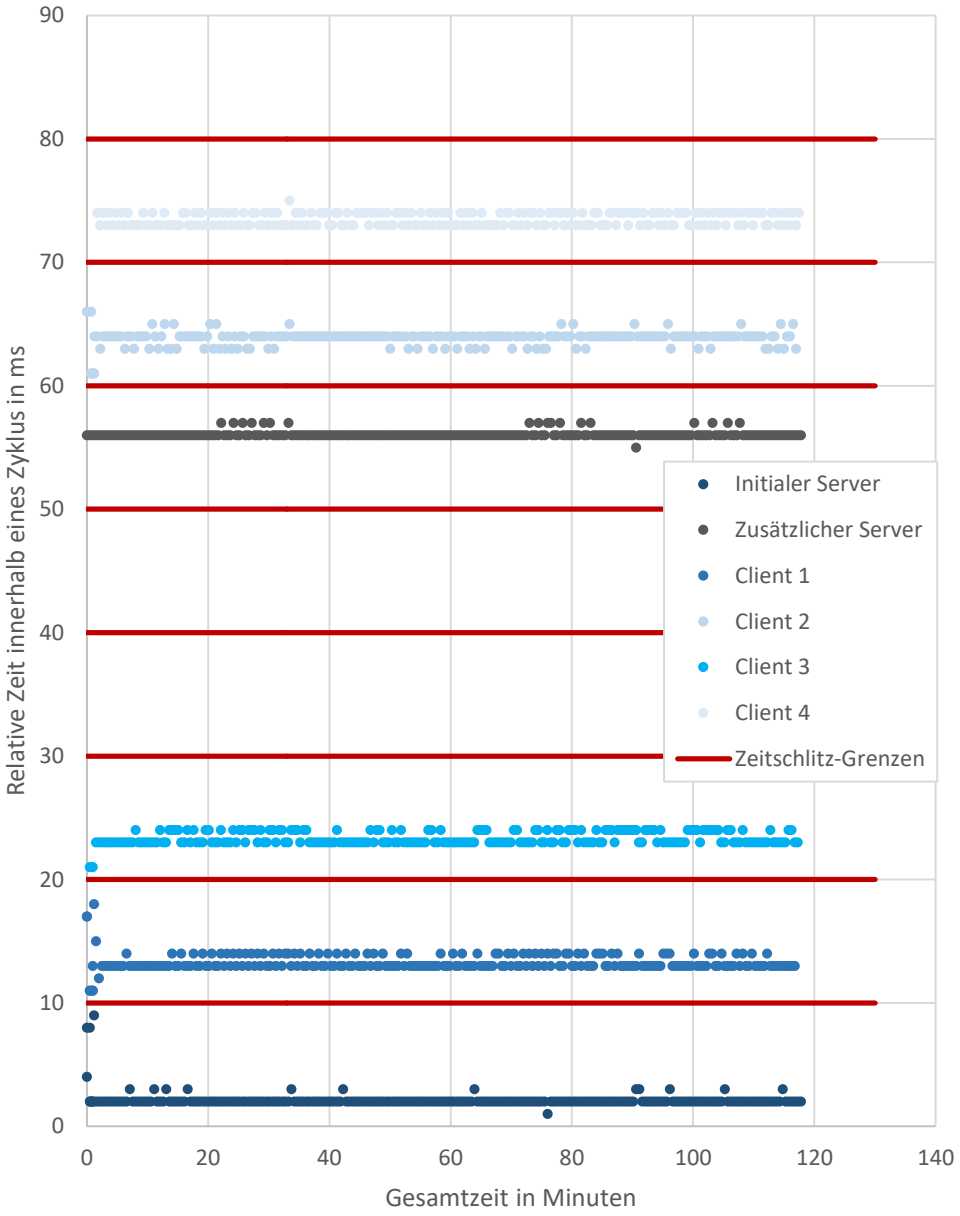

Abbildung 55: Darstellung der von den Servern gemessenen Empfangszeitpunkte der einzelnen Nachrichten der Clients und des zusätzlichen Zeitservers [60].

8 Ableitung einer Echtzeit-Spezifikation für CoAP

Die vorangegangenen Kapitel haben gezeigt, dass eine zeitlich deterministische Kommunikation über Standard-Ethernet auf Basis des CoAP möglich ist. Dabei sind die in dieser Arbeit vorgestellten Ansätze von der verwendeten Kommunikationstechnologie unabhängig und lassen sich vollständig mit den CoAP-inherenten Mechanismen umsetzen. Aus den beschriebenen Verfahren lässt sich entsprechend eine allgemein gültige Echtzeiterweiterung für das CoAP ableiten. In den nachfolgenden Abschnitten werden zunächst generelle Überlegungen, die bei der Definition einer entsprechenden Standarderweiterung beachtet werden müssen, betrachtet. Anschließend wird ein einheitliches Datenformat zum Austausch verfahrensrelevanter Daten erarbeitet. Die Standarderweiterung wurde bisher noch nicht veröffentlicht.

8.1 Definition neuer Ressourcen, Optionen und Verfahrensweisen

Auf Grund seiner modularen Struktur ließe sich eine Echtzeiterweiterung nahtlos als eigener Substandard in den CoAP-Standard integrieren. In der angestrebten Standarderweiterung müssen zunächst die nötigen Schnittstellen verankert werden. Tabelle 11 gibt einen Überblick über diese Schnittstellen und ihre jeweiligen Aufgabe.

Ressource / Option	URI / Optionsnummer	Aufgabe
timeslot	./well-known/timeslot	Diese Ressource erlaubt das Abrufen eines Zeitschlitzes von einem Zeitserver. Sie muss von allen echtzeitfähigen CoAP-Servern angeboten werden.
time	./well-known/time	Über diese Ressource kann die Zeitsynchronisation mit einem Server gestartet werden.
synclient	./well-known/synclient	Hinter dieser Ressource verbirgt sich ein CoAP-Client, der die Zeitsynchronisation zwischen zwei Servern ermöglicht.
SYN	# 40	Mit Hilfe der SYN-Headeroption zeigt ein Client seinen Willen zur Zeitsynchronisation an. Über diese Option wird auch das zu verwendende Synchronisationsverfahren bestimmt.

Tabelle 11: Ressourcen und Header-Optionen, die in einer Echtzeiterweiterung für den CoAP-Standard festzuschreiben sind.

Weiterhin müssen die Abläufe sowohl zur Zeitsynchronisation als auch zur Vergabe der Zeitschlitze beschrieben werden. Diese Beschreibung muss dabei das Verhalten des Zeitservers sowie der übrigen CoAP-Geräte umfassen. Hierbei müssen beide Varianten des Zeitservers, zentralisiert (vergleiche Kapitel 6) und verteilt (vergleiche Kapitel 7) Beachtung finden. Wird ein verteilter Zeitserver verwendet, muss weiter zwischen dem initialen und einem nachträglich bestimmten Zeitserver unterschieden werden. Der verteilte Zeitserver soll als optionaler Mechanismus in die Standarderweiterung aufgenommen werden. Zwar erhöht er die Skalierbarkeit und die Robustheit gegenüber Serverfehlern durch die Entlastung der Zeitserver und die redundante Speicherung der Informationen über den Kommunikationszyklus. Jedoch wird hierdurch auch die Komplexität des Gesamtsystems erhöht. In Anbetracht der Tatsache, dass CoAP auf Umgebungen mit stark ressourcenbeschränkten Geräten ausgelegt ist und die Vorteile des verteilten Zeitservers besonders in größeren Netzwerken zum Tragen kommen, wird eine verpflichtende Unterstützung dieser Funktion nicht als zielführend erachtet. Vielmehr soll eine Auswahl der geeigneten Mechanismen durch den Anwender unter Berücksichtigung des jeweiligen Einsatzszenarios möglich bleiben. Die Verhaltensweise der übrigen Netzwerkteilnehmer bleibt von der Entscheidung für eine Zeitservervariante unberührt. Die Abbildungen 56, 57 und 58 zeigen schematisch den Ablauf innerhalb eines initialen und eines zusätzlichen Zeitservers sowie eines einfachen Netzwerkteilnehmers. Dabei orientieren sich die Abläufe jeweils an den in den Kapitel 7.1, 7.2 und 7.3 beschriebenen Mechanismen und unterteilen sich in eine Initialisierungs- und eine Kommunikationsphase.

In der Intitialisierungsphase wird zunächst der initiale Zeitserver bestimmt. Dieser startet einen Discovery-Vorgang, um weitere potentielle Zeitserver im Netzwerk zu finden. Von diesen wählt der initiale Zeitserver n als zusätzliche Zeitserver aus. Die Auswahl kann dabei anhand unterschiedlicher Kriterien erfolgen. In der prototypischen Umsetzung werden die Zeitserver mit der kürzesten Antwortzeit ausgewählt, da auf diese Weise mögliche Synchronisationsfehler zwischen den Zeitservern minimiert werden können. Es sind jedoch auch andere Kriterien denkbar, wie beispielsweise das Einbeziehen von Topologie-Informationen, um eine möglichst gleichmäßige Verteilung der Zeitserver im Netzwerk zu gewährleisten. Die beste Vorgehensweise hängt dabei stark von dem jeweiligen Anwendungsfall ab, sodass hierzu in einer Standard-Erweiterung keine Vorgaben gemacht werden sollen. Weiterhin startet der initiale Zeitserver einen Synchronisationsclient auf den ausgewählten Servern mittels eines POST-Requests. Hierbei übergibt er jedem der neu ausgewählten Zeitserver eine vollständige Liste der aktiven Zeitserver. Weiterhin beobachtet der initiale Zeitserver die jeweiligen timeslot-Ressourcen der anderen Server mit Hilfe von CoAP Observe.

Nachdem der Synchronisationsclient auf den zusätzlich akquirierten Zeitservern gestartet wurde, abonnieren diese zunächst Änderungen an der „/.well-known/timeslot“-Ressource des initialen und der anderen zusätzlich ausgewählten Zeitserver, um so direkt über Veränderungen in der Zyklusaufteilung informiert zu werden. Anschließend wird über den Synchronisationsclient selbst ein Zeitschlitz angefordert, der zukünftig von dem jeweiligen Server genutzt werden soll. Abschließend wird die

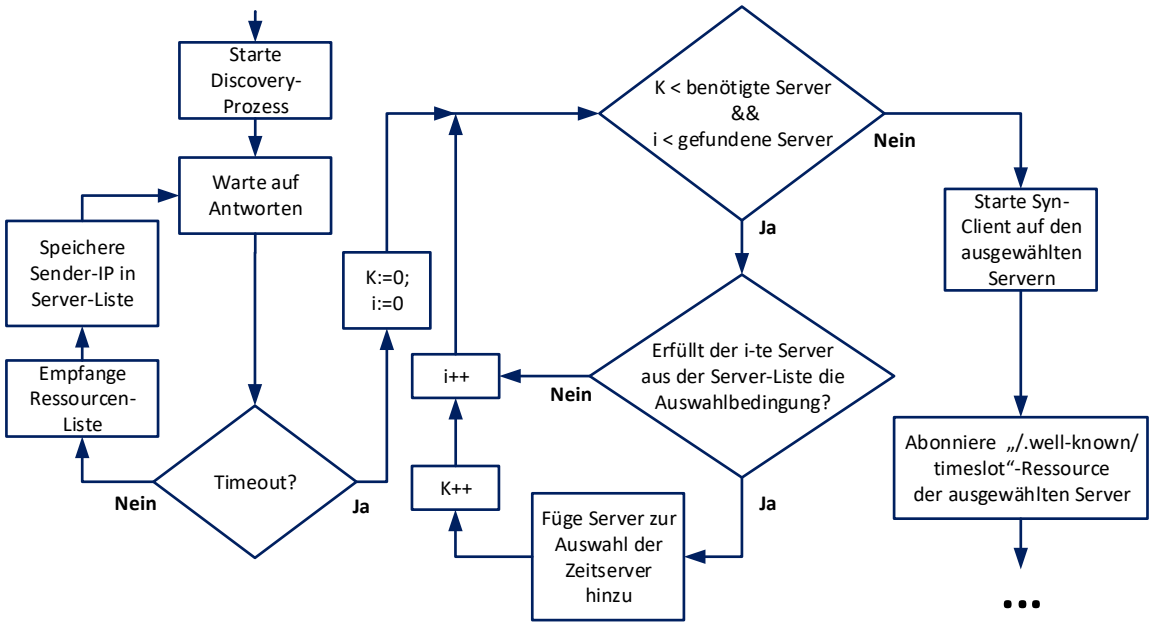

Abbildung 56: Darstellung der Vorgehensweise des initialen Zeitservers bei der Suche und Auswahl zusätzlicher Zeitserver als Flussdiagramm.

Systemzeit mit dem initialen Zeitserver synchronisiert. Sind alle Zeitserver untereinander synchronisiert und haben ihre Sicht auf den Zeitzyklus abgeglichen, ist die Initialisierungsphase beendet.

In der Kommunikationsphase gleichen die zusätzlichen Zeitserver periodisch ihre Systemzeit mit dem initialen Zeitserver ab. Die hierfür zu wählende Periode hängt zum einen von der Genauigkeit des verwendeten Synchronisationsverfahrens ab. Zum anderen spielen dabei die Genauigkeit der in dem jeweiligen Gerät verbauten Uhr sowie die im Gesamtsystem angestrebte Genauigkeit der Systemzeit eine Rolle. In der Kommunikationsphase können zudem dynamisch Clients in das Netzwerk eingebracht werden. Die Clients suchen hierbei zunächst nach einem Zeitserver im Netzwerk. Von diesem fordern sie anschließend einen Zeitschlitz an. Hierbei wird ihnen von dem angefragten Zeitserver auch der für sie jeweils zuständige Zeitserver mitgeteilt. Welcher Zeitserver für den jeweiligen Client zuständig ist, hängt von dem Ziel der Verteilung ab. In der prototypischen Umsetzung wird eine möglichst gleichmäßige Auslastung der Zeitserver angestrebt. Jedoch kann die Zuordnung auch anhand von anderen Kriterien, wie beispielsweise der räumlichen Nähe, erfolgen. Da die Entscheidung für ein Verteilungskriterium vom jeweiligen Anwendungsfall abhängt, sollte dies dem Entwickler überlassen bleiben und nicht in einer Standarderweiterung festgeschrieben werden. Der Client führt die Zeitsynchronisation mit dem ihm zugeteilten Zeitserver durch. Die Zeitsynchronisation wird dabei ebenfalls periodisch wiederholt.

In der prototypischen Umsetzung wählt der Netzwerkteilnehmer über die SYN-Header-Option in seinem Synchronisations-Request das zu verwendende Synchronisationsverfahren. Dieses Vorgehen

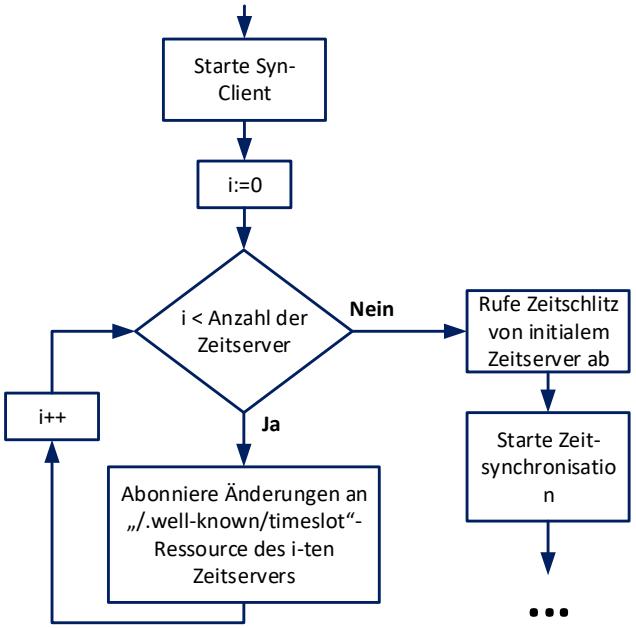

Abbildung 57: Darstellung der Abläufe auf den zusätzlichen Zeitservern während der Initialisierungsphase als Flussdiagramm.

ist in der Praxis problematisch, da so ein einzelner Netzwerkteilnehmer durch die Auswahl eines Synchronisationsverfahrens mit zu geringer Genauigkeit das gesamte TDMA-Verfahren kompromitieren kann. Auf diese Weise wird das Echtzeitverhalten der Kommunikation gefährdet. Daher soll die Entscheidungsgewalt über das Synchronisationsverfahren in einer Standarddefinition auf den Zeitserver übertragen werden. Um dies zu ermöglichen, fügt der anfragende Netzwerkteilnehmer die SYN-Header-Option mehrfach in seinen Synchronisations-Request ein und listet so alle von ihm unterstützten Synchronisationsverfahren auf. Der Zeitserver wählt aus diesen Verfahren nun dasjenige aus, welches am besten den zeitlichen Anforderungen im Netzwerk entspricht. In seiner Antwort fügt der Zeitserver ebenfalls die SYN-Header-Option mit dem ausgewählten Synchronisationsverfahren ein. Auf diese Weise weiß der anfragende Netzwerkteilnehmer, welches Verfahren verwendet wird und wie er die in der Antwort enthaltenen Daten interpretieren muss. In zukünftigen Synchronisations-Requests fügt der CoAP-Client die SYN-Header-Option nur einmal mit dem jeweils ausgewählten Verfahren ein. Somit wird eine ständige neue Prüfung der Verfahren auf der Serverseite unterbunden und es werden Systemressourcen auf dem Server eingespart.

Weiterhin muss in der Standarderweiterung ein Katalog von möglichen Synchronisationsverfahren definiert werden. Anfänglich soll dieser Katalog die in den Kapiteln 6.1 und 7.2 beschriebenen Synchronisationsmodi enthalten. Das „Synchronization Mode“-Feld in der SYN-Header-Option erlaubt mit einer Breite von 8 Bit eine Gesamtzahl von 255 möglichen Verfahren. Die verbleibenden Plätze in dem Katalog sollen nach und nach durch die Community mit Verfahren, wie zum Beispiel dem in

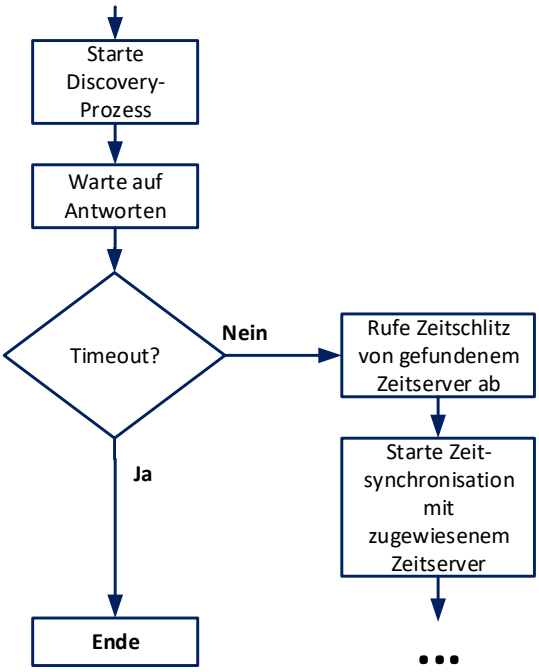

Abbildung 58: Darstellung der Abläufe auf der Client-Seite beim Beitritt in das Netzwerk.

Nummer	Verfahren
0x00	*Cristians Algorithmus*
0x01	Verbesserter *Cristians Algorithmus*
0x02 - 0xFF	Frei belegbar

Tabelle 12: Katalog der Synchronisationsmodi für eine RT-CoAP-Standard-Erweiterung.

dem Standard IEEE 1588 beschriebenen Precision Time Protocol (PTP), erweitert werden. Weiterhin soll Raum für anwendungsspezifische Verfahren gelassen werden. Tabelle 12 gibt einen Überblick über den Aufbau des Verfahrenskatalogs.

Für jedes neue Verfahren soll eine Sub-Spezifikation entstehen, die beschreibt, wie das jeweilige Synchronisationsverfahren auf CoAP und die in dieser Arbeit beschriebene Zeitserverstruktur übertragen werden soll.

8.2 Definition eines einheitlichen Datenformates

Weiterhin werden im Rahmen der beschriebenen Mechanismen viele unterschiedliche Daten ausgetauscht. So übertragen die CoAP-Geräte Informationen über ihre zeitlichen Anforderungen an die Kommunikation an die Zeitserver. Auf der anderen Seite übertragen die Zeitserver alle benötigten Informationen über den Kommunikationszyklus, wie die Zykluszeit, den zugewiesenen Zeitschlitz und den zuständigen Zeitserver, an die anderen Geräte. Zudem können auch die Daten, welche für die Zeit-

synchronisation benötigt werden, von unterschiedlicher Komplexiät sein. Dies wird bereits bei einer Gegenüberstellung der in den Kapiteln 6.1 und 7.2 beschriebenen Synchronisationsverfahren deutlich. Die SYN-Header-Option bietet zudem die Möglichkeit, beliebige weitere Synchronisationsverfahren zu definieren. In den bisherigen prototypischen Umsetzungen der dargestellten Verfahren mit jCoAP wurden die Daten in festgelegter Reihenfolge hintereinander übertragen. In dieser Darstellungsform fehlen zudem Informationen über die Einheiten der jeweiligen Zeitwerte, sodass bisher von einer Darstellung in Millisekunden ausgegangen wird. Um eine interoperable Kommunikation zwischen den einzelnen CoAP-Geräten auch über verschiedene CoAP-Implentierungen hinweg zu gewährleisten, muss ein einheitliches und erweiterbares Datenformat definiert werden. Hierfür stehen verschiedene Möglichkeiten der Umsetzung zur Verfügung. Ein weit verbreitetes Format, welches sich durch eine hervorragende Erweiterbarkeit auszeichnet, wird mit der Extensible Markup Language (XML) beschrieben. XML basiert jedoch auf der Verwendung menschenlesbarer Tags zur Strukturierung der Daten. Weiterhin können XML-Schemata definiert werden, welche die zu verwendenden Tags und ihre Reihenfolge in einem Dokument vorgeben. Demzufolge werden die so beschrieben Nachrichten sehr groß, was einer Verwendung in ressourcenbeschränkten Umgebungen, für die CoAP entwickelt wurde, entgegen steht. Mit dem Efficient XML Interchange (EXI) besteht die Möglichkeit, XML-basierte Nachrichten erheblich zu komprimieren. Es wurde von dem World Wide Web Consortium (W3C) im Rahmen der Efficient XML Interchange Working Group im März 2011 standardisiert. EXI kann dabei in zwei möglichen Modi ausgeführt werden, schema-informed und schema-less. Beim schema-informed Modus wird aus einem XML-Schema ein Automat generiert, welcher anschließend verwendet wird, um die Tags durch Indizes zu ersetzen und so einen sogenannten EXI-Stream zu generieren. Auf diese Weise kann eine erhebliche Kompression der XML-Dokumente erreicht werden. Ist im Vorfeld kein XML-Schema bekannt, so kann EXI im schema-less Modus angewendet werden. Hierbei werden die entsprechenden Indizes während des Kodierungsvorgangs beim ersten Auftreten eines Tags gebildet. Im Ergebnis kommt jeder Tag nur ein einziges Mal in seiner Klartext-Form vor. Entsprechend führt der schmema-less-Modus zu einer geringeren Kompression, ermöglicht aber auch das Kodieren und Dekodieren von XML-Dokumenten mit unbekannter Struktur. Das W3C führte auch Untersuchungen zur Effizienz von EXI durch [110]. Dabei stand neben der Kompressionsrate auch die Verarbeitungszeit bei der Kodierung und Dekodierung der Nachrichten im Vordergrund. Die Ergebnisse der Untersuchung zeigen eine hervorragende Kompressionsrate von EXI. Jedoch offenbaren sie auch, dass die Kodierung beziehungsweise De-Kodierung je nach Dokumentenklasse und -größe bis zu zehn Mal länger dauern kann als bei klassischen XML-Dokumenten. Diesem Problem nehmen sich Altmann et al. in ihrer Arbeit an [8]. Ziel ist es dabei nicht nur, die Verarbeitungszeit zu reduzieren, sondern vor allem ein deterministisches Zeitverhalten zu gewährleisten. Zu diesem Zweck entwerfen Altmann et al. eine reine Hardware-Umsetzung sowie ein Hardware-Software-Co-Design eines EXI-Parsers und vergleichen deren Zeitverhalten mit einer klassischen Software-Umsetzung. Die Ergebnisse zeigen, dass mit dem Hardware-Software-Co-Design eine Reduktion der Verarbeitungszeit um 23% erreicht werden kann. Die reine Hardware-Umsetzung erzielt hingegen ei-

ne Verkürzung der zum Nachrichten-Parsing benötigten Zeit um 95%. Dies legt die Verwendung von EXI-kodierten XML-Nachrichten für den Datenaustausch nahe. Jedoch ist das Abstellen auf spezialisierte Hardware in heterogenen Geräte-Ensembles problematisch. In solchen Szenarien kann nicht davon ausgegangen werden, dass alle Geräte im Netzwerk über diese Hardware verfügen. Als Kompromisslösung wird in dieser Arbeit daher ein Datenformat auf Basis der JavaScript Object Notation (JSON) vorgeschlagen [18]. Obwohl die Notation in JSON erhebliche Ähnlichkeiten mit XML aufweist, bestehen bedeutende Unterschiede. Während es sich bei XML um eine Sprache zur Beschreibung von Daten und ihrer Struktur handelt, so lassen sich mit JSON hingegen nur bereits strukturierte Daten kompakt darstellen. JSON beinhaltet somit nur implizit Informationen über die Struktur der Daten. Allerdings lassen sich Objekte deutlich kompakter darstellen, da auf ein System aus öffnenden und schließenden Tags verzichtet wird. In Listing 1 wird ein beispielhaftes JSON-Objekt beschrieben. Objekte werden in JSON durch geschweifte Klammern begrenzt. Innerhalb dieser Objekte erfolgt eine Zuordnung von Attributen mit frei wählbarem Namen zu deren Werten. Einem Attribut kann dabei auch ein weiteres JSON-Objekt als Wert zugewiesen werden.

```
{
    "Attribut1_1":"Wert1",
    "Attribut1_2":"Wert2",
    "Attribut1_3":  {
                        "Attribut2_1":"Wert2_1",
                        "Attribut2_2":"Wert2_2"
                    }
}
```

Listing 1: Prinzipieller Aufbau eines JSON-Objektes.

Das Fehlen einer expliziten Beschreibung der Datenstruktur in JSON wirkt sich in dem hier betrachteten Anwendungsfall nicht negativ aus. Da der Client das Synchronisationsverfahren über die SYN-Header-Option selbst mitbestimmt, kann davon ausgegangen werden, dass er bereits über ausreichendes Vorwissen über die zu erwartenden Daten verfügt. Zudem kann eine entsprechende Benennung der Objekt-Attribute aus der Beschreibung des jeweiligen Synchronisationsverfahrens aus dem CoAP-Standard entnommen werden. Vielmehr steht hier die einheitliche Darstellung und die Abgrenzbarkeit einzelner Attributwerte im Vordergrund. Listing 2 zeigt die JSON-Darstellung der Zeitschlitzinformationen, welche beim Abrufen eines Zeitschlitzes übertragen werden. Neben den eigentlichen Werten werden zudem Einheiten für die Zeitwerte mit übertragen. Um eine kompakte Darstellung zu gewährleisten, sollen die Einheiten als Index dargestellt werden. Die Zuordnung der Indizes zu den jeweiligen Einheiten soll in der Standarderweiterung festgehalten werden.

```
{
    "msg_type":"slot",
    "server":"<IPv4_Addr>",
    "slot":<uint>,
    "offset":<uint>,
    "length":<uint>,
    "period":<uint>,
    "cap":<uint>,
    "cycle":<uint>,
    "tunit":<Index>
}
```

Listing 2: Aufbau der Antwort des Zeitservers beim Abruf eines Zeitschlitzes.

Eine ähnliche Repräsentation der Daten lässt sich auch für die in Kapitel 7.2 beschriebene Zeitsynchronisation definieren (vgl. Listing 3).

```
{
    "msg_type":"syn",
    "t_s":<uint>,
    "t_p":<uint>,
    "t_delay":<uint>,
    "tunit":<Index>
}
```

Listing 3: Struktur der SYN-Response unter Verwendung des verbesserten Algorithmus von Cristian.

Die Nachrichten lassen sich dabei beliebig anpassen, sodass auch andere noch zu definierende Synchronisationsverfahren unterstützt werden können. Ein weiteres Argument für die Nutzung des JSON-Formats ist die native Unterstützung von JSON als Content Format durch die CoAP-Basis-Spezifikation [99].

9 Zusammenfassung

Der Ausgangspunkt dieser Arbeit war die Problemstellung, dass die bisherigen Ansätze für eine zeitlich deterministische Kommunikation den Anforderungen des IIoT bzw. der „Industrie 4.0“-Bewegung nicht gerecht werden. Dies liegt vor allem darin begründet, dass diese sich maßgeblich auf die unteren Schichten des ISO/OSI-Referenzmodells beziehen. Die Folge sind die Verwendung spezialisierter Hardware und proprietärer Protokollanpassungen, welche zu Inkompatibilitäten zwischen den einzelnen Echtzeitlösungen auf der einen und Standardtechnolgien auf der anderen Seite führen. Die Konsequenz daraus ist die Teilung des Netzwerks in unterschiedliche Kommunikationsdomänen, deren Grenzen sich nur über Gateways überwinden lassen. Dies steht aber dem Grundgedanken der totalen horizontalen und vertikalen Integration entgegen. Zudem ergeben sich negative wirtschaftliche Folgen für den Betreiber der Anlagen. Durch die Verwendung proprietärer Hardware entstehen hohe Kosten bei der Anschaffung neuer Geräte, wobei gleichzeitig die Flexibilität bei der Auswahl neuer Geräte erheblich eingeschränkt wird, da sie die richtigen Schnittstellen bereitstellen müssen. Zudem wird das Phänomen des „Vendor Lock-in“, also die Bindung an einen einzigen Hersteller, verstärkt.

Daher beschäftigte sich diese Arbeit mit der zentralen Frage, inwieweit sich eine echtzeitfähige Kommunikation über rein Software-basierte Mechanismen erreichen lässt. Ein besonderer Fokus lag dabei auf der Verwendung weit verbreiteter Standard-Hardware (z.B. Ethernet) und Netzwerk-Protokoll-Stacks (TCP/UDP, IP) sowie der Plattformunabhängigkeit der verwendeten Software, um eine einfach Integration unterschiedlicher Geräte zu ermöglichen. Hierzu wurden zunächst die für das weitere Verständnis der Arbeit notwendigen technischen Grundlagen erläutert. Anschließend wurden mit CoAP und DPWS zwei im IoT-Umfeld weit verbreitete Protokoll-Standards vorgestellt und gegeneinander abgewogen. Dabei stand nicht nur der Funktionsumfang der jeweiligen Protokolle im Vordergrund, sondern auch die prinzipielle Eignung für den Einsatz in Umgebungen mit Echtzeitanforderungen. Im Ergebnis dieser Abwägung wurde das CoAP als geeigneter Kandidat für den weiteren Verlauf dieser Arbeit ausgewählt. Aussschlaggebend hierfür waren zum einen die im direkten Vergleich mit DPWS sehr kleinen Nachrichtengrößen. Zum anderen spielt die standardmäßige Nutzung von UDP auf der Transportschicht eine ausschlaggebende Rolle, da es anders als das bei DPWS verwendete TCP ein deutlich vorhersagbareres Kommunikationsverhalten aufweist. Das darauf folgende Kapitel gab einen Überblick über die historische Entwicklung und den aktuellen Stand der Technik im Bereich der Echtzeitkommunikation. Die Kernpunkte bildeten hierbei aktuell verwendete Feldbus-Systeme sowie Industrial-Ethernet-Lösungen und der TSN Substandard für Ethernet. Weiterhin wurde HaRTKad als eine rein Software-basierte Lösung zur Echtzeitkommunikation vorgestellt. Dabei wurden neben der Funktionsweise auch die Schwachstellen näher untersucht. Diese liegen zum einen in der Plattformabhängigkeit der Umsetzung, welche hemmend auf die Verbreitung von HaRTKad wirkt. Zum anderen müssen Standard-IoT-Protokolle erst in HaRTKad integriert und gegebenenfalls angepasst werden, um eine interoperable und selbstständige Interaktion der Geräte zu ermöglichen.

Denn HaRTKad stellt hierfür keine geeigneten Schnittstellen zur Verfügung und stellt selbst auch keinen etablierten IoT-Standard dar, der unverzichtbare Funktionen, wie zum Beispiel Eventing, unterstützt. Im weiteren Verlauf der Arbeit wurde Schritt für Schritt der Weg hin zu einem alternativen Software-basierten Ansatz zur zeitlich deterministischen Kommunikation allein auf Basis des CoAP-Standards beschrieben. Dabei wurde zunächst das eigens entwickelte jCoAP-Framework beschrieben. Das Ziel von jCoAP war es, eine effiziente und performante Lösung zu schaffen, welche der Geräte-Heterogenität im IoT bzw. IIoT durch die Nutzung von plattformunabhängigen Technologien gerecht wird. Die Entscheidung für eine Umsetzung in Java wurde durch die Plattformunabhängigkeit und die hohe Verfügbarkeit von JVMs für eine hohe Zahl verschiedener Gerätearchitekturen begründet. Ein Vergleich von jCoAP mit der weit verbreiteten CoAP-Implementierung Californium, welche auch als Referenz-Implementierung bei der Eclipse Foundation zum Einsatz kommt, hat gezeigt, dass jCoAP deutlich performanter ist. Allerdings wurde auch offenbar, dass beide Implementierungen starke Schwankungen im Zeitverhalten der Kommunikation aufweisen. Hierfür konnten zwei Ursachen identifiziert werden: die zeitlich nicht-deterministische Verarbeitung der Nachrichten auf den Geräten sowie die zeitlichen Schwankungen bei der Kommunikation über Ethernet. Das darauf folgende Kapitel beschäftigte sich zunächst mit der Protokollverarbeitung auf dem Gerät. Die zentrale Frage dieses Kapitels war, bis zu welcher Ebene im Software-Stack hinunter Veränderungen vorgenommen werden müssen, um ein deterministisches Zeitverhalten zu erreichen. Es konnte gezeigt werden, dass eine echtzeitfähige Verarbeitung ohne Eingriffe in den Protokollstapel erreicht werden kann, hierfür jedoch ein echtzeitfähiges Scheduling auf Betriebssystem-Ebene erforderlich ist. Um dies zu erreichen, wurde im weiteren Verlauf der Arbeit ein präemptiver Linux Kernel als Betriebssystem verwendet, da es sich hierbei um die zur Zeit beste Variante eines freien Open-Source-Betriebsystems mit Echtzeiteigenschaften handelt, welches für eine hohe Anzahl unterschiedlicher Plattformen verfügbar ist.

Kapitel 6 befasste sich erstmals mit dem Zeitverhalten des Nachrichtenaustausches. Um den Einfluss des nicht-deterministischen Verhaltens des Kommunikationsmediums auf das Zeitverhalten der Kommunikation in den Hintergrund treten zu lassen, wurde ein TDMA-basierter Lösungsansatz vorgeschlagen. In diesem Zusammenhang wurden Mechanismen für eine zentralisierte Zeitsynchronisation und das Management der Zeitschlitze vorgestellt und beschrieben, wie diese mit Hilfe von CoAP-inherenten Basisfunktionen umgesetzt werden können. In einer praktische Untersuchung mit realen Geräten in einem Testbed konnte die prinzipielle Eignung der dargestellten Mechanismen zur Erreichung einer echtzeitfähigen Kommunikation gezeigt werden. Anschließend wurde ein Lösungsansatz zur Verteilung der Zeitserver-Funktionalität aus dem vorangegangen Kapitel vorgestellt. Das Ziel war hierbei, das Problem des SPoF bei einem zentralisierten Zeitserver zu beseitigen und so die Robustheit des Systems zu erhöhen. Zugleich sollte durch eine Verteilung der Synchronsationslast auf mehrere Zeitserver die Skalierbarkeit des Systems verbessert werden. In einem abschließenden Kapitel wurde beschrieben, wie sich aus den in dieser Arbeit untersuchten Mechanismen eine Echtzeiterweiterung

für den CoAP-Standard ableiten lässt. Hierbei wurden vor allem ein einheitliches Nachrichtenformat für den Austausch von Steuerungsdaten sowie Anpassungen im Protokollablauf diskutiert.

9.1 Ergebnisdiskussion

Im Ergebnis dieser Arbeit kann die Forschungsfrage, ob eine echtzeitfähige Kommunikation für das IIoT mit einer rein Software-basierten und plattformunabhängigen Lösung erreicht werden kann, bejaht werden. Die praktischen Untersuchungen haben gezeigt, dass sowohl eine zeitlich deterministische Protokollverarbeitung auf Geräteebene als auch ein zeitlich vorhersagbares Verhalten auf Netzwerkebene erzielt werden kann. Im Rahmen der Untersuchungen wurde deutlich, dass hierfür einige Anpassungen des Software-Stapels bis hinunter auf die Betriebssystemebene erforderlich sind. Zwar können Veränderungen auf einer einzelnen Schicht positiven Einfluss auf das Zeitverhalten der Kommunikation haben, allerdings lassen sich auf diese Weise einzelne Ausreißer nicht verhindern. Die notwendigen Anpassungen beziehen sich auf Betriebssystemebene jedoch ausschließlich auf das Scheduling, etwa durch Priorisierung einzelner Prozesse und das Einfügen neuer Unterbrechungspunkte in Kernelmodule des Betriebssystems. Außerdem muss bei der Verwendung von Java eine RTSJ-konforme JVM zum Einsatz kommen, um eine zeitlich deterministische Speicherverwaltung (echtzeitfähige Garbage Collection) zu erreichen. Auf Protokollebene finden nur Veränderungen auf der Anwendungsschicht durch Erweiterung des CoAP-Protokolls statt, welche sich auf Grund der modularen Struktur des CoAP-Standards in einen separaten Substandard auslagern lassen. Somit bleiben die unteren Schichten des Protokollstapels (UDP, IP, Ethernet) unberührt und die Standardkonformität erhalten.

Weiterhin haben die Untersuchungen mit jCoAP in Verbindung mit der RT-JVM JamaicaVM einen Trade-Off zwischen der erreichten mittleren Verarbeitungszeit und der Vorhersagbarkeit der Ausführungszeit offenbart. Durch die Nutzung von Threads mit Echtzeitpriorität und einer echtzeitfähigen JVM stieg die mittlere RTT in den Versuchen um 47,13% an, während die Standardabweichung um 95,17% reduziert wurde. Für den Anstieg der Verarbeitungszeit spielen zwei Faktoren eine Rolle. Zum einen führen die durch die JVM für die Ausführung einzelner Prozesse eingeplanten Zeitpuffer zu einem leichten Anstieg der insgesamt benötigten Zeit. Zum anderen wirkt sich die deterministische GC negativ auf die Ausführungszeit aus. Gegenüber dem Einfluss der GC sind die übrigen Zeitpuffer als vernachlässigbar klein anzusehen. Üblicherweise wird die GC durch die JVM bedarfsorientiert gestartet, je nachdem welche Speichermenge durch nicht mehr benötigte Datenobjekte belegt wird. In der Folge läuft die GC so selten wie möglich, jedoch kann nicht vorhergesagt werden, wann sie gestartet und wie viel Zeit sie in Anspruch nehmen wird. Die RTSJ führt eine deterministische GC ein, um diese Unsicherheit zu beseitigen. Hierbei wird die GC zu fest eingeplanten Zeitpunkten für die immer gleiche Dauer gestartet. Da so keine bedarfsgerechte Speicherfreigabe mehr möglich ist, müssen die für die GC verfügbaren Zeiträume durch den Scheduler der JVM großzügig

geplant werden, andernfalls wird das Volllaufen des Speichers riskiert. Für die Einordnung der Echtzeitfähigkeit eines Systems ist allerdings die zeitliche Vorhersagbarkeit von deutlich höherer Bedeutung als die tatsächliche Ausführungszeit. Zudem ließe sich die Verarbeitungszeit von jCoAP durch Optimierungen deutlich verbessern. In der RTSJ sind beispielsweise Speicherkonzepte beschrieben, die eine GC überflüssig machen. Mit dem sogenannten Scoped Memory können Speicherbereiche definiert werden, die nur bestimmten Prozessen für einen bestimmten Zeitraum zur Verfügung stehen und danach automatisch wieder freigegeben werden. Jedoch handelt es sich dabei um ein rein programmiertechnisches und kein wissenschaftliches Problem. Zudem ist die Nutzung von Scoped Memories mit hohem Aufwand bei der Programmierung und anschließenden Programmtests verbunden, da die Zuweisung der zu nutzenden Speicherbereiche lediglich implizit im Programm-Code erfolgt. Dies macht es für den Entwickler schwierig, zu entscheiden, welcher Prozess zu welchem Zeitpunkt tatsächlich innerhalb eines bestimmten Speicherbereichs agiert. Aus diesen Gründen wurde im Rahmen dieser Arbeit zunächst auf die Nutzung solcher Konzepte verzichtet.

Der in Rahmen dieser Arbeit entwickelte TDMA-Mechanismus auf CoAP-Basis ermöglicht eine zeitlich deterministische Kommunikation zwischen einer Vielzahl von Netzwerkteilnehmern. Dabei ist es nicht von Bedeutung, ob die auf den Schichten 1 bis 4 des ISO/OSI-Modells verwendeten Technologien prinzipiell echtzeitfähig sind. Dies wurde am Beispiel von Standard-Ethernet demonstriert, welches, bedingt durch die von der Netzwerklast abhängigen Switching-Latenzen und das Medienzugriffsverfahren, kein deterministisches Zeitverhalten aufweist. Jedoch gilt dies nur eingeschränkt für drahtlose Kommunikationstechnologien wie WLAN. Hier kann zwar eine Verbesserung des Zeitverhaltens erreicht werden, ein tatsächlicher Determinismus wird allerdings durch die anteilig zufallsbasierte Kanalzugriffszeit und die Anfälligkeit für externe Störungen verhindert. Diese Störungen lassen sich weder vorhersagen noch effektiv unterbinden.

In den durchgeführten Experimenten wurden bisher nur Zykluszeiten von 100 ms verwendet, was lediglich der niedrigsten Echtzeitklasse entspricht. Für den effektiven Einsatz im Umfeld der Industrieautomation müssen jedoch Zykluszeiten von bis zu 1 ms erreicht werden. Die erreichbare Zykluszeit wird hauptsächlich von drei Faktoren begrenzt: der für die Nachrichtenverarbeitung benötigten Zeit, der Genauigkeit der Zeitsynchronisation und der Worst-Case-Ende-zu-Ende-Kommunikationslatenz, welche von der verwendeten Technologie und der Struktur des Verbindungsnetzes abhängt. Wie bereits beschrieben, lässt sich die Verarbeitungszeit für die CoAP-Nachrichten noch deutlich reduzieren. Weiterhin weisen die in Experimenten verwendeten Synchronisationsverfahren eine teilweise sehr geringe Synchronisationsgenauigkeit auf. Allerdings dienten sie lediglich zur Demonstration der prinzipiellen Funktionsfähigkeit des Gesamtkonzeptes in einer Proof-of-Concept-Implementierung. Die Schnittstellen zur Zeitsynchronisation wurden aber so gestaltet, dass neue und genauere Synchronisationsverfahren leicht integriert werden können. Auf diese Weise steht einer Reduktion der Zykluszeit nichts entgegen. Eine Abschätzung der Worst-Case-Latenz lässt sich mit CoAP-eigenen Mechanismen nur schwer erreichen, da auf der Anwendungsebene keine Informationen über die Struktur des

Verbindungsnetzes verfügbar sind. In diesem Bereich können sich jedoch Synergieeffekte mit SDN-basierten Ansätzen und dem neuen TSN-Standard für Ethernet ergeben.

Der in Kapitel 7 vorgestellte Ansatz für einen verteilten Zeitserver soll zum einen die Robustheit des Systems bei Serverausfällen aber auch die Skalierbarkeit des TDMA-Ansatzes erhöhen. Zwar konnte gezeigt werden, dass sich der von der Zeitsynchronisation verursachte Aufwand auf den einzelnen Zeitservern verringert, jedoch ist aufgrund einer fehlenden simulativen Untersuchung für eine größere Anzahl von Netzwerkteilnehmern eine Aussage über die tatsächliche Skalierbarkeit nur schwer möglich. Simulationen sind jedoch häufig selbst mit großen Unsicherheiten verbunden, da es nicht möglich ist, alle in der Realität auftretenden Effekte im Simulationsmodell zu berücksichtigen. Aus diesem Grund wurde in dieser Arbeit die Evaluation in einem Testbed mit realen Geräten gegenüber einem simulativen Ansatz vorgezogen. Um bessere Aussagen über die Skalierbarkeit treffen zu können, sollte zukünftig daher ein größeres Testbed aufgebaut werden.

9.2 Ausblick

In der Ergebnisdiskussion im vorangegangenen Kapitel wurde bereits auf einige noch bestehende Probleme hingewiesen. Zum einen sollen die vorgestellten Mechanismen zukünftig in Netzwerken mit einer höheren Zahl von Netzwerkteilnehmern erprobt werden, um die Skalierbarkeit der Verfahren tiefgehender zu untersuchen. In diesem Zusammenhang sollen auch neue Ansätze zur Auswahl der zusätzlichen Zeitserver und die Zuweisung der Clients zu dem jeweiligen Zeitserver untersucht werden. Bisher werden als zusätzliche Zeitserver diejenigen Server akquiriert, die die Echtzeiterweiterung unterstützen und deren Antwort auf die Discovery-Nachricht des initialen Zeitservers als erstes empfangen wird. Auf diese Weise werden Zeitserver mit einer geringen Latenz zum initialen Zeitserver bevorzugt. Ausgehend von der Annahme, dass die niedrige Latenz daher rührt, dass die Nachrichten weniger Infrastrukturelemente, wie zum Beispiel Switches, durchlaufen müssen, kann geschlussfolgert werden, dass die ausgewählten Zeitserver auch eine gewisse räumliche Nähe zu dem initialen Zeitserver aufweisen (vgl. Abbildung 59 a)).

Dieses Clustering der Zeitserver kann sich auf zweierlei Arten negativ auswirken. Zum einen kann die Robustheit des Systems verringert und ein einzelner Switch zu einem SPoF für das gesamte Netzwerk werden. Zum anderen können die Latenzen zwischen den anderen Netzwerkteilnehmern und dem ihnen jeweils zugewiesenen Zeitserver im Mittel deutlich höher ausfallen als die Latenzen zwischen den Zeitservern. Wie die Erläuterungen zur Zeitsynchronisation gezeigt haben, hängt die Genauigkeit der Synchronisation von der Latenz und insbesondere der Messgenauigkeit der Latenz ab. Entsprechend nimmt die Qualität der Zeitsynchronisation zwischen den Netzwerkteilnehmern und den Zeitservern gegenüber der Synchronisation der Zeitserver untereinander ab. Um dem entgegen zu wirken, könnten Ansätze auf Basis des Software Defined Networking (SDN) zur Akquise der zusätzlichen Zeitserver entwickelt und untersucht werden. SDN bietet den Vorteil, dass bei der Auswahl der Zeitserver Infor-

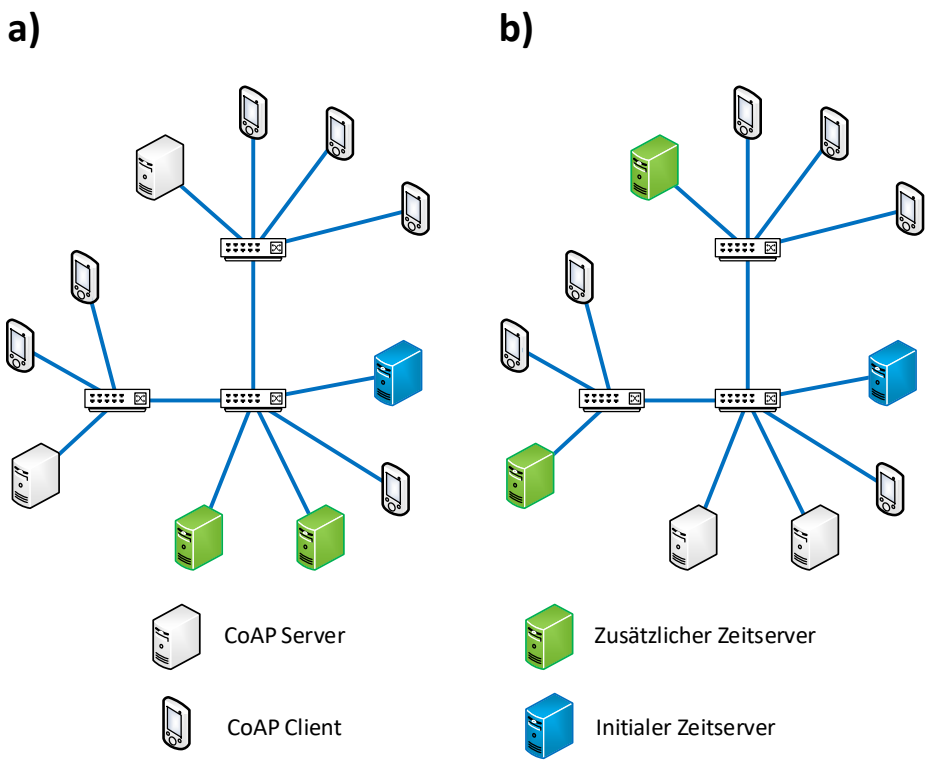

Abbildung 59: Fall einer aus globaler Netzwerksicht ungünstigen Wahl der zusätzlichen Zeitserver (a) und die mit SDN angestrebte Verteilung der zusätzlichen Zeitserver (b).

mationen über die tatsächliche Netzwerk-Topologie miteinbezogen werden könnten. Dies ermöglicht es, bei der Verteilung der aktiven Zeitserver im Netzwerk einen geeigneten Kompromiss zwischen der Zeitserver-Zeitserver- und der Zeitserver-Client-Distanz zu finden (vgl. Abbildung 59 b)). Die Informationen, die mit SDN zur Verfügung stehen, könnten zudem dazu genutzt werden, den anderen Netzwerkteilnehmern den für sie jeweils am nächsten gelegenen Zeitserver zuzuweisen. Dies führt zu im Mittel kürzeren Latenzen der Netzwerkteilnehmer zu ihren Zeitservern und somit zu einer Verbesserung der Synchronisationsgenauigkeit.

Wie in der Ergebnisdiskussion bereits erläutert wurde, ist die Genauigkeit der Zeitsynchronisation ein limitierender Faktor bei der Erreichung möglichst niedriger Zykluszeiten. Daher soll zukünftig der Katalog der zur Verfügung stehenden Verfahren zur Zeitsynchronisation mit genaueren Synchronisationsverfahren erweitert werden. Denkbar wäre hier beispielsweise die Einführung einer an das im Standard IEEE 1588 beschriebene Precision Time Protocol (PTP) angelehnten Zeitsynchronisation.

Ein weiterer Faktor, der einer drastischen Reduktion der Zykluszeit entgegen steht, ist die für die Nachrichtenverarbeitung auf dem jeweiligen Gerät benötigte Zeit. In der Ergebnisdiskussion konnte die von der RTSJ beschriebene deterministische Garbage Collection als maßgebliche Ursache für die vergleichsweise hohe Verarbeitungszeit von jCoAP im Echtzeit-Modus bestimmt werden. Um diese

Zeitkomponente zu minimieren, sollen die innovativen Konzepte der RTSJ zum Speichermanagement (vgl. Scoped Memory) auch in späteren Versionen des jCoAP-Frameworks Verwendung finden.

Abschließend sollten die erarbeiteten Verfahren, Schnittstellen und Datenformate als Entwurf einer eigenen Standard-Erweiterung für CoAP offiziell zur Standardisierung bei der IETF eingereicht werden.

A Abbildungsverzeichnis

B Tabellenverzeichnis

Listings

C Abkürzungsverzeichnis

AVB	Audio-/Video-Bridging
CDMA	Code Division Multiple Access
CoAP	Constrained Application Procotol
COTS	Commercial Off The Shelf
CWND	Congestion Window
DHT	Distributed Hash Table
DPWS	Devices Profile for Web Services
EXI	Efficient XML Interchange
FDMA	Frequency Division Multiple Access
GC	Garbage Collection
H2M	Human to Machine
HTTP	Hypertext Transfer Protocol
IE	Industrial Ethernet
IETF	Internet Engineering Task Force
IIoT	Industrial Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization
ISP	Internet Service Provider
JVM	Java Virtual Machine
M2M	Machine to Machine
MQTT	Message Queue Telemetry Transport
OSI	Open Systems Interconnection
QoS	Quality of Service
REST	Representational State Transfer

RTSJ	Real-Time Specification for Java
RTT	Round Trip Time
RWND	Receiver's Advertised Window
SDMA	Space Division Multiple Access
SDN	Software Defined Networking
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SPoF	Single Point of Failure
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TSN	Time-Sensitive Networking
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WADL	Web Application Description Language
WSDL	Web Service Description Language
XML	Extensible Mark-up Language

D Literaturverzeichnis

[1] *Ethernet adoption in process automation to double by 2016, 2013*, Panel Building & System Integration.

[2] *RFC 791: INTERNET PROTOCOL*. `https://tools.ietf.org/html/rfc791`, 1981. Abruf: März 2018.

[3] *RFC 793: TRANSMISSION CONTROL PROTOCOL*. `https://tools.ietf.org/html/rfc791`, 1981. Abruf: März 2018.

[4] *IEEE Standard for Ethernet*. IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012), Seiten 1–4017, March 2016.

[5] AICAS GMBH: *JamaicaVM 6.3 User Manual - Java Technology for Critical Embedded Systems*, May 2014.

[6] AL-FUQAHA, ALA, MOHSEN GUIZANI, MEHDI MOHAMMADI, MOHAMMED ALEDHARI und MOUSSA AYYASH: *Internet of things: A survey on enabling technologies, protocols, and applications*. IEEE Communications Surveys & Tutorials, 17(4):2347–2376, 2015.

[7] ALLMAN, M, V. PAXSON und E BLANTON: *RFC 5681: TCP Congestion Control*. `https://tools.ietf.org/html/rfc5681`, 2009. Abruf: März 2018.

[8] ALTMANN, VLADO, JAN SKODZIK, PETER DANIELIS, NAM PHAM VAN, FRANK GOLATOWSKI und DIRK TIMMERMANN: *Real-time capable hardware-based parser for efficient xml interchange*. In: *Communication Systems, Networks & Digital Signal Processing (CSNDSP), 2014 9th International Symposium on*, Seiten 395–400. IEEE, 2014.

[9] ASHJAEI, MOHAMMAD, PAULO PEDREIRAS, MORIS BEHNAM, REINDER J BRIL, LUIS ALMEIDA und THOMAS NOLTE: *Response time analysis of multi-hop HaRTES ethernet switch networks*. In: *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*, Seiten 1–10. IEEE, 2014.

[10] ASHJAEI, MOHAMMAD, LUIS SILVA, MORIS BEHNAM, PAULO PEDREIRAS, REINDER J BRIL, LUIS ALMEIDA und THOMAS NOLTE: *Improved message forwarding for multi-hop HaRTES real-time ethernet networks*. Journal of Signal Processing Systems, 84(1):47–67, 2016.

[11] BANKS, ANDREW und RAHUL GUPTA: *MQTT Version 3.1.1*. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Abruf: September 2018.

[12] BELLIARDI, RUDY, BEN BROSGOL, PETER DIBBLE, DAVID HOLMES und ANDY WELLINGS: *The Real-Time Specification for Java*. http://www.rtsj.org/docs/rtsj_1.0.2_spec.pdf, Januar 2006. Abruf: August 2017.

[13] BERGER, SJ: *ARINC 629 digital communication system - application on the 777 and beyond*. Microprocessors and Microsystems, 20(8):463–471, 1997.

[14] BERGMANN, OLAF: *libcoap: C-Implementation of CoAP*. http://libcoap.sourceforge.net. Abruf: September 2018.

[15] BHARADWAJ, RAGHU: *Mastering Linux Kernel Development: A kernel developer's reference manual*. Packt Publishing Ltd., 2017.

[16] BOHN, HENDRIK und FRANK GOLATOWSKI: *Web Services for Embedded Devices.* In: *Embedded Systems Handbook, Second Edition*. CRC Press, 2009.

[17] BORMANN, C. und Z. SHELBY: *RFC 7959: Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. https://tools.ietf.org/html/rfc7959, 2016. Abruf: März 2018.

[18] BRAY, T.: *RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format*. https://tools.ietf.org/html/rfc7159, 2014. Abruf: März 2018.

[19] BUTZIN, BJÖRN, BJÖRN KONIECZEK, FRANK GOLATOWSKI, DIRK TIMMERMANN und CHRISTOPH FIEHE: *Applying the BaaS reference architecture on different classes of devices*. In: *Modelling, Analysis, and Control of Complex CPS (CPS Data), 2016 2nd International Workshop on*, Seiten 1–6. IEEE, 2016.

[20] CENA, GIANLUCA, IVAN CIBRARIO BERTOLOTTI, STEFANO SCANZIO, ADRIANO VALENZANO und CLAUDIO ZUNINO: *Evaluation of EtherCAT distributed clock performance*. Industrial Informatics, IEEE Transactions on, 8(1):20–29, 2012.

[21] CRISTIAN, FLAVIU: *Probabilistic clock synchronization*. Distributed computing, 3(3):146–158, 1989.

[22] DANIELIS, PETER, JAN SKODZIK, VLADO ALTMANN, LENNARD LENDER und DIRK TIMMERMANN: *Dynamic search tolerance at runtime for lookup determinism in the DHT-based P2P network Kad*. In: *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*, Seiten 355–360, Jan 2015.

[23] DANIELIS, PETER, JAN SKODZIK, VLADO ALTMANN, EIKE BJÖRN SCHWEISSGUTH, FRANK GOLATOWSK, DIRK TIMMERMANN und JÖRG SCHACHT: *Survey on Real-Time Com-*

munication Via Ethernet in Industrial Automation Environments. In: *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2014)*, Seiten 1–8, Barcelona, Spain, September 2014.

[24] DAVARI, SADEGH und LUI SHA: *Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions*. ACM SIGOPS Operating Systems Review, 26(2):110–120, 1992.

[25] DAVIS, DOUG, ASHOK MALHOTRA, KATY WARR und WU CHOU: *Web services metadata exchange (ws-metadataexchange)*. World Wide Web Consortium, Recommendation REC-ws-metadata-exchange-20111213, 2011.

[26] DE SMEDT, VALENTIJN, GEORGES GIELEN und WIM DEHAENE: *Temperature-and Supply Voltage-Independent Time References for Wireless Sensor Networks*. Springer, 2015.

[27] DEERING, S. und R. HINDEN: *Internet Protocol, Version 6 (IPv6) Specification*. `https://tools.ietf.org/html/rfc8200`, 2017. Abruf: März 2018.

[28] DOMINGO, MARI CARMEN: *An overview of the Internet of Things for people with disabilities*. Journal of Network and Computer Applications, 35(2):584–596, October 2012.

[29] ENSTE, UDO und JOCHEN MÜLLER: *Datenkommunikation in der Prozessindustrie: Darstellung und anwendungsorientierte Analyse*. Oldenbourg Industrieverlag, 2007.

[30] ETHERCAT TECHNOLOGY GROUP: *Ethercat - Technical introduction and overview*. `https://www.ethercat.org/en/technology.html`. Abruf: August 2017.

[31] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE: *ETSI CTI Plugtests Guide First Draft V0.0.15*. `http://www.etsi.org/plugtests/CoAP/About_CoAP.htm`, February 2012. Abruf: Juli 2017.

[32] EVANS, PETER C. und MARCO ANNUNZIATA: *Industrial Internet: Pushing the Boundaries of Minds and Machines*. In: *General Electric Tech. Report*, November 2012.

[33] EZECHINA, MA, KK OKWARA und CAU UGBOAJA: *The Internet of Things (Iot): A Scalable Approach to Connecting Everything*. The International Journal of Engineering and Science, 4(1):09–12, 2015.

[34] FARSI, M, K RATCLIFF und MANUEL BARBOSA: *An introduction to CANopen*. Computing & Control Engineering Journal, 10(4):161–168, 1999.

[35] FAYYAD-KAZAN, HASAN, LUC PERNEEL und MARTIN TIMMERMAN: *LinuxPREEMPT-RT vs. commercial RTOSs: how big is the performance gap?* GSTF Journal on Computing (JoC), 3(1), 2018.

[36] FELBER, PASCAL, PETER KROPF, ERYK SCHILLER und SABINA SERBU: *Survey on Load Balancing in Peer-to-Peer Distributed Hash Tables.* IEEE Communications Surveys and Tutorials, 16(1):473–492, 2014.

[37] FELSER, MAX: *Real-time ethernet-industry prospective*. Proceedings of the IEEE, 93(6):1118–1129, 2005.

[38] FETTWEIS, GERHARD P.: *The Tactile Internet: Applications and Challenges*. Vehicular Technology Magazine, IEEE, 9, Issue: 1:64–70, March 2014.

[39] FIELDING, ROY THOMAS: *Architectural styles and the design of network-based software architectures*. Doktorarbeit, University of California, Irvine, 2000.

[40] FOROUZAN, A BEHROUZ: *Data communications & networking (sie)*. Tata McGraw-Hill Education, 2006.

[41] FYSARAKIS, KONSTANTINOS, IOANNIS ASKOXYLAKIS, OTHONAS SOULTATOS, IOANNIS PAPAEFSTATHIOU, CHARALAMPOS MANIFAVAS und VASILIOS KATOS: *Which iot protocol? comparing standardized approaches over a common m2m application*. In: *Global Communications Conference (GLOBECOM), 2016 IEEE*, Seiten 1–7. IEEE, 2016.

[42] GABALE, VIJAY, BHASKARAN RAMAN, KAMESWARI CHEBROLU und PURUSHOTTAM KULKARNI: *LiT MAC: Addressing the Challenges of Effective Voice Communication in a Low Cost, Low Power Wireless Mesh Network*. In: *Proceedings of the First ACM Symposium on Computing for Development*, ACM DEV '10, Seiten 5:1–5:11, New York, NY, USA, 2010. ACM.

[43] GARCIA, NUNO M und JOEL JOSE PC RODRIGUES: *Ambient assisted living*. CRC Press, 2015.

[44] GLEIXNER, THOMAS: *The realtime preemption patch (PREEMPT_RT) - Concepts and mainline integration*. `ftp://ftp.polsl.pl/pub/linux/kernel/people/tglx/preempt-rt/rtlws2006.pdf`, 2006. Abruf: September 2018.

[45] GOMEZ, CARLES und JOSEP PARADELLS: *Wireless home automation networks: A survey of architectures and technologies*. IEEE Communications Magazine, 48(6):92–101, 2010.

[46] GRÜNER, STEN, JULIUS PFROMMER und FLORIAN PALM: *RESTful industrial communication with OPC UA*. IEEE Transactions on Industrial Informatics, 12(5):1832–1841, 2016.

[47] GUIDO MORITZ, FRANK GOLATOWSKI: *IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) and Constrained Application Protocol (CoAP)*. In: *Industrial Communication Technology Handbook, Second Edition*. CRC Press, August 2014.

[48] HAMZA, HAMZA und STEVE COUNSELL: *Scoped Memory in RTSJ Applications Dynamic Analysis of Memory Consumption*. In: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, Seiten 221–225. IEEE, 2011.

[49] HARTKE, K.: *RFC 7641: Observing Resources in the Constrained Application Protocol (CoAP)*. `https://tools.ietf.org/html/rfc7641`, 2015. Abruf: März 2018.

[50] HU, TIANLIANG, PENG LI, CHENGRUI ZHANG und RILIANG LIU: *Design and application of a real-time industrial Ethernet protocol under Linux using RTAI*. International Journal of Computer Integrated Manufacturing, 26(5):429–439, 2013.

[51] HUNKELER, URS, HONG LINH TRUONG und ANDY STANFORD-CLARK: *MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks*. In: *3rd international conference on Communication systems software and middleware and workshops (comsware 2008)*, Seiten 791–798. IEEE, January 2008.

[52] IEEE 802.11 PRESENTATIONS: *What 802.11 is doing*. `http://www.ieee802.org/11/presentation.html`, 2015. Abruf: Mai 2018.

[53] INTEL CORPORATION: *Intel® Galileo-Mainboard Schaltplan*. `https://www.intel.com/content/dam/support/us/en/documents/galileo/sb/galileo_schematic.pdf`. Abruf: September 2018.

[54] KAGERMANN, HENNING, WOLF-DIETER LUKAS und WOLFGANG WAHLSTER: *Industrie 4.0: Mit dem Internet der Dinge auf dem Weg zur 4. industriellen Revolution*. VDI nachrichten, 13(11), 2011.

[55] KASPARICK, MARTIN, BENJAMIN BEICHLER, BJÖRN KONIECZEK, ANDREAS BESTING, MICHAEL RETHFELDT, FRANK GOLATOWSKI und DIRK TIMMERMANN: *Measuring latencies of IEEE 11073 compliant service-oriented medical device stacks*. In: *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*, Seiten 8640–8647. IEEE, 2017.

[56] KASPARICK, MARTIN, STEFAN SCHLICHTING, FRANK GOLATOWSKI und DIRK TIMMERMANN: *Medical DPWS: New IEEE 11073 standard for safe and interoperable medical device communication*. In: *Standards for Communications and Networking (CSCN), 2015 IEEE Conference on*, Seiten 212–217. IEEE, 2015.

[57] KLASEN, FRITHJOF, VOLKER OESTREICH und MICHAEL VOLZ: *Industrial Communication with Fieldbus and Ethernet*. VDE-Verlag, 2011.

[58] KLINGER, ANDREAS und DAVID FRANZ: *Echtzeit- und Deadline-Scheduling von Linux*. `https://www.embedded-software-engineering.de/echtzeit-und-deadline-scheduling-von-linux-a-554154/`, April 2016. Abruf: September 2018.

[59] KONIECZEK, BJÖRN, MARTIN KASPARICK, STEFAN SCHUSTER, MICHAEL RETHFELDT, FRANK GOLATOWSKI und DIRK TIMMERMANN: *Towards a TDMA-based Real-Time Extension for the Constrained Application Protocol*. In: *12th IEEE World Conference on Factory Communication Systems (WFCS)*, Aveiro, Portugal, May 2016.

[60] KONIECZEK, BJÖRN, MICHAEL RETHFELDT, FRANK GOLATOWSKI und DIRK TIMMERMAN: *A Distributed Time Server for the Real-Time Extension of CoAP*. In: *19th IEEE Symposium on Real-Time Distributed Computing (ISORC)*, Seiten 84–91, York, UK, May 2016.

[61] KONIECZEK, BJÖRN, MICHAEL RETHFELDT, FRANK GOLATOWSKI und DIRK TIMMERMANN: *Real-Time Communication for the Internet of Things using jCoAP*. In: *18th IEEE Symposium on Real-Time Computing (ISORC)*, Auckland, New Zealand, April 2015.

[62] KONIECZEK, BJÖRN, JAN SKODZIK, PETER DANIELIS, VLADO ALTMANN, MICHAEL RETHFELDT und DIRK TIMMERMANN: *HaRTKad: A P2P-based concept for deterministic communication and its limitations*. In: *Computers and Communication (ISCC), 2016 IEEE Symposium on*, Seiten 1157–1162. IEEE, 2016.

[63] KOVATSCH, MATTHIAS: *A Low-Power CoAP for Contiki*. In: *8th IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*, Seiten 855–860, Valencia, Italy, October 2011.

[64] KOVATSCH, MATTHIAS: *CoAP for the web of things: from tiny resource-constrained devices to the web browser*. In: *UbiComp '13 Adjunct, Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, Seiten 1495–1504, Zurich, Switzerland, September 2013.

[65] KOVATSCH, MATTHIAS, MARTIN LANTER und ZACH SHELBY: *Californium: Scalable Cloud Services for the Internet of Things with CoAP*. In: *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, 2014.

[66] KUMAR, PANKAJ: *Java (JVM) Memory Model - Memory Management in Java*. `https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java`, April 2018. Abruf: September 2018.

[67] KUROSE, JAMES F. und KEITH ROSS: *Computer Networking - A Top-Down Approach*. Pearson Education, 6 Auflage, 2013.

[68] KYUSAKOV, RUMEN, PABLO P. PEREIRA, JENS ELIASSON und JERKER DELSING: *EXIP: A Framework for Embedded Web Development*. ACM Transactions on the Web (TWEB), Volume 8 Issue 4(23), October 2014.

[69] LENG, QUAN, YI-HUNG WEI, SONG HAN, ALOYSIUS K MOK, WENLONG ZHANG und MASAYOSHI TOMIZUKA: *Improving control performance by minimizing jitter in RT-WiFi networks*. In: *2014 IEEE Real-Time Systems Symposium (RTSS)*, Seiten 63–73. IEEE, 2014.

[70] LERCHE, CHRISTIAN, KLAUS HARTKE und MATTHIAS KOVATSCH: *Industry adoption of the Internet of Things: A constrained application protocol survey*. In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, Seiten 1–6. IEEE, 2012.

[71] LERCHE, CHRISTIAN, NICO LAUM, GUIDO MORITZ, ZEEB ELMAR, FRANK GOLATOWSKI und DIRK TIMMERMANN: *Implementing powerful Web Services for highly resource-constrained devices*. In: *Pervasive Computing and Communication Workshops (PerCom Workshops), IEEE International Conference*, Seattle, WA, USA, March 2011.

[72] LEVÄ, TAPIO, OLEKSIY MAZHELIS und HENNA SUOMI: *Comparing the cost-efficiency of CoAP and HTTP in Web of Things applications*. Decision Support Systems, 63:23–28, July 2014.

[73] MAYMOUNKOV, PETAR und DAVID MAZIERES: *Kademlia: A peer-to-peer information system based on the xor metric*. In: *International Workshop on Peer-to-Peer Systems*, Seiten 53–65. Springer, 2002.

[74] MELZER, INGO: *Service-orientierte Architekturen mit Web Services: Konzepte-Standards-Praxis*. Springer-Verlag, 2010.

[75] MODI, VIPUL und DEVON KEMP: *Web Services Dynamic Discovery (WS-Discovery) Version 1.1*. `http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf`, 2009. Abruf: März 2018.

[76] MORITZ, GUIDO, DIRK TIMMERMANN, REGINA STOLL und FRANK GOLATOWSKI: *Encoding and compression for the devices profile for web services*. In: *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*, Seiten 514–519. IEEE, 2010.

[77] NEUMANN, ARNE, LUKASZ WISNIEWSKI, RAKASH SIVASIVA GANESAN, PETER ROST und JÜRGEN JASPERNEITE: *Towards integration of Industrial Ethernet with 5G mobile networks*. In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, Seiten 1–4. IEEE, 2018.

[78] NOF, SHIMON Y: *Springer handbook of automation*. Springer Science & Business Media, 2009.

[79] ODVA: *The organization that supports network technologies built on the common industrial protocol (cip) - device net, ethernet/ip, componet, and controlnet*. `http://www.odva.org`. Abruf: Februar 2014.

[80] ORACLE: *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*. `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/sizing.html`. Abruf: September 2018.

[81] PAPAZOGLOU, MIKE P und WILLEM-JAN VAN DEN HEUVEL: *Service oriented architectures: approaches, technologies and research issues*. The VLDB journal, 16(3):389–415, 2007.

[82] PFEIFFER, JONAS H, MARTIN KASPARICK, BENJAMIN STRATHEN, CHRISTIAN DIETZ, MAX E DINGLER, TIM C LUETH, DIRK TIMMERMANN, KLAUS RADERMACHER und FRANK GOLATOWSKI: *OR. NET RT: how service-oriented medical device architecture meets real-time communication*. Biomedical Engineering/Biomedizinische Technik, 63(1):81–93, 2018.

[83] PIGAN, RAIMOND und MARK METTER: *Automating with PROFINET: Industrial Communication Based on Industrial Ethernet*. Publicis Publishing, 2008.

[84] POSTEL, J.: *RFC 768: User Datagram Protocol*. `https://tools.ietf.org/html/rfc768`, 1980. Abruf: März 2018.

[85] PUTTNIES, HENNING, BJÖRN KONIECZEK, JAKOB HELLER, DIRK TIMMERMANN und PETER DANIELIS: *Algorithmic approach to estimate variant software latencies for latency-sensitive networking*. In: *Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2016 IEEE 7th Annual*, Seiten 1–7. IEEE, 2016.

[86] QIAN, KUN, FENGYUAN REN, DANFENG SHAN, WENXUE CHENG und BO WANG: *XpressEth: Concise and efficient converged real-time Ethernet*. In: *Quality of Service (IWQoS), 2017 IEEE/ACM 25th International Symposium on*, Seiten 1–6. IEEE, 2017.

[87] RAHMAN, A. und E. DIJK: *RFC 7390: Group Communication for the Constrained Application Protocol (CoAP)*. `https://tools.ietf.org/html/rfc7390`. Abruf: März 2018.

[88] RAMAMRITHAM, KRITHI: *Real-time databases*. Distributed and parallel databases, 1(2):199–226, 1993.

[89] RECOMMENDATION, ITUTX: *200 (1994)— ISO/IEC 7498-1: 1994*. Information technology–Open systems interconnection–Basic reference model: The basic model, 1994.

[90] RODRIGUES, LUIS, JOEL GUERREIRO und NOÉLIA CORREIA: *RELOAD/CoAP architecture with resource aggregation/disaggregation service*. In: *Personal, Indoor, and Mobile Radio*

Communications (PIMRC), 2016 IEEE 27th Annual International Symposium on, Seiten 1–6. IEEE, 2016.

[91] ROSTAN, MARTIN: *Industrial Ethernet Technologies: Overview*. In: *Tech Report 2011*. EtherCAT Technology Group, 2011.

[92] SANTOS, RUI, MORIS BEHNAM, THOMAS NOLTE, PAULO PEDREIRAS und LUÍS ALMEIDA: *Multi-level hierarchical scheduling in ethernet switches*. In: *Proceedings of the ninth ACM international conference on Embedded software*, Seiten 185–194. ACM, 2011.

[93] SCHLESINGER, R und A SPRINGER: *VABS - A new approach for Real Time Ethernet*. In: *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, Seiten 4506–4511. IEEE, 2013.

[94] SCHLIMMER, J: *A technical introduction to the devices profile for web services*. Microsoft Corporation, Web Services Technical Articles, 2004.

[95] SCHMIDT, KLAUS WERNER und ECE GÜRAN SCHMIDT: *Distributed real-time protocols for industrial control systems: Framework and examples*. Parallel and Distributed Systems, IEEE Transactions on, 23(10):1856–1866, 2012.

[96] SCHNELL, GERHARD und WD DOSE: *Bussysteme in der Automatisierungs-und Prozesstechnik: Grundlagen, Systeme und Anwendungen der industriellen Kommunikation*. Wiesbaden, Vieweg+ Teubner Verlag, 2012.

[97] SCHREINER, RÜDIGER: *Computernetzwerke: von den Grundlagen zur Funktion und Anwendung*. Carl Hanser Verlag GmbH Co KG, 2016.

[98] SEIKO EPSON CORPORATION: *kHz RANGE CRYSTAL UNIT FC-135R FC-135/FC-255 - Brief Sheet*. `https://support.epson.biz/td/api/doc_check.php?dl=brief_FC-255&lang=en`. Abruf: September 2018.

[99] SHELBY, Z., K. HARTKE und C. BORMANN: *RFC 7252: The Constrained Application Protocol*. `https://tools.ietf.org/html/rfc7252`. Abruf: März 2018.

[100] SKODZIK, JAN, VLADO ALTMANN, PETER DANIELIS, ARNE WALL und DIRK TIMMERMANN: *A kad prototype for time synchronization in real-time automation scenarios*. In: *WTC 2014; World Telecommunications Congress 2014; Proceedings of*, Seiten 1–6. VDE, 2014.

[101] SKODZIK, JAN, PETER DANIELIS, VLADO ALTMANN, BJÖRN KONIECZEK, EIKE BJÖRN SCHWEISS GUTH, FRANK GOLATOWSKI und DIRK TIMMERMANN: *CoHaRT: Deterministic Transmission of Large Data Amounts using CoAP and Kad*. In: *IEEE Internatilnal Conference on Industrial Technology 2015 (IEEE ICIT)*, Sevilla, Spain, March 2015.

[102] SKODZIK, JAN, PETER DANIELIS, VLADO ALTMANN, JENS ROHRBECK, DIRK TIMMERMANN, THOMAS BAHLS und DANIEL DUCHOW: *DuDE: A distributed computing system using a decentralized P2P environment*. In: *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*, Seiten 1048–1055. IEEE, 2011.

[103] SKODZIK, JAN, PETER DANIELIS, VLADO ALTMANN und DIRK TIMMERMANN: *Time synchronization in the DHT-based P2P network Kad for real-time automation scenarios*. In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a*, Seiten 1–6. IEEE, 2013.

[104] SKODZIK, JAN, PETER DANIELIS, VLADO ALTMANN und DIRK TIMMERMANN: *HaRTKad: A Hard Real-Time Kademlia Approach*. In: *11th IEEE Consumer Communications & Networking Conference (CCNC)*, Seiten 566–571, 2014.

[105] STUTZBACH, DANIEL und REZA REJAIE: *Improving Lookup Performance Over a Widely-Deployed DHT*. In: *INFOCOM*, Seiten 1–12, 2006.

[106] TANENBAUM, ANDREW S. und DAVID J. WETHERALL: *Computernetzwerke*. Pearson Deutschland GmbH, 2012.

[107] VERHAPPEN, IAN und AUGUSTO PEREIRA: *Foundation Fieldbus*. ISA, 2008.

[108] WEI, YI-HUNG, QUAN LENG, SONG HAN, ALOYSIUS K MOK, WENLONG ZHANG, MASAYOSHI TOMIZUKA, TIANJI LI, DAVID MALONE und DOUGLAS LEITH: *RT-WiFi: Real-time high speed communication protocol for wireless control systems*. ACM SIGBED Review, 10(2):28–28, 2013.

[109] WEISER, MARK: *The Computer for the 21 st Century*. Scientific american, 265(3):94–105, 1991.

[110] WHITE, GREG, JAAKKO KANGASHARJU, DON BRUTZMAN und STEPHEN WILLIAMS: *Efficient XML Interchange Measurements Note*. `https://www.w3.org/TR/exi-measurements/`, 2007. Abruf: März 2018.

[111] WOLLSCHLAEGER, MARTIN, THILO SAUTER und JUERGEN JASPERNEITE: *The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0*. IEEE Industrial Electronics Magazine, 11(1):17–27, 2017.

[112] YOON, IL-CHUL, ALAN SUSSMAN, ATIF MEMON und ADAM PORTER: *Direct-dependency-based software compatibility testing*. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, Seiten 409–412. ACM, 2007.

[113] YOUNG, STEPHEN JOHN: *Real time languages - Design and Development*. Ellis Horwood Limited, 1982.