UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR TELEMATIK

EIN NAMENSZENTRISCHER ANSATZ ZUR REALISIERUNG VON DIENSTEN IM INTERNET DER DINGE

DISSERTATION

DIPL.-INF. TORSTEN TEUBLER

Dipl.-Inf. Torsten Teubler

Aus dem Institut für Telematik der
Universität zu Lübeck
Institutsdirektor: Prof. Dr. Stefan Fischer

EIN NAMENSZENTRISCHER ANSATZ ZUR REALISIERUNG VON DIENSTEN IM INTERNET DER DINGE

Inauguraldissertation zur Erlangung der Doktorwürde der
Universität zu Lübeck – Sektion Informatik/Technik

vorgelegt von Dipl.-Inf. Torsten Teubler
aus Deggendorf

Lübeck, 2018

Prüfungskommission:

Vorsitzender:	Prof. Dr. Philipp Rostalski
1. Berichterstatter:	Prof. Dr. Horst Hellbrück
2. Berichterstatter:	Prof. Dr. Erik Maehle

Tag der mündlichen Prüfung: 09. April 2019

Zum Druck genehmigt. Lübeck, den 25. April 2019

VORWORT

Die vorliegende Dissertation ist größtenteils im Rahmen meiner Anstellung an der Fachhochschule Lübeck im Fachbereich Elektrotechnik und Informatik entstanden. Dort war ich am CoSA Kompetenzzentrum für Kommunikation, Systeme und Anwendungen als wissenschaftlicher Mitarbeiter tätig. Ihren Ursprung hat diese Arbeit in den Forschungsprojekten Real-World G-Lab und DataCast. Hier wurde die Integration von ressourcenbeschränkten Geräten in das Internet beziehungsweise ein inhaltszentrisches Kommunikationsparadigma für das Internet untersucht. Diese Forschungsprojekte waren meine persönliche Motivation, die Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge zu untersuchen.

Ohne die Unterstützung und Motivation meiner Kollegen des CoSA Kompetenzzentrums wäre die Fertigstellung dieser Arbeit nicht möglich gewesen. Daher möchte ich mich an dieser Stelle ganz herzlich bei ihnen bedanken. Mein Dank gebührt auch den studentischen Hilfskräften, die mich während meiner Zeit an der Fachhochschule tatkräftig unterstützt haben. Insbesondere mein ehemaliger Kommilitone Dr. Sebastian Ebers hat mich während der Erstellung dieser Arbeit fachlich und persönlich beraten. Vielen Dank dafür.

Mein besonderer Dank gilt meinem Betreuer und ehemaligen Vorgesetzten Prof. Dr. Horst Hellbrück, der diese Arbeit erst möglich gemacht hat. Ich danke ihm für die konstruktiven Diskussionen, die wertvollen Anregungen, die ausführlichen Rückmeldungen zu meinen Entwürfen und natürlich für die abschließende Begutachtung der Arbeit. Weiterhin danke ich ihm ganz herzlich für die gute Zusammenarbeit und sein Vertrauen während meiner Zeit an der Fachhochschule.

Abschließend möchte ich bei meinen Freunden, Verwandten und der Familie für ihre Unterstützung, ihre Motivation und ihr Verständnis bedanken und mich bei ihnen dafür entschuldigen, dass ich während der Erstellung der Arbeit oft abwesend war.

Tettnang, im April 2019
Torsten Teubler

KURZFASSUNG

In der Entwicklung des Internets zeigen sich zwei wesentliche Trends, die das Potential haben, das Internet in der Zukunft zu verändern: der inhaltszentrische Ansatz (engl. Content-Centric Networking (CCN)) und das Internet der Dinge (engl. Internet of Things (IoT)). Bei CCN werden Daten bzw. Inhalte mit Namen adressiert und nicht wie bei dem heutigen knotenzentrischen Kommunikationsparadigma Geräte bzw. Knoten. Im IoT sind vernetzte, internetfähige Alltagsgegenstände bzw. Dinge Teil des Internets.

Mit dem namenszentrischen Ansatz zur Realisierung von Diensten im Internet der Dinge sieht diese Arbeit das inhaltszentrische gegenüber dem knotenzentrischen Paradigma im IoT vor. Merkmale von CCN wie das Zwischenspeichern von Daten im Netz, das flexible Knotenmodell und die frei wählbaren Namen eröffnen dem IoT neue Potenziale. In dieser Arbeit werden schwerpunktmäßig ressourcenbeschränkte Geräte im IoT wie z. B. Sensorknoten betrachtet. Ressourcenbeschränkte Geräte sind im heterogenen IoT der bestimmende Faktor für ein durchgängiges Kommunikationsparadigma.

Neben den Potenzialen der Anwendung von CCN im IoT gibt es Herausforderungen: *(i)* CCN ist für die Verteilung von Inhalten im Internet konzipiert, nicht für die Entwicklung von Diensten im IoT. Der konzeptionelle Unterschied ist eine Herausforderung bei der Anwendung von CCN im IoT. *(ii)* Gegenüber dem vertrauten knotenzentrischen Paradigma ist das Entwickeln von Diensten mit dem inhaltszentrischen Paradigma eine Herausforderung. *(iii)* Lange, sprechende Namen von CCN sind für ressourcenbeschränkte Geräte im IoT aufgrund der Verarbeitung und ihres Ressourcenverbrauchs bzgl. Speicher und Nachrichten eine Herausforderung. *(iv)* Das Messaging Pattern von CCN sieht eine starre Abfolge von Nachrichten Anfragen (Interest) und Antwort (Content) vor. Das Fehlen von Messaging Pattern z. B. für periodische Updates von Sensorwerten, ist eine Herausforderung für die Anwendung von CCN im IoT.

Diese Arbeit schlägt Maßnahmen zum Umgang mit den Herausforderungen vor: beispielsweise CCN-IoT – eine CCN-Implementierung für das inhaltszentrische Internet der Dinge, eine auf CCN-IoT aufbauende Dienstarchitektur mit werkzeugunterstützter Codegenerierung zur Entwicklung von Diensten, Strategien zur effizienten Verarbeitung sowie zur Verkürzung und effizienten Repräsentation von Namen. Flexible Dienste nutzen die Gültigkeitsdauer von Nachrichten um den vielfältigen Anforderungen der Anwendungen umzusetzen. Außerdem werden Daten im Nachrichtenkopf transportiert. Somit lassen sich beliebige Messaging Pattern realisieren.

Zusammenfassend verbessern die in dieser Arbeit entwickelten Maßnahmen die Entwicklung von Diensten im inhaltszentrischen IoT. CCN-IoT und die werkzeugunterstützte Codegenerierung sind Wegbereiter für die namenszentrischen Dienste. Damit ist die zukünftige Basis für eine effiziente Entwicklung von Anwendungen in einem inhalts-/namenszentrischen Internet der Dinge mit seinen heterogenen Plattformen geschaffen. Die Effektivität der Maßnahmen wird entweder anhand von Beispielen belegt bzw. durch Messungen und Simulationen evaluiert.

INHALTSVERZEICHNIS

Kurzfassung vii

Inhaltsverzeichnis ix

Abkürzungsverzeichnis xiii

Glossar xvii

1 Einführung 1
- 1.1 Beiträge der Arbeit . . . 5
- 1.2 Aufbau der Arbeit . . . 9

2 Grundlagen 11
- 2.1 Internet . . . 11
- 2.2 Internet der Dinge . . . 14
  - 2.2.1 Drahtlose Sensornetze . . . 16
  - 2.2.2 Technologien für Heimautomation . . . 21
  - 2.2.3 Message Queue Telemetry Transport . . . 22
- 2.3 Dienst und Anwendung . . . 23
  - 2.3.1 Dienstanbieter, Dienstnutzer, Anfrage und Antwort . . . 24
  - 2.3.2 Anwendung und Anwender . . . 25
- 2.4 Inhaltszentrischer Ansatz . . . 26
  - 2.4.1 CCN-Namen . . . 26
  - 2.4.2 Knotenmodell . . . 28
  - 2.4.3 CCN-Datenstrukturen . . . 28
  - 2.4.4 Nachrichtenverarbeitung . . . 29
- 2.5 Zusammenfassung . . . 32

3 Namenszentrische Dienste 33
- 3.1 Verwandte Arbeiten . . . 33
- 3.2 Inhaltszentrische Ansätze im Internet der Dinge . . . 35
  - 3.2.1 Herausforderungen . . . 36
  - 3.2.2 Potenziale . . . 45
- 3.3 Konzept . . . 50
  - 3.3.1 Dienste . . . 50
  - 3.3.2 Dienstmethode . . . 54
  - 3.3.3 Dienstbeschreibung . . . 60
- 3.4 Zusammenfassung . . . 60

4 Architektur und Implementierung 63
4.1 Verwandte Arbeiten 63
4.2 Architektur 64
4.2.1 Systemarchitektur 65
4.2.2 Softwarearchitektur 71
4.3 Implementierung 76
4.3.1 Puffer für Namen und Nachrichten 76
4.3.2 Nachrichtenverarbeitung 82
4.3.3 Evaluation Nachrichtenverarbeitung 84
4.3.4 CCN-Datenstrukturen 88
4.3.5 Speicheroptimierung der FIB 89
4.3.6 Evaluation Speicheroptimierung der FIB 90
4.4 Zusammenfassung 97

5 Dienstbeschreibung und Werkzeugunterstützung 99
5.1 Verwandte Arbeiten 100
5.1.1 JSON-Grundlagen 101
5.2 Schema der Dienstbeschreibung 101
5.3 Entwicklung von Diensten 109
5.3.1 Werkzeug zu Codegenerierung 109
5.3.2 Architektur und Implementierung 110
5.3.3 Evaluation Werkzeugunterstützung 115
5.4 Zusammenfassung 119

6 Namen für Dienste 121
6.1 Verwandte Arbeiten 121
6.2 Lebenszyklusmodell 123
6.3 Konzept für Namen 123
6.4 Namen mit besonderer Bedeutung 125
6.4.1 Root Name 126
6.4.2 Link-Local Name 126
6.4.3 Standardnamen 127
6.5 Namen und Gateways 129
6.5.1 Name Solicitation Service 129
6.5.2 Verkürzung von Namen 131
6.6 Zusammenfassung 137

7 Basisdienste 139
7.1 Simple-Radio Service 140
7.2 Unicast Faces und Backpath-Radio 142
7.3 Neighborhood Service 145
7.4 ID Service 148
7.5 Basisdienste auf dem Gateway 150
7.6 Dienste zur Authentifizierung 151
7.7 Zusammenfassung 154

8 Zusammenfassung und Ausblick 157

A Anhang 161
A.1 Implementierung 161
A.1.1 Typ-Längen-Felder 161
A.1.2 Buffer-Implementierungen 162
A.1.3 Buffer-Iteratoren 167
A.1.4 CCN-Datenstrukturen in CCN-IoT 169
A.2 NDN-Testbed 173
A.2.1 Abbildungscharakteristik 173
A.2.2 Länge der Präfixe 175
A.3 Dienstbeschreibung und Werkzeugunterstützung 176
A.3.1 Vollständiges Schema der Dienstbeschreibung 176
A.3.2 Dienstbeschreibung Beispiel 178
A.3.3 Implementierung der Datentypen und Methoden in Description 182
A.3.4 Implementierung der Codegeneratoren 183

Abbildungsverzeichnis 189

Tabellenverzeichnis 193

Quelltextverzeichnis 195

Publikationsverzeichnis 197

Literaturverzeichnis 201

ABKÜRZUNGSVERZEICHNIS

µIP
MicroIP (eine 6LoWPAN Implementierung)

API
Application Programming Interface

BNF
Backus-Naur-Form

CCN
Content-Centric Networking

CCNx
Content-Centric-Networking-Implementierung des Palo Alto Research Centers

CoAP
Constrained Application Protocol

CS
Content Store

DHCP
Dynamic Host Configuration Protocol

DNS
Domain Name System

DODAG
Destination Oriented Directed Acyclic Graph

FIB
Forwarding Information Base

HTTP
Hypertext Transfer Protocol

ICMP
Internet Control Message Protocol

ICMPv6
Internet Control Message Protocol for IPv6

IETF
Internet Engineering Task Force

IoT
Internet of Things

IP
Internet Protocol

IPv4
Internet Protocol Version 4

IPv6
Internet Protocol Version 6

IRC
Internet Relay Chat

JSON
Java Script Object Notation

JSON-WSP
JSON Web Service Protocol

LAN
Local Area Network

LISP
Locator/Identifier Separation Protocol

M2M
Machine-to-Machine (dt. Maschine-zu-Maschine)

MAC
Media Access Control

MEP
Message Exchange Pattern (auch Messaging Pattern)

MQTT
Message Queue Telemetry Transport

NDN
Named Data Networking

OSPF
Open Shortest Path First

PIT
Pending Interest Table

REST
REpresentional State Transfer

RFC
Request For Comments

RFID
Radio-Frequency Identification

ROLL
Routing Over Low power and Lossy networks

RPC
Remote Procedure Call

RPL
Ripple; Routing-Protokoll für drahtlose Sensornetze

RSVP
Resource reSerVation Protocol

SCTP
Stream Control Transmission Protocol

SMTP
Simple Mail Transfer Protocol

SOA
Service-Oriented Architecture

SOAP
Simple Object Access Protocol

STL
Standard Template Library

TCP
Transmission Control Protocol

TLD
Top Level Domain

TLV
Type-Length-Value

UDDI
Universal Description Discovery and Integration

UDP
User Datagram Protocol

UML
Unified Modeling Language

URI
Uniform Resource Identifier

URL
Uniform Ressource Locator

WADL
Web Application Description Language

WLAN
Wireless Local Area Network

GLOSSAR

6LoWPAN
6LoWPAN steht für *IPv6 over Low power Wireless Personal Area Network* und ist eine Portierung von IPv6 für die drahtlose Vernetzung von ressourcenbeschränkten Geräten. Seiten: 5, 18, 19, 35, 36, 41, 42

Ad-hoc-Netz
In einem *Ad-hoc-Netz* sind die Geräte nicht über eine vorher installierte Infrastruktur vernetzt. Die Geräte bauen die Infrastruktur, meist über Funk, selbstständig auf. Seiten: 16, 17, 36, 65, 76, 142, 145–147

CCN-IoT
CCN-IoT ist die Implementierung des inhaltszentrischen Ansatzes für ressourcenbeschränkte Geräte im Internet der Dinge, die im Rahmen der Arbeit entstanden ist. Seiten: 6, 7, 63, 64, 71, 72, 74–76, 78, 79, 82–85, 87, 88, 90, 91, 93, 190

Clean-Slate
Clean-Slate bedeutet wörtlich „unbeschriebene Tafel“ oder „Neuanfang“. In der Forschung am Internet der Zukunft sieht ein Clean-Slate Ansatz die grundlegende Änderung der Architektur des Internets vor. Seiten: 3, 35

Content Object
Das *Content Object* ist ein Nachrichtentyp des inhaltszentrischen Ansatzes. Content Objects werden mit Interests angefragt und transportieren Daten. Seiten: 28–32

Content Store
Der *Content Store* ist eine Datenstruktur, die bei Implementierungen des inhaltszentrischen Ansatzes verwendet wird und Content Objects speichert. Seiten: 28–32

Content-Centric Networking
Bei *Content-Centric Networking* werden Inhalte anstelle von Endgeräten im Netz adressiert. Das wird als *inhaltszentrischer Ansatz* bezeichnet. Seite: 2

Ende-zu-Ende
Bei dem *Ende-zu-Ende*-Paradigma sind Endgeräte miteinander verbunden, wie beispielsweise beim Telefonnetz. Datennetze folgen dem *Ende-zu-Ende*-Paradigma, wenn nur die Endgeräte den Zustand einer Verbindung speichern. Seiten: 1, 2, 4, 5, 22, 38, 39, 46, 49

Ethernet

Ethernet ist eine Technologie, die Hardware und Protokolle für drahtgebundene Datenübertragung spezifiziert. Seiten: 13, 21

Evolutionärer Ansatz

In der Forschung am Internet der Zukunft werden Ansätze, die Internettechnologien iterativ verbessern, ohne dabei die grundlegende Architektur des Internets zu ändern, als *evolutionäre* Ansätze bezeichnet. Seite: 3

Face

Ein *Face* ist im Kontext des inhaltszentrischen Ansatzes eine Verallgemeinerung eines Interfaces. Seiten: 28–31

Forwarding Information Base

Die *Forwarding Information Base* ist eine Datenstruktur, die bei Implementierungen des inhaltszentrischen Ansatzes verwendet wird. Sie dient als Weiterleitungstabelle für Interests. Seiten: 28–30, 89–96

Gateway

Ein *Gateway* stellt einen Übergang zwischen zwei verschiedenartigen Netzen her. Gateways realisieren den Übergang zwischen verschiedenen Protokollen oder Übertragungstechnologien. In dieser Arbeit realisieren Gateways insbesondere den Übergang zwischen drahtgebundenen und drahtlosen Netzen. Seiten: 14, 15, 18, 22, 23, 48, 64–66, 123, 129–137, 150, 151

IEEE 802.15.4

IEEE 802.15.4 ist ein Funkstandard für drahtlose Sensornetze. Seiten: 16, 42

Inhaltszentrischer Ansatz

siehe *Content-Centric Networking*. Seiten: 2–4, 37–39

Interest

Der *Interest* ist ein Nachrichtentyp des inhaltszentrischen Ansatzes. Mit Interests werden Content Objects angefragt. Seiten: 28–31

Internet of Things, dt. Internet der Dinge

Der Begriff wird in dieser Arbeit als eine Erweiterung des Internets um Geräte, die in Gegenständen des alltäglichen Lebens eingebaut sind, verstanden. Seiten: 3–11, 14, 16, 18, 19, 21–23

Internet Protocol, dt. Internet Protokoll

Das *Internet Protokoll* bildet zusammen mit anderen Protokollen die Basis für die Kommunikation im heutigen Internet. Es erlaubt die Adressierung von Endpunkten (siehe IP-Adresse) und einen verbindungslosen Austausch von Nachrichten. Das Internet Protokoll existiert aktuell in den Versionen 4 (IPv4) und 6 (IPv6). Seiten: 1, 13, 23

IP-Adresse

IP-Adressen werden zur Adressierung von Geräten oder Gruppen von Geräten in Netzwerken eingesetzt, die auf dem Internet Protokoll (IP) basieren. Ein Gerät kann mehrere IP-Adressen haben. Seiten: 13, 14, 37, 38, 41, 46, 48, 55, 128

ISO/OSI-Referenzmodell
Das *ISO/OSI-Referenzmodell* ist ein abstraktes Modell für Protokolle in Computernetzen. Seiten: 11–13, 18

Knotenzentrischer Ansatz
Computernetze, in denen die beteiligten Kommunikationspartner (Knoten) mit einer eindeutigen Adresse (z. B. IP-Adresse) adressiert werden, werden in dieser Arbeit als *knotenzentrisch* bezeichnet. Seiten: 2, 4, 5, 8, 19, 21, 22, 31, 34, 36–38

Messaging Pattern
Der Begriff *Messaging Pattern* oder auch Message Exchange Pattern (MEP) bedeutet frei übersetzt „Nachrichtenaustauschmuster". Dabei handelt es sich um eine Abfolge von Nachrichten, die durch ein Protokoll vorgegeben ist. Seiten: 5, 37, 42–45, 48, 49

Named Data Networking
Named Data Networking ist eine Implementierung von *Content-Centric Networking*. Seiten: 2, 92, 93, 122, 173

Pending Interest Table
Die *Pending Interest Table* ist eine Datenstruktur, die bei Implementierungen des inhaltszentrischen Ansatzes verwendet wird und Interests speichert. Seiten: 28–31

Router
Ein *Router* ist ein Gerät zur Weiterleitung von Paketen in einem paketorientierten Netz. Ein Router arbeitet nur auf der Vermittlungs- bzw. Internetschicht des ISO/OSI- bzw. TCP/IP-Referenzmodells und grenzt sich somit von einem Gateway ab. Seiten: 1, 3, 15, 19, 38, 39, 48, 65

Service Oriented Architecture, dt. Serviceorientierte Architektur
Bei einer *serviceorientierten Architektur* handelt es sich um ein Architekturparadigma von Anwendungen, die mittels Diensten (engl. Services) in einem verteilten System (lokales Netz, Internet) realisiert werden. Dienste sind Software, die eine definierte Funktionalität über das Netz bereitstellen. Seiten: 4, 6, 23, 33, 61, 100

Simple Object Access Protocol
Das *Simple Object Access Protocol* ist ein vom World Wide Web Konsortium standardisiertes Protokoll zum Austausch von Daten und für entfernte Methodenaufrufe. Seiten: 33, 42

TCP/IP
Mit dem Begriff *TCP/IP* wird die Familie der Internetprotokolle bezeichnet. Seiten: 1, 2, 14, 15, 18, 22, 23

TCP/IP-Referenzmodell
Das *TCP/IP-Referenzmodell* ist ein Modell, dem konkrete Implementierungen von Netzwerkstapeln im Internet folgen. Aus dem TCP/IP-Referenzmodell ist das ISO/OSI-Referenzmodell als eine Verallgemeinerung hervorgegangen. Seiten: 12, 13

KAPITEL 1

EINFÜHRUNG

KAUM eine Technologie hat unser Leben so stark in so kurzer Zeit beeinflusst, wie das *Internet*. Mitte der 60er Jahre des letzten Jahrhunderts wurde das Internet für militärische Zwecke entworfen und in den 70er Jahren für die akademische Forschung freigegeben. In den 90er Jahren hat es sich in weiteren Lebensbereichen etabliert und innerhalb von zwei Dekaden die Art der Kommunikation der Menschen tiefgreifend verändert.

Die Kommunikation über das Internet ist ein komplexes Zusammenspiel von verteilter Hard- und Software. Dieses Zusammenspiel funktioniert durch Regeln zur Kommunikation. Solche Regeln werden als *Protokolle* bezeichnet. Protokolle definieren das Format der Daten bei der Kommunikation (Syntax) und das Verhalten der an der Kommunikation beteiligten Geräte (Semantik). Die Protokolle, die die Basis für die Kommunikation im Internet bilden, sind das Internet Protocol (IP), das Transmission Control Protocol (TCP) als verbindungsorientiertes Protokoll und das User Datagram Protocol (UDP) als verbindungsloses Protokoll. Diese Protokolle bilden die Familie der Internetprotokolle und werden unter dem Namen TCP/IP zusammengefasst.

Die TCP/IP-Protokollfamilie basiert auf Überlegungen aus den 60er und 70er Jahren und bildet auch heute noch die Grundlage des Internets. Ein wesentliches Entwurfskriterium von TCP/IP ist das *Ende-zu-Ende-Paradigma*, was bedeutet, dass einzig die Endgeräte den Zustand der Verbindung speichern. Bei TCP/IP werden die Nachrichten im Netz nur weitergeleitet, was eine einfachere Hardware im Netz und eine Unabhängigkeit von der Übertragungstechnologie erlaubt. Leiner et al. schrieben dazu im Jahre 2009 in „A Brief History of the Internet“ [94], dass die grundlegende Idee des Internets die Verbindung von beliebig konzeptionierten Netzwerken ist. Beliebig konzeptioniert bedeutet in diesem Zusammenhang unterschiedliche Übertragungstechnologien, wie drahtgebundene terrestrische oder drahtlose satellitengestützte Netze. Weiterhin heißt es bei Leiner et al., dass Router als „Black Boxes“ konzipiert werden, die keine Information über den Paketfluss oder die Verbindung speichern. Das macht die Router einfacher und robuster gegenüber Fehlern.

Das Internet mit seinem Ende-zu-Ende-Paradigma erfüllte die Anforderungen in seiner Anfangszeit. Eine Anforderung war beispielsweise das Teilen von knappen Ressourcen wie Bandlaufwerke oder Großrechner, wie Jacobson et al. in „Networking Named Content“ [86] schreiben. Für populäre Anwendungen in der Anfangszeit des Internets wie E-Mail oder das File Transfer Protocol (FTP), die ihren Ursprung in den Jahren 1971 [152] beziehungsweise 1985 [133] hatten, reichte das Ende-zu-Ende-Paradigma aus.

Jacobson et al. führen ebenso aus, dass sich die Anforderungen bezüglich der Kommunikation im Internet im Laufe der Zeit geändert haben. Heute tritt die Kommunikation mit Rechnern über das Internet zugunsten von Daten oder Inhalten in den Hintergrund. So spielt der Zugriff auf knappe Ressourcen, wie die oben genannten Bandlaufwerke oder Großrechner, keine Rolle mehr, dagegen wächst die Menge der Inhalte im Internet rasant. Nach einer Schätzung, erschienen in dem Artikel „THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East“ von John Gantz und David Reinsel [76] aus dem Jahre 2012 wird die Menge der Inhalte von heute 20 000 Exabytes (Stand 2017/2018) auf 40 000 Exabytes im Jahr 2020 angewachsen sein. (Hinweis: Ein Exabyte sind 10^{18} Byte.)

Mit den Servern, die die Inhalte bereitstellen, wird heute nach wie vor mit TCP/IP und damit nach dem Ende-zu-Ende-Paradigma kommuniziert. Für den Anwender spielt es hingegen keine Rolle, von welchem Rechner der Inhalt bezogen wird. Wichtig für den Anwender ist der Inhalt sowie der Ursprung und die Authentizität des Inhalts. Der Ursprung des Inhalts bezeichnet keinen bestimmten Rechner, sondern eine Instanz (Person, Firma, etc.), die den Inhalt erzeugt und/oder veröffentlicht hat und für die Authentizität des Inhalts bürgt. Dieser Instanz vertraut der Anwender, dass der Inhalt authentisch ist. Im heutigen knotenzentrischen Internet werden nicht die Inhalte an sich authentifiziert, sondern die Rechner, von denen die Inhalte bereitgestellt werden. HTTPS, beschrieben in RFC 2818 [136], ist ein Beispiel für ein knotenzentrisches Verfahren, welches Authentifizierung nutzt.

Daten- oder *inhaltszentrische Ansätze* wie Content-Centric Networking (CCN), im Jahre 2007 von Jacobson et al. in einem Whitepaper [85] vorgestellt, oder auch Named Data Networking (NDN) [178] von Zhang et al. stellen einen Paradigmenwechsel weg vom Ende-zu-Ende-Paradigma dar. Bei inhaltszentrischen Ansätzen werden nicht Rechner, sondern Inhalte mit Namen adressiert. Die Adressierung von Inhalten ermöglicht eine einfachere verteilte Speicherung der Inhalte im Netz, was laut Jacobson et al. den Vorteil hat, dass die Inhalte näher bei den Anwendern und somit für die Anwender schneller verfügbar sind.

Weiterhin realisieren inhaltszentrische Ansätze Authentizität auf Basis des Inhalts und nicht wie beim Ende-zu-Ende-Paradigma über eine Verbindung zwischen zwei Rechnern. Inhalte werden beim inhaltszentrischen Ansatz am Ursprung signiert und beim Empfang ihre Authentizität überprüft. Das Ende-zu-Ende-Paradigma setzt hingegen voraus, dass der Rechner, der den Inhalt bereitstellt auch unter der Kontrolle der Instanz steht, die den Inhalt erzeugt hat beziehungsweise für die Authentizität des Inhalts bürgt.

Inhaltszentrische Ansätze werden zu den sogenannten „Clean-Slate“-Ansätzen gezählt. Clean-Slate bedeutet wörtlich „unbeschriebene Tafel“ oder, weniger bildlich ausgedrückt, Neuanfang. Ein Neuanfang ist es deshalb, da Clean-Slate-Ansätze eine Architektur des Internets vorschlagen, die zu der heutigen Architektur grundlegend verschieden ist. Roberts [138] und Pan et al. [124] geben in ihren Studien einen Überblick über Clean-Slate-Ansätze für ein zukünftiges Internet.

Von den Clean-Slate-Ansätzen werden die „evolutionären“ Ansätze abgegrenzt, die bestehende Internettechnologien iterativ verbessern, ohne dabei mit der grundlegenden Architektur zu brechen. Jennifer Rexford und Constantine Dovrolis stellen in „Future Internet Architecture: Clean-Slate Versus Evolutionary Research“ [137] Clean-Slate- und evolutionäre Ansätze gegenüber und diskutieren die Vor- und Nachteile.

Ein Beispiel für einen evolutionären Ansatz ist IPv6 (siehe RFC 2460 [62]). IPv6 ist der designierte Nachfolger des IPv4. Bei IPv6 gibt es theoretisch 2^{128} Adressen, da die Adressen 128 Bit lang sind. Im Gegensatz dazu sind es bei IPv4 nur 2^{32} Adressen, wobei wie bei IPv6 nicht der komplette Adressbereich genutzt wird. Durch einen einfacheren Aufbau des Nachrichtenkopfs (engl. Header) bei IPv6 ist der Rechenaufwand auf den Routern geringer als bei IPv4 und es gibt in IPv6 bereits einfache Mechanismen zur automatischen Adresskonfiguration, was die Ausbringung von Geräten vereinfacht.

Unabhängig von der Diskussion um Clean-Slate-Ansätze und im Speziellen dem inhaltszentrischen Ansatz hat sich im Zuge der fortschreitenden Miniaturisierung von Computern das *„Internet der Dinge“* (engl. Internet of Things (IoT)) etabliert. Der Begriff Internet der Dinge wurde bereits im Jahre 1999 von Kevin Ashton [32] geprägt, wie Ashton selbst in dem im Jahre 2009 erschienen Artikel „That ‘Internet of Things’ Thing“ schreibt. Nach heutigem Verständnis ist das Internet der Dinge eine Erweiterung des Internets mit mikroelektronischen Geräten, wie beispielsweise in dem Buch „Internet of Things (IoT): Ein Wegweiser durch das Internet der Dinge“ [114] von Stefan Müller beschrieben. Diese mikroelektronischen Geräte sind mit Mitteln zur Kommunikation ausgestattet und werden in andere elektronische Geräte oder Gegenstände – zusammengefasst als *Dinge* – integriert.

Beispiele für das Internet der Dinge sind vernetzte Geräte im Haushalt wie Thermostate an Heizungen oder Klimaanlagen, die über das Internet ferngesteuert werden oder sich, basierend auf Informationen aus dem Internet, selbstständig regeln. Weitere Anwendungen des Internet der Dinge gibt es im industriellen Bereich. Benardos und Vosniakos [41] sehen die Vorteile eines Internet der Dinge im industriellen Bereich darin, dass Daten von den Dingen (Sensoren oder Maschinen) direkt bei der Produktion, wo sie entstehen, als auch im Internet für die Planung und Steuerung von Prozessen verwendet werden. Da Xu et al. [60] zeigen in Ihrer Studie weitere Anwendungen des Internet der Dinge in der Industrie, beispielsweise in der Intra- und Interlogistik, in der Nahrungsmittelversorgung und im Bergbau. Sprenger und Engemann [153] gehen davon aus, dass die Dinge zu Akteuren werden, wenn sie auf Grundlage der (im gesamten Internet der Dinge) gesammelten Daten selbstständig Entscheidungen treffen.

Die Arbeit von Da Xu et al. sowie alle anderen oben aufgeführten Arbeiten, Studien und Bücher gehen davon aus, dass das Internet der Dinge dem Ende-zu-Ende-Paradigma folgt. Diese Arbeit verfolgt einen anderen Ansatz, der sich in heutige Konzepte und zukünftige Entwicklungstrends wie in Abbildung 1.1 dargestellt eingliedert. In Abgrenzung zu den inhaltszentrischen Ansätzen werden Ansätze, die dem Ende-zu-Ende-Paradigma folgen, in dieser Arbeit als *knotenzentrisch* bezeichnet.

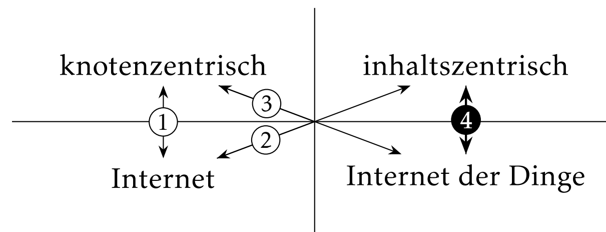


Abbildung 1.1 – *Einordnung der Arbeit in die Paradigmen und Entwicklungen des Internets*

Das heutige Internet ist knotenzentrisch (1). Neben dem knotenzentrischen Internet gibt es den Entwicklungstrend des inhaltszentrischen Internets (2). Das Internet der Dinge wird gegenwärtig knotenzentrisch entworfen (3). Der inhaltszentrische Ansatz und das Internet der Dinge sind beide wichtige Forschungsgebiete. In dieser Arbeit wird ein *inhaltszentrisches Internet der Dinge* betrachtet, Entwicklungstrend (4) in Abbildung 1.1.

Motivation für diese Arbeit sind dabei große Potenziale, die sich bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge ergeben. So ermöglichen Namen eine flexible Art der Adressierung. Das Knotenmodell des inhaltszentrischen Ansatzes orientiert sich nicht am hierarchischen Schichtenmodell für knotenzentrische Netzwerkprotokolle, sondern an einem flachen Modell. Dieses flache Knotenmodell des inhaltszentrischen Ansatzes erlaubt ein flexibles Design von lose gekoppelten Anwendungen, die ein erweitertes Aufgabenspektrum als die serviceorientierten Architekturen (engl. Service-Oriented Architecture (SOA)) abdecken. Mit dem flachen Knotenmodell lassen sich Dienste implementieren, die nicht zum Aufgabenspektrum von SOA gehören, wie beispielsweise ein Routing-Mechanismus, der in Kapitel 7 dargestellt wird. Erläuterungen zu SOA finden sich bei Nicolai Josuttis in seinem SOA-Manifest [88] oder in seinem Buch „SOA in Practice: The Art of Distributed System Design“ [89].

Neben den Potenzialen gibt es auch Herausforderungen bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge. So ist der inhaltszentrische Ansatz für die Verteilung von Daten entwickelt worden, jedoch nicht für die Entwicklung von Diensten. Dienste werden im Internet der Dinge aber eine bedeutende Rolle spielen. Diese Meinung vertreten unter anderem Da Xu et al. [60] in ihrer Studie mit dem Titel „Internet of Things in Industries: A Survey“. Da Xu et al. sehen in serviceorientierten Architekturen für das Internet der Dinge eine Schlüsseltechnologie, um die heterogenen Geräte des Internet der Dinge wechselseitig zu integrieren. Dieser Auffassung schließt sich diese Arbeit an.

Eine weitere Herausforderung ist, dass der inhaltszentrische Ansatz Inhalte mit Namen variabler Länge adressiert. Namen sind in den Nachrichten und in den Weiterleitungstabellen auf den Knoten enthalten. Nachrichten, die die Inhalte transportieren, werden auf den Knoten zwischengespeichert. Das Speichern der Namen benötigt somit Arbeitsspeicher, der auf vielen Geräten im Internet der Dinge eine beschränkte Ressource ist.

Ebenso eine Herausforderung ist das starre Messaging Pattern des inhaltszentrischen Ansatzes, das eine starre Abfolge von Anfragen und Antworten vorsieht. Möglicherweise ist das knotenzentrische Ende-zu-Ende-Paradigma, das flexiblere Messaging Pattern erlaubt, für das Internet der Dinge besser geeignet.

Die Nutzung der Potenziale und Lösungsvorschläge für die Herausforderungen der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge bilden die Grundlage für die Beiträge, die im folgenden Abschnitt vorgestellt werden.

1.1 BEITRÄGE DER ARBEIT

Die Arbeit enthält insgesamt sieben Beiträge. In der folgenden Darstellung werden die Beiträge in der Abfolge aufgeführt, wie sie in der Arbeit behandelt werden.

Der **erste Beitrag** der Arbeit ist eine **Diskussion über die Kompatibilität ressourcenbeschränkter Geräte im Internet der Dinge mit zukünftigen inhaltszentrischen Ansätzen**. Ressourcenbeschränkte Geräte sind beispielsweise drahtlose Sensorknoten, die nur mit einem Mikrocontroller ausgestattet und batteriebetrieben sind.

Bei der Betrachtung der Kompatibilität stehen ressourcenbeschränkte Geräte im Fokus, da sie als schwächste Vertreter in einem heterogenen Internet der Dinge der bestimmende Faktor für die Entwicklung eines einheitlichen Kommunikationsparadigmas sind. Ein einheitliches Kommunikationsparadigma wird favorisiert, denn es erleichtert die Integration von Geräten und Diensten. Aus dem Grunde schlagen die Befürworter eines knotenzentrischen Internet der Dinge das an IPv6 angelehnte 6LoWPAN [165] vor, da es so aus ihrer Sicht ein einheitliches Kommunikationsparadigma für ein knotenzentrisches Internet gäbe.

Die Diskussion gliedert sich dabei in die Teile **Herausforderungen** und **Potenziale**. Insgesamt werden vier Herausforderungen identifiziert. Lange Namen oder das starre Messaging Pattern des inhaltszentrischen Ansatzes sind beispielsweise Herausforderungen für die Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge.

Den Herausforderungen stehen Potenziale des Einsatzes von inhaltszentrischen Ansätzen im Internet der Dinge gegenüber. Identifizierte Potenziale sind das flexible Knotenmodell und die frei wählbaren Namen des inhaltszentrischen Ansatzes. Die Potenziale des inhaltszentrischen Ansatzes versprechen flexiblere Anwendungen im Internet der Dinge.

Zusammengefasst handelt es sich bei dem ersten Beitrag einerseits um eine kritische Auseinandersetzung zum Einsatz inhaltszentrischer Paradigmen im Internet der Dinge, wie sie in der aktuellen Literatur bisher nicht, oder nicht in der Ausführlichkeit wie hier, beschrieben wurden. Andererseits werden Argumente und Lösungen zum Einsatz von inhaltszentrischen Ansätzen im Internet der Dinge dargestellt, die das Konzept von inhaltszentrischen Ansätzen erweitern.

Der **zweite Beitrag** dieser Arbeit führt das neue Konzept der **„namenszentrischen Dienste“** ein. Namenszentrische Dienste wurden erstmals der Veröffentlichung „Name-Centric Service Architecture for Cyber-Physical Systems (Short Paper)“ (Hellbrück, Teubler, Fischer) [9] vorgestellt. Das Konzept der namenszentrischen Dienste wird aus den Potenzialen entwickelt, die in der Diskussion über die Kompatibilität ressourcenbeschränkter Geräte im Internet der Dinge mit zukünftigen inhaltszentrischen Ansätzen identifiziert wurden. Die identifizierten Potenziale zielen nicht mehr nur auf mit Namen adressierte Daten oder Inhalte ab, sondern stellen mit Namen adressierte *Dienste* in den Vordergrund. Außerdem wird in diesem Beitrag das Konzept der *Dienstmethode* (engl. Service Method), bekannt aus den serviceorientierten Architekturen, auf namenszentrische Dienste übertragen und erweitert.

Gegenüber den serviceorientierten Architekturen bedienen die namenszentrischen Dienste ein breiteres Spektrum an Aufgaben. Teilaspekte des erweiterten Aufgabenspektrums der namenszentrischen Dienste wurde in drei Veröffentlichungen behandelt, die im Rahmen der Arbeit entstanden sind. In „Efficient Data Aggregation with CCNx in Wireless Sensor Networks“ (Teubler et al.) [6] wurde ein Verfahren zur Datenaggregation in drahtlosen Sensornetzen mit namenszentrischen Diensten vorgestellt. „A Solution for the Naming Problem for Name-Centric Services“ (Teubler et al.) [5] stellt namenszentrische Dienste zur Konfiguration von Namen im Internet der Dinge vor und „Architecture and Message Processing for Name-Centric Services in Wireless Sensor Networks“ (Teubler, Hellbrück) [3] zeigt Beispiele für lose gekoppelte namenszentrische Dienste.

Zusammenfassend führt der zweite Beitrag in das neue Konzept namenszentrischer Dienste ein. Es werden Gemeinsamkeiten und Unterschiede des Dienstkonzeptes und Dienstbegriffs von namenszentrischen Diensten und serviceorientierten Architekturen erörtert. Basierend auf dem Konzept führt dieser Beitrag die namenszentrischen Dienste ein und bildet somit eine Überleitung zur Architektur.

Der **dritte Beitrag** der Arbeit beschreibt die **Architektur des Gesamtsystems und die Implementierung der Teilsysteme** [3]. Der Fokus dieses Beitrags liegt auf der Beschreibung der Systemschnittstellen mit Hilfe der Unified Modeling Language (UML) [139].

Bei der Systemarchitektur werden die Schnittstellen zwischen den Geräten und Diensten beschrieben. Die Softwarearchitektur beschreibt die Schnittstellen zwischen den namenszentrischen Diensten und *CCN-IoT*. CCN-IoT ist ein leichtgewichtiges, CCNx nachempfundenes inhaltszentrisches Protokoll für drahtlose

Sensorknoten, welches im Rahmen dieser Arbeit entstanden ist. CCN-IoT orientiert sich an CCN-WSN [135] von Zhong et al., wobei sich CCN-IoT noch exakter am Design von CCNx orientiert.

Neben der Architektur werden auch ausgewählte Implementierungsdetails von CCN-IoT vorgestellt, wie das durchgängige, aufeinander aufbauende und effiziente Format für die Darstellung von Namen, Nachrichten und Dienstmethoden. Namen, Nachrichten und Dienstmethoden basieren auf einer sequenziellen, *Buffer* genannten Datenstruktur. Bei der effizienten Verarbeitung von Buffern spielt das Entwurfsmuster des Iterators eine besondere Rolle.

Außerdem werden alternative ressourcenschonende Implementierungen der Datenstruktur Forwarding Information Base (FIB), die eine Kernkomponente von CCNx ist, entworfen und umgesetzt. Diese alternativen Implementierungen wurden in der Veröffentlichung „Memory Efficient Forwarding Information Base for Content-Centric Networking" (Teubler, Hellbrück) [1] erstmals vorgestellt. Ziel der alternativen Implementierungen ist die Verringerung des Speicherverbrauchs auf den Knoten.

Die Architektur ist neu, da weder die Arbeit von Jacobson et al. [85] noch die von Zhang et al. [178] eine Architektur und die Schnittstellen der Komponenten Dienste beschreiben. Die Darstellung der Schnittstellen zeigt die Beziehung zwischen den namenszentrischen Diensten und der inhaltszentrischen Netze.

Der **vierte Beitrag** der Arbeit ist die Entwicklung einer Beschreibung für namenszentrische Dienste, kurz **Dienstbeschreibung** genannt, auf der Basis des JSON-Formats [183], die um ein Werkzeug für Codegenerierung ergänzt wird. Hierbei werden Konzepte aus der Veröffentlichung „Tool Chain for Application Development with Name-Centric Services" (Teubler, Hellbrück) [4] vorgestellt und weiterentwickelt.

Die Dienstbeschreibung basiert auf der die Idee der menschen- und maschinenlesbaren Web-Service Description Language (WSDL), ist aber leichtgewichtiger als WSDL und damit für ressourcenbeschränkte Geräte im zukünftigen Internet der Dinge geeignet. Wie WSDL beschreibt die hier vorgestellte Dienstbeschreibung die Schnittstelle von Diensten.

Das Werkzeug zur Codegenerierung ist ein Hilfsmittel für Entwickler von namenszentrischen Diensten für das Internet der Dinge, da sie aus einer Dienstbeschreibung Anwendungscodes zur Nutzung und zur Entwicklung von namenszentrischen Diensten generieren. Die Konzepte von Dienstbeschreibung und Werkzeugunterstützung sind nicht neu. Da es keine Implementierung gibt, die den Anforderungen im inhaltszentrischen Internet der Dinge genügt, wurde ein Werkzeug entwickelt.

Der **fünfte Beitrag** der Arbeit behandelt **Namen für Dienste** und schlägt eine Lösung für das Namensproblem namenszentrischer Dienste (Teubler et al.) [5] für das namenszentrische Internet der Dinge vor. Die bisherigen Beiträge haben nur die Syntax von Namen behandelt, jedoch nicht die Semantik. In dieser Arbeit wird ein *Lebenszyklusmodell* für namenszentrische Dienste vorgeschlagen, von dem die Semantik für Namen abgeleitet wird. Das Lebenszyklusmodell

unterteilt den Prozess von der Entstehung bis zur Ausbringung von namenszentrischen Diensten in eine definierte Folge von Prozessabschnitten. In jedem Prozessabschnitt wird der Name erweitert.

Der **sechste Beitrag** der Arbeit führt die sogenannten **Basisdienste** ein. Basisdienste sind namenszentrische Dienste, die Grundfunktionalitäten realisieren, die nicht Teil von CCN-IoT sind. CCN-IoT ermöglicht nur die Kommunikation von namenszentrischen Diensten, die sich auf einem Knoten befinden. Eine Kommunikation über Knotengrenzen hinweg wird wiederum mit namenszentrischen Basisdiensten realisiert. Namenszentrische Basisdienste wurden auf der NetSys 2017 im Rahmen der Demonstration mit dem Titel „Name-Centric Services for the Internet of Things“ (Teubler, Hellbrück) [2] einem Fachpublikum vorgestellt.

Neben der Kommunikation von Diensten auf unterschiedlichen Knoten verbindet ein *Gateway-Dienst* drahtlose und drahtgebundene Netze. Der Gateway-Dienst wird auf Knoten ausgeführt, die mit Hardware für drahtlose und drahtgebundene Kommunikation ausgestattet sind. Gateway-Dienste nehmen beispielsweise eine Paketkonvertierung vor, wenn die verbundenen Netze unterschiedliche Paketformate verlangen.

Die Realisierung der Grundfunktionalitäten mittels namenszentrischer Dienste in diesem Beitrag erlaubt die Kommunikation der Knoten untereinander flexibel an die heterogenen Bedingungen im Internet der Dinge anzupassen. Bei den knotenzentrischen Ansätzen für das Internet der Dinge hingegen werden Annahmen über Netztopologien und den Nachrichtenfluss getroffen. Das schränkt laut „A Critical Evaluation of the IPv6 Routing Protocol for Low Power and Lossy Networks (RPL)“ von Clausen et al. [57] die Flexibilität der knotenzentrischen Ansätze ein. Namenszentrische Dienste zur Realisierung der Grundfunktionalitäten folgen dem im zweiten Beitrag entwickelten Konzept und werden mit der Werkzeugunterstützung aus dem vierten Beitrag implementiert.

Die **Evaluation** stellt den **siebten Beitrag** dieser Arbeit dar und basiert maßgeblich auf den im Rahmen dieser Arbeit entstanden Veröffentlichungen. Teile der Evaluation werden immer passend zum jeweiligen Kontext in der Arbeit präsentiert.

In der Veröffentlichung „Architecture and Message Processing for Name-Centric Services in Wireless Sensor Networks“ [3] wird die Verarbeitungsgeschwindigkeit von Namen und Nachrichten auf ressourcenbeschränkten Geräten evaluiert. Insbesondere wird hier der Vorteil, den Iterator in einen schnellen, nur lesenden und einen langsameren, modifizierenden Iterator aufzuteilen, quantitativ belegt.

Die Evaluation in „Memory Efficient Forwarding Information Base for Content-Centric Networking“ (Teubler, Hellbrück) [1] zeigt die Speichereffizienz der alternativen Implementierungen der Forwarding Information Base (FIB). Für die auf Bloom-Filtern basierenden Implementierungen werden die Rahmenbedingungen gezeigt, unter der die Implementierungen speichereffizienter sind.

In der Veröffentlichung „Tool Chain for Application Development with Name-Centric Services“ [4] wird eine neue Methode zur quantitativen Evaluation der werkzeugunterstützten Codegenerierung vorgeschlagen und angewendet. Bei

der Evaluation wird im ersten Schritt als Metrik die zyklomatische Komplexität angewendet, um zu zeigen, dass der von dem Werkzeug generierte Code eine maßgebliche Komplexität besitzt. Um die Effizienz der Werkzeugunterstützung zu messen, werden im zweiten Schritt die logischen Codezeilen der Dienstbeschreibung mit denen des generierten Codes verglichen, um so die Einsparung an Codezeilen zu messen.

1.2 AUFBAU DER ARBEIT

Nach dieser Einführung folgen die Grundlagen in Kapitel 2. Die Grundlagen führen wichtige Begriffe und verwandte Arbeiten zum Themenkomplex Internet und Internet der Dinge ein, die für das Verständnis der Arbeit notwendig sind. Ein wichtiger Teil in den Grundlagen ist die Vorstellung des inhaltszentrischen Ansatzes.

Mit dem Verständnis der Grundlagen wird im Kapitel 3 zunächst der Beitrag Herausforderungen und Potenziale, die eine Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge mit sich bringt, diskutiert. Mit dem Hintergrund der Herausforderungen und der Potenziale wird das Konzept der namenszentrischen Dienste aus dem inhaltszentrischen Ansatz entwickelt. Weiterhin werden die Begriffe Dienstmethode und Dienstbeschreibung im Kontext von namenszentrischen Diensten erläutert.

Nach dem Konzept folgen Architektur und Implementierung in Kapitel 4. In dem Kapitel wird zunächst die Systemarchitektur des Internet der Dinge vorgestellt. Die Systemarchitektur beschreibt die Hard- und Softwarekomponenten und die Schnittstellen zwischen den Komponenten. Auf die Systemarchitektur folgt die Softwarearchitektur, die die Softwarekomponenten und deren Schnittstellen im Detail betrachtet. Die Softwarekomponenten sind die Grundlage der Implementierung von *CCN-IoT*, die nach der Architektur vorgestellt wird. In diesem Abschnitt wird zuerst die Implementierung und das Nachrichtenformat von CCN-IoT erklärt. Im Anschluss erfolgt die Evaluation der Nachrichtenverarbeitung. Nach der Implementierung von CCN-IoT werden Vorschläge zur speichereffizienten Implementierung der Datenstruktur Forwarding Information Base gemacht, die im Anschluss evaluiert werden.

Kapitel Architektur und Implementierung hat die namenszentrischen Dienste beschrieben, jedoch nicht die Entwicklung von Diensten. Die Entwicklung von namenszentrischen Diensten wird in Kapitel 5 vorgestellt. Am Anfang der Entwicklung eines namenszentrischen Dienstes steht die Dienstbeschreibung im JSON-Format, die als Eingabe für die Werkzeugunterstützung dient. Die Werkzeugunterstützung generiert aus der Dienstbeschreibung Quellcode, der zur Interaktion der namenszentrischen Dienste mit CCNx oder CCN-IoT benötigt wird. In diesem Kapitel wird die Dienstbeschreibung anhand eines Schemas erläutert. Weiterhin werden Architektur und Implementierung der Werkzeugunterstützung vorgestellt und die Dienstbeschreibung evaluiert, wie im Beitrag Evaluation in Abschnitt 1.1 beschrieben.

In den bisherigen Kapiteln wurden der Aufbau und die Nutzung von Namen nicht behandelt. Kapitel 6 stellt ein Konzept zur Namensvergabe für Dienste

vor, dass auf einem Lebenszyklusmodell für namenszentrische Dienste basiert. Weiterhin werden in diesem Kapitel Konfigurationsmechanismen und zustandsbehaftete Verfahren zur Verkürzung von Namen vorgeschlagen.

Anwendungen von namenszentrischen Diensten und Namen werden in Kapitel 7 behandelt, wo die sogenannten Basisdienste vorgestellt werden. Die Basisdienste Neighborhood-Service und Backpath-Routing, zeigen mit der Zusammenarbeit von lose gekoppelten namenszentrischen Diensten, wie aus einfachen Diensten komplexere Dienste werden.

Ab Kapitel 3 bis Kapitel 6 werden jeweils am Anfang verwandte Arbeiten vorgestellt. Verwandte Arbeiten grenzen die Beiträge des jeweiligen Kapitels vom aktuellen Stand der Entwicklung ab oder zeigen die Konzepte auf, die von den jeweiligen anderen Entwicklungstrends übernommen wurden.

Die Arbeit schließt mit Kapitel 8, in der eine Zusammenfassung der Arbeit und ein Ausblick auf noch offene Fragestellungen beim inhalts-/namenszentrischen Internet beziehungsweise Internet der Dinge gegeben wird.

KAPITEL 2

GRUNDLAGEN

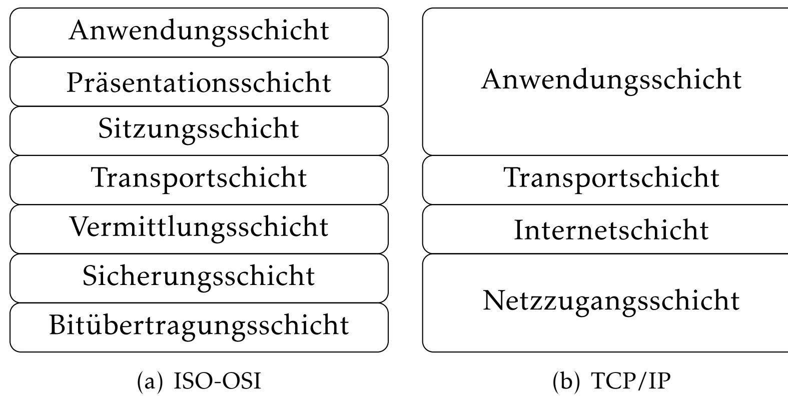
In diesem Kapitel werden Begriffe, wichtige Details zu Technologien und Paradigmen eingeführt, die für das Verständnis der Arbeit hilfreich sind und die Arbeit in einen Kontext einordnen. Die Begriffe im Titel der Arbeit geben bereits Teile des Kontextes vor. Der Titel setzt sich aus den Begriffen *namenszentrisch*, *Dienste*, *Internet* und *Dinge* zusammen, wobei die letzten beiden Begriffe den feststehenden Begriff *Internet der Dinge* bilden. Die Begriffe Internet, Internet der Dinge und Dienste, werden in diesem Kapitel in Abschnitt 2.1 beziehungsweise Abschnitt 2.2 näher erläutert. Der inhaltszentrische Ansatz bildet die Grundlage für die namenszentrischen Dienste. Dieser wird in Abschnitt 2.4 eingeführt.

2.1 INTERNET

Der geläufigste Begriff aus den vier oben aufgeführten Begriffen ist *Internet*. Das Internet ist ein weltweiter Verbund von Rechnernetzen, wobei die Rechnernetze von unterschiedlichen Organisationen betrieben werden. Ein wesentliches Element dieser Vernetzung sind die Protokolle, die die Regeln für die Kommunikation in den Rechnernetzen und der Rechnernetze untereinander sicherstellen. Bei der Darstellung der Protokolle haben sich die Referenzmodelle für Netzwerkprotokolle in Abbildung 2.1 durchgesetzt.

Referenzmodelle für Netzwerkprotokolle werden auch als *Schichtenmodelle* bezeichnet, da sie aus einzelnen Schichten bestehen. Die Schichten sind übereinander gestapelt und daher sind auch die Begriffe Stapel (engl. Stack), *Protokollstapel*, *Netzwerkstapel* oder *Netzwerkstack* gebräuchlich. Als Netzwerkstapel oder Netzwerkstack werden insbesondere konkrete Implementierungen bezeichnet, die den Referenzmodellen folgen.

Das ISO/OSI-Referenzmodell [61] in Abbildung 2.1(a) ist ein abstraktes Modell für digitale Netzwerkprotokolle. Es besteht aus sieben Schichten und jeder Schicht kommt dabei eine dedizierte Aufgabe zu. Die Aufgaben werden im Folgenden kurz vorgestellt.



(a) ISO-OSI (b) TCP/IP

Abbildung 2.1 – *Referenzmodelle für Netzwerkprotokolle*

Die unterste Schicht, Schicht 1, ist die Bitübertragungsschicht. Auf der Bitübertragungsschicht wird definiert, wie Bits über ein Medium übertragen werden. Darunter fallen Dinge wie Bitkodierung, Steckernormen und Funkfrequenzen bei drahtlosen Technologien.

Schicht 2 ist die Sicherungsschicht. Auf der Sicherungsschicht wird geregelt, wie die Daten bei einer Übertragung über das Übertragungsmedium in Rahmen (engl. Frames) aufgeteilt und abgesichert werden. Unter die Absicherung fällt die Definition von Fehlererkennungs- und Fehlerkorrekturmechanismen. Eine weitere Aufgabe der Sicherungsschicht ist der Medienzugriff. Der Medienzugriff definiert mit der MAC-Adresse (engl. Media Access Control) auch eine Adressierung von Netzwerkschnittstellen beziehungsweise Geräten. Die MAC-Adresse ist eine eindeutige Identifikation der Schnittstelle/des Gerätes in einem Netz.

Die Vermittlungsschicht auf Schicht 3 stellt Verbindungen zwischen Sender- und Empfängerknoten im Netz her. Wichtigste Aufgabe der Vermittlungsschicht ist die Wegewahl und Vermittlung von Paketen (engl. Routing) zwischen Sender und Empfänger eines Paketes. Damit die Wegewahl funktioniert, wird auf Schicht 3 die globale Adressierung umgesetzt.

Schicht 4 ist die Transportschicht. Die Transportschicht kontrolliert den Datenfluss zwischen der Sender- und Empfängeranwendung und stellt sicher, dass die Daten unverfälscht und in der Reihenfolge, in der sie gesendet wurden, beim Empfänger ankommen.

Die Sitzungsschicht auf Schicht 5 steuert den Verbindungsauf- und -abbau zwischen Sender und Empfänger. Weiterhin regelt sie den Nachrichtenaustausch zwischen den Teilnehmern auf der Transportschicht. Sind die Teilnehmer einer Verbindung beispielsweise nicht in der Lage, Nachrichten gleichzeitig zu senden und zu empfangen, wird durch die Sitzungsschicht die Reihenfolge festgelegt, in der die Teilnehmer Nachrichten senden und empfangen.

Die Darstellungsschicht, Schicht 6, bringt die Daten in eine einheitliche Form. Im Internet gibt es verschiedene Plattformen beziehungsweise Betriebssysteme mit unterschiedlichen Zeichenkodierungen. Die unterschiedlichen Zeichenko-

dierungen der Teilnehmer werden auf der Darstellungsschicht der jeweiligen Plattform in die plattformspezifische Zeichenkodierung konvertiert, um dann der nächst höheren Schicht in der richtigen Kodierung zur Verfügung zu stehen.

Die Anwendungsschicht, Schicht 7, umfasst die Anwendungsprotokolle. Anwendungsprotokolle werden von Anwendungen genutzt, die Dienste im Netz bereitstellen. Solche Anwendungen sind beispielsweise das Bereitstellen (Webserver) und Betrachten (Webbrowser) von Webseiten oder dem Senden, Empfangen und Verteilen von E-Mail.

Das TCP/IP-Referenzmodell in Abbildung 2.1(b) ist nach den zwei bedeutendsten Protokollen im Internet, dem Transmission Control Protocol (TCP) und dem Internet Protocol (IP), benannt. Konkrete Implementierungen von Netzwerkstapeln im Internet folgen dem TCP/IP-Referenzmodell. Es besteht aus vier Schichten: Anwendungs-, Transport-, Internet- und Netzzugangsschicht. Die Anwendungsschicht subsumiert die Sitzungs-, Darstellungs- und Anwendungsschicht (Schichten 5 bis 7) des ISO/OSI-Referenzmodells. Die Netzzugangsschicht des TCP/IP-Referenzmodells subsumiert die Bitübertragungs- und Sicherungsschicht (Schichten 1 und 2) des ISO/OSI-Referenzmodells. Das TCP/IP-Referenzmodell wurde in der Anfangszeit des Internets entwickelt. Damit wurde es noch vor dem ISO/OSI-Referenzmodell entwickelt und in der Tat hat das TCP/IP-Referenzmodell die Entwicklung des TCP/IP-Referenzmodells beeinflusst, wie Mohammed M. Alani in „Guide to OSI and TCP/IP Models“ [27] schreibt.

Die wichtigsten Protokolle auf jeder Schicht des TCP/IP-Referenzmodells sind in Tabelle 2.1 aufgeführt. Auf der Internetschicht ist IP das wichtigste Protokoll. Daher wird es auch als Rückgrat des *Internets* bezeichnet. Daneben finden sich auf der Internetschicht das auf IP aufbauende Internet Control Message Protocol (ICMP) [131, 58] sowie Routing-Protokolle.

Schicht	Protokolle
Anwendungsschicht	HTTP, SMTP, ICMP, IRC, DHCP, DNS, ...
Transportschicht	TCP, UDP, SCTP, RSVP, ...
Internetschicht	*IP* (IPv4 und ICMP, IPv6 mit ICMPv6), Routing-Protokolle (z. B. OSPF)
Netzzugangsschicht	Ethernet, 802.11 (WLAN), ...

Tabelle 2.1 – *Auswahl von Internetprotokollen nach Schichten*

IP stellt wenig Anforderungen an die darunterliegende Schicht und hat einen begrenzten Funktionsumfang. Die Aufgabe von IP, definiert in RFC 791 [132], ist der Transport von Paketen von Quellen zu Senken über miteinander verbundenen Netzwerke. Nach RFC 791 werden Quellen und Senken dabei über eine Adresse fester Länge, die *IP-Adresse*, identifiziert. Zu beachten ist, dass mit Quellen und Senken keine Netzknoten, sondern *Schnittstellen* gemeint sind. Netzknoten haben mindestens eine Schnittstelle und damit mindestens eine IP-Adresse.

IP gibt es in den Versionen 4 (IPv4) und 6 (IPv6), wobei IPv6 der designierte Nachfolger von IPv4 ist. Das Protokollverhalten ist bei beiden Versionen ähnlich, jedoch unterscheiden sich in den Header-Formaten und in Zusatzfunktionen, die in IPv6 neu hinzugekommen sind, wie automatische Adresskonfiguration, Fragmentierung und Sicherheitsfunktionen. Der bedeutendste Unterschied zwischen IPv4 und IPv6 ist die Größe des Adressraums, was sich in der Länge der IP-Adresse niederschlägt. IPv4 benutzt 32 Bit lange Adressen, IPv6 hingegen 128 Bit lange Adressen. Die öffentlichen Adressen (Adressen, zu denen im Internet weitergeleitet wird) im Adressraum von IPv4 sind mittlerweile alle vergeben. Das steht einem weiteren Wachstum des IP-basierten Internets entgegen. Mit der Ablösung von IPv4 durch IPv6 ist eine Erschöpfung des Adressraumes auch in ferner Zukunft nicht zu erwarten.

2.2 INTERNET DER DINGE

Der Begriff *Internet der Dinge* (engl. Internet of Things, kurz IoT) wurde im Jahre 1999 von Kevin Ashton, Mitarbeiter am Massachusetts Institute of Technology (MIT), geprägt [32]. Damals wurde der Begriff im Kontext der Radio-Frequency Identification-Technologie (RFID) gebraucht. Bei RFID handelt es sich um elektronische Barcodes. Informationen zu RFID finden sich in dem Buch „Radio-Frequency Identification Fundamentals and Applications“ von Klaus Finkenzeller [74]. In den folgenden Jahren etablierte sich der Begriff des Internet der Dinge so, wie er heute benutzt wird: Das Internet der Dinge sind vernetzte und in das Internet integrierte Geräte, die in Gegenständen des alltäglichen Lebens eingebaut sind. Die Verwendung des Begriffs Internet der Dinge findet man ebenso in den ITU Internet Reports von 2005 [157]. Der Begriff Internet der Dinge wird beispielsweise von Madakam et al. [101] oder Bian et al. [45] dahingehend kritisiert, da er inflationär und ohne genaue Definition verwendet wird.

Diese Arbeit orientiert sich an der Definition des Begriffs Internet der Dinge, wie er von Mattern und Flörkemeier in „Vom Internet der Computer zum Internet der Dinge“ [104] geprägt wird. Die Definition besteht aus zwei Teilen:

DEFINITION 2.1 – *Internet der Dinge:*

1. *Der Begriff* Internet *wird als Metapher in der Art interpretiert, dass Dinge das Internet genauso nutzen wie wir Menschen.*
2. *In einem Internet der Dinge verfügt jedes Ding über einen (TCP/IP-)*Netzwerkstack. *Wenn es nicht möglich ist, auf den Geräten selbst einen Netzwerkstack zu implementieren, implementiert alternativ ein Proxy oder ein Gateway einen Netzwerkstack für die Geräte.*

Punkt 2 in der obigen Definition 2.1 ähnelt der Argumentation von Vasseur und Dunkels [165], die sich auf ressourcenbeschränkte Geräte in *verlustbehafteten Netzen mit niedrigem Energieverbrauch* bezieht. Vasseur und Dunkels fordern keinen vollständigen TCP/IP-Netzwerkstack auf den Knoten, sondern dass die Knoten ein zu IP *semantisch kompatibles* Protokoll implementieren. Protokolle sind semantisch kompatibel, wenn sie das gleiche Verhalten haben. Nachrich-

tenformate zwischen semantisch kompatiblen Protokollen sind unterschiedlich. Die Übersetzung der Nachrichtenformate wird an einem Gateway oder an einem sogenannten *Border-Router* durchgeführt.

Das Konzept der Konvertierung von Nachrichtenformaten, kurz *Paketkonvertierung*, grenzen Vasseur und Dunkels von dem Konzept der *Protokollkonvertierung* ab. Bei einer Protokollkonvertierung werden Netze mit semantisch unterschiedlichen Protokollen über ein Protokollkonvertierungs-Gateway verbunden. Semantisch inkompatible Protokolle haben jeweils verschiedene Mechanismen für Routing, Dienstgüte, Fehlerbehandlung und Sicherheit. Als Beispiel führen Vasseur und Dunkels die Konvertierung von verschiedenen Routing-Protokollen an. Bei der Konvertierung von verschiedenen Routing-Protokollen müssen Metriken und sonstige Paradigmen des jeweiligen Routing-Protokolls in das andere übersetzt werden. Dies ist häufig nicht direkt möglich und daher kommt meist nur eine vollständige Trennung der Protokolle auf dem Gateway in Betracht. Die Trennung der Protokolle skaliert oft nicht, da eine Speicherung von komplexen Zuständen für jede einzelne Übertragung auf dem Gateway erforderlich ist. Bei Änderungen an Protokollen sind ebenso Änderungen am Gateway nötig und damit hat man einen doppelten Wartungsaufwand. Aufgrund der Problematiken bei der Skalierung und Wartung raten Vasseur und Dunkels von Protokollkonvertierungen ab.

In die Kategorie Protokollkonvertierung fällt auch ein Proxy, der stellvertretend für ressourcenbeschränkte Geräte im Netz einen TCP/IP-Netzwerkstack implementiert, wie ihn Mattern und Flörkemeier beschreiben (siehe Punkt 2 in Definition 2.1). Eine Protokollkonvertierung ist dabei nicht auf eine bestimmte Schicht begrenzt, wie bei Alcaraz et al. in „Wireless Sensor Networks and the Internet of Things: Do We Need a Complete Integration?“ [29] dargestellt.

Das oben genannte Beispiel, die Konvertierung von Routing-Protokollen, ist angreifbar, da nach heutigem Stand der Forschung Routing-Protokolle für verlustbehaftete Netze mit niedrigem Energieverbrauch unterschiedlich zu den im Internet benutzten Routing-Protokollen sind und eine Konvertierung von Routing-Protokollen generell nicht stattfindet. Eine Übersicht von Routing-Protokollen, die in drahtlosen Sensornetzen verwendet werden, findet sich in dem Internet Draft von Levis et al. [97].

Die Argumentation, dass Protokollkonvertierungen problematisch sind, ist nachvollziehbar. Das belegt auch die Veröffentlichung „Transparent Integration of Non-IP WSN into IP Based Networks“ von Teubler et al. [7], wo drahtlose Sensorknoten, auf denen das Datenverteilungsprotokoll *AutoCast* von Wegener et al. [169] implementiert ist, in ein TCP/IP-Netz integriert werden. Die Integration erfolgt hier mit dem protokollkonvertierenden Gateway *EZgate*. EZgate ist im Rahmen dieser Arbeit entstanden und basiert auf dem in Java implementierten TCP/IP-Protokollstapel EZnet [167] von Walther und Fischer. Als einzige Anwendung wird in „EZgate – A Flexible Gateway for the Internet of Things“ von Teubler et al. [8] die Erreichbarkeitsprüfung mit ICMP Echo [131] gezeigt. Jeder Knoten mit dem AutoCast Protokoll wird bei diesem Ansatz manuell in eine Konfigurationsdatei eingetragen, EZgate speichert den Zustand jeder Übertragung.

Im Folgenden werden drei wichtige Ansätze für das Internet der Dinge vorgestellt: *Drahtlose Sensornetze* in Abschnitt 2.2.1, Technologien für *Heimautomation* in Abschnitt 2.2.2 und *Message Queue Telemetry Transport* (MQTT) in Abschnitt 2.2.3. Die Drahtlosen Sensornetze werden detailliert behandelt, da in dieser Arbeit eine Methode zur Integration namenszentrischer Dienste auf drahtlosen Sensorknoten in ein inhaltszentrisches Internet implementiert wird. Drahtlose Sensorknoten sind in der insbesondere wissenschaftlichen Forschung relevant. Die Technologien für Heimautomation sind im Einzelhandel verfügbar und werden hier kurz vorgestellt, um Anwendungsszenarien für das Internet der Dinge und die Integration der Geräte in das Internet zu zeigen. MQTT wurde im Jahre 1999 entwickelt, um mit vernetzten ressourcenbeschränkten Geräten zu kommunizieren. Es wird hier kurz eingeführt, da es sich um einen Ansatz handelt der mit dem Aufkommen des Internet der Dinge an Popularität gewonnen hat, wie mehrere Publikationen zur praktischen Umsetzung von Anwendungen mit MQTT nahelegen. Hier sind beispielsweise „The MQTT Handbook - Everything You Need to Know about MQTT“ von Osborn [123], „MQTT im IoT: Einstieg in die M2M-Kommunikation“ von Edler et al. [68] oder „M2M by Eclipse“ von Obermaier und Cabé [120] zu nennen.

2.2.1 DRAHTLOSE SENSORNETZE

Drahtlose Sensornetze (engl. Wireless Sensor Networks, kurz WSN) sind laut Shelby und Bormann [149] seit Mitte der 90er Jahre des letzten Jahrhunderts Gegenstand der Forschung und werden mittlerweile als ein bedeutender Teil des Internet der Dinge wahrgenommen. Auch diese Arbeit ist teilweise im Rahmen der Forschung an drahtlosen Sensornetzen im Real-World G-Lab-Projekt [10, 11, 12] entstanden.

Drahtlose Sensornetze bestehen aus kleinen, meist batteriebetriebenen Computern, den sogenannten *Sensorknoten*, die mit Sensoren ausgestattet und drahtlos über Funk vernetzt sind. Sie verfügen über wenig Programm- und Arbeitsspeicher, eine geringe Rechenleistung, da die Sensorknoten meist nur mit 8 Bit Mikrocontrollern ausgestattet sind. Die Funkschnittstelle hat nur eine geringe Reichweite und unterstützt nur eine geringe Rahmengröße. IEEE 802.15.4 [82] ist der Funkstandard für drahtlose Sensornetze, die Rahmengröße beträgt inklusive Prüfsumme nur 127 Byte. Drahtlose Sensornetze fallen in die Kategorie der verlustbehafteten Netze mit niedrigem Energieverbrauch. Sensorknoten gehören zu den *eingebetteten Systemen*, da sie in die Umwelt und damit in einem größeren Kontext eingebettet sind (siehe z. B. „Eingebettete Systeme“ von Wehmeyer und Marwedel [170]).

Mit ihren Sensoren messen die Sensorknoten Daten aus ihrer Umwelt und leiten diese per Funk an eine zentrale Sammelstelle weiter. Da die Reichweite der Funkschnittstelle begrenzt ist, dienen Knoten auf dem Weg als sogenannte Relays, das heißt sie empfangen die Nachricht und senden diese weiter. Nachrichten werden so über die Funkreichweite der Knoten hinaus weitergeleitet. Da die Sensorknoten ihre eigene Infrastruktur aufbauen, nennt man diese Netze auch *Ad-hoc-Netze* (ad hoc, lat. „für den Moment/Augenblick gemacht“).

Abbildung 2.2 zeigt ein Beispiel für ein drahtloses Ad-hoc-Netz. Die Funkreichweite ist als Kreis konzentrisch um die Antenne der Sensorknoten dargestellt. In dem Beispiel ist Knoten *B* in Funkreichweite von Knoten *A*, daher ist *B* in der Lage, Nachrichten von *A* zu empfangen. Umgekehrt ist auch *A* in Funkreichweite von *B* und somit ist auch *A* in der Lage, Nachrichten von *B* zu empfangen. Gleiches gilt für *B* und *C*. *A* und *C* sind nicht in Funkreichweite, daher ist eine direkte Kommunikation zwischen *A* und *C* nicht möglich. Eine Kommunikation zwischen *A* und *C* erfolgt über Knoten *B*, der als Relay fungiert.

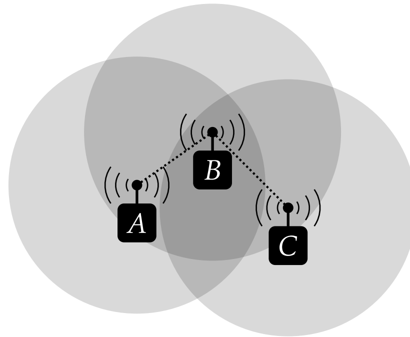


Abbildung 2.2 – *Beispiel für ein drahtloses Ad-hoc-Netz*

Sind drahtlose Sensorknoten im Ad-hoc-Netz in der Lage, miteinander zu kommunizieren, existiert ein Kommunikationskanal (engl. *Link*) zwischen den Knoten. In Abbildung 2.2 sind die Links als gepunktete Linie dargestellt. Die Gesamtheit aus Knoten und Links wird *Topologie* genannt. Eine wissenschaftlich relevante Fragestellung ist, wie das Routing der Nachrichten in drahtlosen Netzen funktioniert; siehe hierzu beispielsweise die Arbeiten von Akyildiz et al. [26], Yick et al. [175] und Taneja und Kush [159].

Der Vollständigkeit halber sei erwähnt, dass die Funkreichweite beziehungsweise die Funkausbreitung in der Realität in der Regel nicht kreisrund, sondern eher eine unregelmäßige Form hat. Um Links in einem drahtlosen Ad-hoc-Netz wie in Abbildung 2.2 zu visualisieren, ist die Vereinfachung tolerierbar.

Drahtlose Sensornetze sind immer noch Gegenstand der Forschung und werden im wissenschaftlichen Bereich aktiv eingesetzt. Potdar et al. [134] stellen Sensorknoten-Plattformen für die Forschung und Gluhak et al. [77] Forschungsumgebungen (Testbeds) für drahtlose Sensornetze vor. Mainwaring et al. zeigen in „Wireless Sensor Networks for Habitat Monitoring“ [102] eine Sensorknotenanwendung zur Habitatbeobachtung von Seevögeln.

Aktuelle Entwicklungen in dem Bereich legen aber nahe, dass es in absehbarer Zukunft weitere Anwendungen in der Industrie und in der Heimautomation geben wird. Als Beispiel ist hier die THREAD-Technologie [163] des THREAD-Konsortiums zu nennen. Das THREAD-Konsortium realisiert zurzeit Lösungen zur Heimautomation von drahtlosen Sensornetzen.

Am Anfang der Forschung von drahtlosen Sensornetzen ging man davon aus, dass die Daten, die sie produzieren, manuell abgeschöpft werden; eine Integration in größere Netze war nicht angedacht [149]. Mit dem Aufkommen der Idee

des Internet der Dinge hat sich die Situation geändert und das Forschungsfeld Internet der Dinge hat die drahtlosen Sensornetze in sich aufgenommen.

Die konsequente Entwicklung im Zuge der Integration von Sensornetzen in das Internet der Dinge ist, dass man die Sensorknoten mit einem zu IPv6 kompatiblen Netzwerkstack versieht. Man geht davon aus, dass IPv4 mittelfristig durch IPv6 abgelöst wird, daher hat man sich bei der Integration von drahtlosen Sensornetzen auf IPv6 fokussiert. Vorreiter auf dem Gebiet der Integration von drahtlosen Sensornetzen in das Internet mit IPv6 sind Vasseur und Dunkels [165] sowie Shelby und Bormann [149]. Die Integration von drahtlosen Sensornetzen erfolgt am „Rand“ (engl. Border oder Edge) des Internets. Das heißt, dass kein Internetverkehr durch das drahtlose Sensornetz geleitet wird. Das Sensornetz ist über einen sogenannten Border- oder Edge Router mit dem Internet verbunden, vergleichbar mit einem lokalen Netzwerk (engl. Local Area Network, kurz LAN), das über ein Gateway an das Internet angebunden ist.

Aufgrund der Ressourcenbeschränkungen der Sensorknoten ist die Implementierung beziehungsweise die Portierung eines TCP/IP-Protokollstapels, wie er auf PCs zu finden ist, nicht möglich. Deshalb wurden semantisch kompatible Protokolle beziehungsweise Adaptions-Layer entwickelt [165], die eine Integration in das Internet erlauben. Auf der Internetschicht ist 6LoWPAN [110, 149] als Adaptions-Layer für IPv6 mittlerweile ein Standard. Abbildung 2.3 zeigt den Netzwerkstack für drahtlose Sensorknoten mit den Zuordnungen zu den Schichten des ISO/OSI-Referenzmodells.

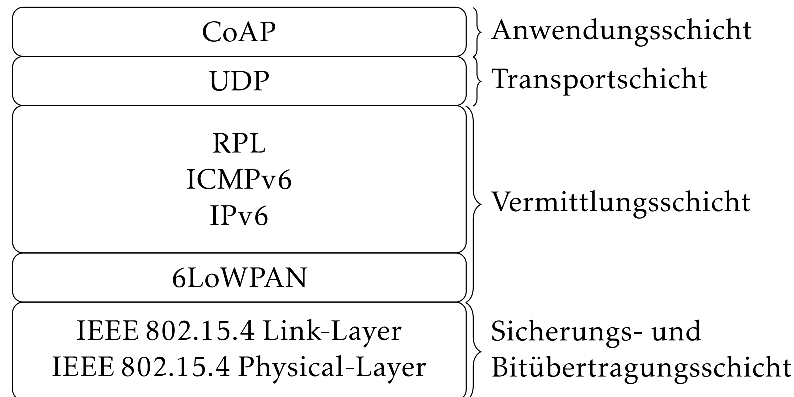


ABBILDUNG 2.3 – *Netzwerkstack für drahtlose Sensorknoten*

In jeder Schicht in Abbildung 2.3 sind die Protokolle aufgeführt, die für die jeweilige Schicht von Bedeutung sind. Die Aufgaben der Protokolle 6LoWPAN, RPL und CoAP, die im Kontext der drahtlosen Sensornetze entwickelt wurden, werden im Folgenden kurz eingeführt. Weiterhin wird der Stand beziehungsweise die Entwicklungsreife des jeweiligen Protokolls betrachtet.

6LOWPAN

6LoWPAN ist eine Kurzform für *IPv6 over Low power Wireless Personal Area Networks* [92, 28]. Die Aufgabe von 6LoWPAN ist es, IPv6 Pakete zum Transport

über IEEE 802.15.4 anzupassen. 6LoWPAN ist eine sogenannte *Adaptions-Schicht* und liegt, wie in Abbildung 2.3 zu sehen, zwischen der IEEE 802.15.4 Sicherungsschicht und IPv6 (Netzwerk- oder Internetschicht). 6LoWPAN wird der Netzwerk- oder Internetschicht zugeordnet. Die von 6LoWPAN vorgenommene Anpassung ist nötig, da IPv6 von der Sicherungsschicht eine unterstützte Rahmengröße von mindestens 1280 Byte erwartet. IEEE 802.15.4 hat eine maximale Rahmengröße von 127 Byte und daher nimmt 6LoWPAN eine Fragmentierung von Rahmen vor, die nicht in einem IEEE 802.15.4 Rahmen passen. Weiterhin implementiert 6LoWPAN eine IPv6 Header-Komprimierung [80] um IPv6 Pakete weiter zu verkleinern.

Die Anfänge von 6LoWPAN reichen auf das Jahr 2007 mit der Veröffentlichung von RFC 4919 [92] zurück. Ebenfalls um diese Zeit sind erste Implementierungen von 6LoWPAN entstanden, wie in „Comparisons of 6LoWPAN Implementations on Wireless Sensor Networks" von Mazzer und Tourancheau [105] dargestellt. Eine Implementierung aus dieser Zeit, die bis heute aktiv gepflegt wird, ist die des Contiki [66, 67] Betriebssystems für drahtlose Sensorknoten mit dem Namen µIP. Im Linux Kernel gibt es seit 2014 eine Implementierung von 6LoWPAN. Deren letzte Änderungen, Stand Mai 2018, datieren auf November 2017. Eine weitere Implementierung von 6LoWPAN findet sich im RIOT Betriebssystem, ebenfalls für ressourcenbeschränkte Geräte [34]. In der Vergangenheit gab es noch weitere Entwicklungen, beispielsweise von der Firma Sensinode, die von der Firma ARM aufgekauft wurde [30]. ARM bietet mit dem nun *mbed* genannten Framework eine eigene Internet of Things Plattform an, die 6LoWPAN unterstützt [31].

Darüber hinaus gibt es noch weitere Implementierungen von 6LoWPAN aus dem akademischen Bereich [105, 95, 144]. Die Autoren kritisieren an den existierenden Implementierungen, wie beispielsweise an µIP, eine mangelnde Modularität, die Erweiterungen kaum möglich machen.

Zusammenfassend scheint sich 6LoWPAN in einem knotenzentrischen Internet der Dinge durchzusetzen. Dass das THREAD-Konsortium [163] und ARM mit dem mbed-Framework auf IPv6 und 6LoWPAN setzen, untermauert diese Aussage.

RPL

RPL (als „Ripple" ausgesprochen, engl. für Wellen erzeugend, kräuselnd) ist der Vorschlag für ein Routing-Protokoll in drahtlosen Sensornetzen durch die ROLL (Routing Over Low power and Lossy networks) Arbeitsgruppe der IETF (Internet Engineering Task Force) [173].

RPL ist ein Distanz-Vektor Routing Protokoll [108], welches einen zielorientierten, gerichteten, azyklischen Graphen (engl. Destination Oriented Directed Acyclic Graph, kurz DODAG) als Topologie im drahtlosen Netz aufbaut. Zielorientiert bedeutet dabei, dass es einen speziellen Knoten gibt, der nur eingehende Kanten, aber keine ausgehenden Kanten hat. Dieser Knoten wird als *Wurzel* (engl. Root) bezeichnet. Die Generierung dieses Graphen startet bei der Wurzel und die Wurzel ist in der Regel ein Edge Router.

Abbildung 2.4 zeigt ein Beispiel für eine DODAG Topologie in einem drahtlosen Sensornetz. Die Wurzel ist farblich hervorgehoben, die Links sind als gepunktete Linien dargestellt. Links, die von RPL in den DODAG mit einbezogen werden, sind als durchgezogene Pfeile dargestellt. Nachrichten werden nur über die DODAG-Links geleitet.

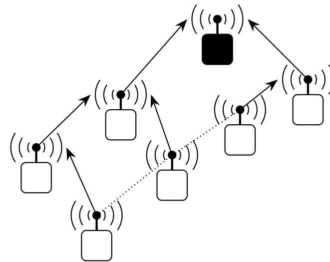


Abbildung 2.4 – *DODAG Topologie in einem drahtlosen Sensornetz*

Clausen et al. schreiben in „A Critical Evaluation of the IPv6 Routing Protocol for Low Power and Lossy Networks (RPL)“ [57], dass es das inoffizielle Ziel von RPL ist, ein Standard Routing-Protokoll für drahtlose Sensornetze zu entwickeln, um eine einheitliche Protokollarchitektur innerhalb aller drahtlosen Sensornetze zu haben. Clausen et al. identifizieren darüber hinaus diverse Schwächen von RPL, die es für den Einsatz in drahtlosen Sensornetzen ungeeignet erscheinen lassen. Zu den Schwächen zählen ein hoher Ressourcenverbrauch, die Annahme von unrealistischen Anwendungsszenarien sowie die Voraussetzung bidirektionaler Links. Details zu den Schwächen werden von Clausen et al. ausführlich erläutert.

Weitere Probleme von RPL sind die hohe Komplexität und die Abhängigkeit von Protokollen zur Auswahl bidirektionaler Links wie Neighbor Unreachability Detection [116] oder Bidirectional Forwarding Detection [90]. Die Unterspezifikation des Standards, die Komplexität und der Ressourcenverbrauch von RPL führen laut Clausen et al. dazu, dass RPL Implementierungen nur unvollständig umgesetzt werden. Aufgrund seiner Schwächen ist RPL aktuell Gegenstand der Forschung, mit der Zielsetzung, es zu verbessern. Kermajani und Gomez [91] sowie Parasuram et al. [125] untersuchen in ihren Arbeiten RPL und schlagen Verbesserungen vor.

CoAP

CoAP, kurz für Constrained Application Protocol [151], ist das Anwendungsprotokoll, das als Internetstandard für drahtlose Sensornetze vorgesehen ist. Es ist semantisch kompatibel zu HTTP (Hypertext Transfer Protocol) [73, 40], jedoch liegt der Fokus von CoAP auf einer Machine-to-Machine (dt. Maschine-zu-Maschine)-Kommunikation (M2M). CoAP basiert auf dem sogenannten *REST* Paradigma. REST ist eine Abkürzung für *Representational State Transfer*. Hierbei handelt es sich um einen Architekturstil für Dienste, bei dem Ressourcen auf einem Server über ihre URI (Uniform Resource Identifier) identifiziert werden.

Ein grundlegendes Konzept von REST sind sogenannte *Ressourcen*. Ressourcen werden über einen eindeutigen Identifizierer, beispielsweise URIs [42], ange-

sprochen. REST erlaubt es vier Operationen auf Ressourcen auszuführen: Create (dt. Erstellen), Retrieve (dt. Erhalten/Bekommen), Update (dt. Erneuern) und Delete (dt. Löschen).

Wesentliche Unterscheidungsmerkmale zwischen HTTP und CoAP sind, dass CoAP binär kodiert und HTTP ASCII-kodiert [44] ist und dass CoAP zusätzliche optionale Maßnahmen zur verlässlichen Datenübertragung bietet, da CoAP in der Regel über UDP (vgl. Abbildung 2.3) und nicht über TCP, welches sonst die verlässliche Übertragung realisiert, ausgeführt wird. Zurawski vergleicht im „Industrial Communication Technology Handbook“ [182] HTTP und CoAP und zeigt neben den oben aufgeführten weitere Gemeinsamkeiten und Unterschiede auf.

Da CoAP auf M2M Kommunikation fokussiert ist, hat es einen Ressource-Discovery Mechanismus, was sich mit „Ressourcenauffindung“ übersetzen lässt. Ressourcen, im Sinne von CoAP sind Daten wie Sensorwerte. Die Ressourcen, die ein Knoten anbietet, werden in einer Ressource mit dem Namen `core` unter dem URI-Pfadpräfix `/.well-known` [151, 118] angeboten. Die Dokumente unter dem Pfadpräfix `/.well-known` folgen dem CoRE (Constrained RESTful Environments) Link Format [150]. Die Links sind einem Ressourcentyp zugeordnet, der die Ressource beschreibt. Eine Anfrage nach `/.well-known/core?rt=temperature-c` liefert alle beispielsweise Links zu Ressourcen, die den Ressourcentyp `temperature-c` haben [150]. Damit eine gezielte Suche nach Ressourcen funktioniert, sind Ressourcentypen und deren Bedeutung im Vorwege bekannt. Aus diesem Grunde hat die IANA (Internet Assigned Numbers Authority) einige Ressourcentypen definiert [78]. Alternativ zu Ressourcentypen werden in CoAP Interface-Typen [150] verwendet, die eine allgemeine Beschreibung von Ressourcen ermöglichen. Die allgemeine Beschreibung erfolgt mit der Web Application Description Language (WADL) [79].

Es gibt mehrere untereinander kompatible Implementierungen von CoAP, auch von Industrieunternehmen. Lerche et al. [96] zählten im Jahre 2012 bereits 20 CoAP Implementierungen.

Zusammenfassend zeigt sich der Trend, dass drahtlose Sensornetze knotenzentrisch entworfen werden. Auf der Vermittlungsschicht kommt das zu IPv6 semantisch kompatible 6LoWPAN zum Einsatz. Als Ad-hoc-Routing-Protokoll wird RPL vorgeschlagen. Dienste in drahtlosen Sensornetzen werden mit CoAP realisiert.

2.2.2 TECHNOLOGIEN FÜR HEIMAUTOMATION

Mit Heimautomation werden die Funktionen eines Hauses, in der Regel Heizung, Licht, Jalousien, Alarmanlagen, usw. automatisch gesteuert. Die Steuerung der Funktionen ist auch interaktiv über das Internet möglich, daher werden Anwendungen der Heimautomation dem Internet der Dinge zugeordnet. Anwendungen zur Heimautomation haben sich parallel zu den drahtlosen Sensornetzen entwickelt. Für Heimautomation gibt es viele Produkte auf dem Markt.

Geräte für Heimautomation, die über mehr Ressourcen wie Rechenleistung und Speicher verfügen, sind über Ethernet und WLAN mit dem Internet verbunden.

Für ressourcenbeschränkte Geräte in der Heimautomation haben sich Standards wie das auf IEEE 802.15.4 basierende ZigBee [181] und Z-Wave [177] etabliert, die mittlerweile weit verbreitet sind. ZigBee als auch Z-Wave sind nur zwei Beispiele für Industriestandards, die von Firmenallianzen aus der Branche der Heimautomation getragen werden.

Die Integration von ZigBee (oder vergleichbaren Technologien) basierten Lösungen in das Internet wird mit Gateways realisiert. Informationen zur Integration von ZigBee Geräten in das Internet finden sich in „IoT Gateway: Bridging Wireless Sensor Networks into Internet of Things" von Zhu et al. [179]. Mittlerweile gibt es bei der ZigBee Allianz ein Umdenken, denn mit *ZigBee IP* [180] ist ein Netzwerkstack, der ebenfalls IPv6 und 6LoWPAN nutzt, standardisiert worden.

Lösungen zur Heimautomation wurden hier kurz vorgestellt, da sie ein Teil des Internet der Dinge sind. Bei der Heimautomation zeigt sich die homogene Architektur des Internet der Dinge, da Geräte mit unterschiedlichsten Ressourcen ein Teil des Internets werden. Alle Lösungen aus dem Bereich der Heimautomation setzen auf einen knotenzentrischen Ansatz. Es zeigt sich, dass es aktuell noch keinen richtig etablierten Netzwerkstapel von Protokollen für das Internet der Dinge gibt. Aus diesem Grunde ist es jetzt noch sinnvoll, den inhaltszentrischen Ansatz als Alternative zu den knotenzentrischen Ansätzen im Internet der Dinge zu betrachten, da der inhaltszentrische Ansatz noch nicht mit einem etablierten Netzwerkstapel konkurriert.

2.2.3 MESSAGE QUEUE TELEMETRY TRANSPORT

Message Queue Telemetry Transport (MQTT) [154] ist ein Publish/Subscribe-Protokoll (dt. veröffentlichen und registrieren) für das Internet der Dinge. Die Geräte/Dinge registrieren sich dabei an einem sogenannten MQTT-Broker, zu dem sie ihre Daten schicken/veröffentlichen beziehungsweise von dem sie Daten erhalten. Nutzer kommunizieren nicht mit den Geräten, sondern nur mit dem Broker. Für ressourcenbeschränkte Geräte, beispielsweise Sensorknoten, gibt es MQTT-SN [156], wobei SN hier für Sensor Network (dt. Sensornetz) steht.

Bei MQTT-SN kommunizieren die Sensorknoten über Gateways mit den Brokern. Gateways sind entweder transparent oder aggregierend. Bei transparenten Gateways baut jeder Sensorknoten über das Gateway eine Ende-zu-Ende-Verbindung zum Broker auf. Auf dem transparenten Gateway erfolgt eine Paketkonvertierung von MQTT-SN zu MQTT und zurück.

Bei einem aggregierenden Gateway bauen die Sensorknoten nur zu dem Gateway jeweils eine Ende-zu-Ende-Verbindung auf. Das aggregierende Gateway baut wiederum zum Broker eine Ende-zu-Ende-Verbindung auf. Daten von mehreren Knoten werden am Gateway aggregiert zum Broker geschickt. Das aggregierende Gateway entscheidet, welche Daten es aggregiert und weiterschickt.

Der gravierendste Unterschied zwischen den Ansätzen von MQTT und 6LoWPAN/CoAP ist, dass bei MQTT kein transparenter Zugriff auf die Sensorknoten stattfindet. Der Nutzer verbindet sich bei MQTT über einen MQTT-Client mit dem Broker. Dieser Client operiert auf der Anwendungsschicht über TCP/IP. Der Ansatz von MQTT folgt Punkt 2 der Definition 2.1 auf Seite 14 des Internet

der Dinge, da der Broker ein Proxy ist, der für die Sensorknoten einen TCP/IP-Netzwerkstapel implementiert. Die Daten, die ein Nutzer mit dem MQTT-Client abruft, heißen bei MQTT *Topics*. Topics werden, ähnlich wie beim inhaltszentrischen Content-Centric Networking, mit hierarchischen Namen adressiert.

MQTT wird hier erwähnt, da es sich, um ein populäres Protokoll für das Internet der Dinge handelt. Die in Kapitel 4 vorgestellte Architektur sieht wie MQTT Gateways für die Integration der drahtlosen Netze in das Internet vor.

2.3 DIENST UND ANWENDUNG

Neben den technischen Grundlagen werden in diesem Kapitel auch allgemeine Begriffe eingeführt. Einer dieser Begriffe ist *Dienst*, der in diesem Abschnitt eingeführt wird.

Im ISO/OSI-Referenzmodell [61] wird ebenfalls der Begriff Dienst verwendet. So bietet im ISO/OSI-Referenzmodell die jeweils untere Schicht der darüberlegenden einen Dienst an. Ein Dienst auf einer Schicht im ISO/OSI-Referenzmodell implementiert ein Protokoll. Abbildung 2.5 zeigt Dienst und Protokoll im ISO/OSI-Referenzmodell und folgt der Darstellung aus „Computer Networks" von Andrew S. Tanenbaum und David J. Wetherall [160]. Mit Hilfe eines Protokolls und der darunterliegenden Schicht wird ein Dienst erbracht. So implementiert die Vermittlungsschicht das Internet Protokoll (vgl. Abschnitt 2.1), das als Dienst eine unzuverlässige Übertragung zwischen zwei Endpunkten anbietet.

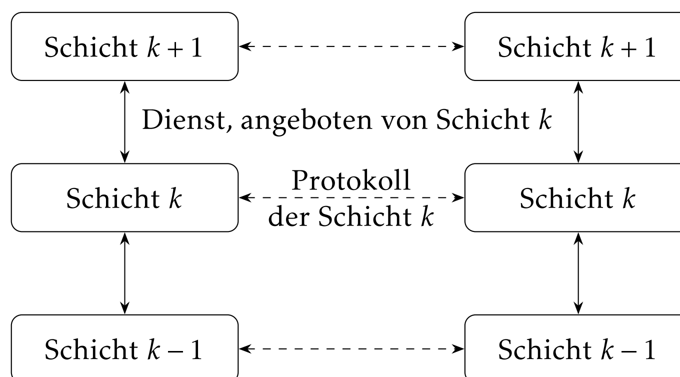


ABBILDUNG 2.5 – *Dienst und Protokoll im ISO/OSI-Referenzmodell*

Im Referenzmodell der serviceorientierten Architekturen (SOA) [100], einem Architekturstil für verteilte Anwendungen, wird der Begriff Dienst folgendermaßen definiert: *The noun "service" is defined in dictionaries as "The performance of work (a function) by one for another." However, service, as the term is generally understood, also combines the following related ideas:* (i) *The capability to perform work for another.* (ii) *The specification of the work offered for another.* (iii) *The offer to perform work for another.*

Nach der obigen Definition erledigt ein Dienst eine *Aufgabe* für jemand anderen. Ein Dienst besitzt die *Fähigkeit* („capability"), Arbeit für jemand anderen durchzuführen. Es gibt eine *Spezifikation* („specification") der Arbeit, die ein Dienst

anderen anbietet und es gibt ein *Angebot* („offer“), dass der Dienst Arbeit für andere verrichtet.

Die obige Definition ist sehr allgemein, ihr fehlt der technische Aspekt. Da in dieser Arbeit auch Aspekte der Implementierung betrachtet werden, werden weitere Definitionen herangezogen, die den Begriff Dienst technisch definieren. Bianco et al. [46] definieren einen Dienst wie folgt: *(i)* Ein Dienst steht für sich selbst. Der Dienst ist modular und wird unabhängig eingesetzt. *(ii)* Ein Dienst ist eine verteilte Komponente. Der Dienst ist über das Netzwerk verfügbar und neben der absoluten Netzwerkadresse über einen Namen erreichbar. *(iii)* Ein Dienst hat eine öffentliche Schnittstelle. Benutzer des Dienstes brauchen nur die Beschreibung der Schnittstelle, die Implementierungsdetails sind für die Nutzer transparent. *(iv)* Ein Dienst ist interoperabel. Nutzer eines Dienstes verwenden einen Dienst unabhängig von verwendeter Programmiersprache oder Plattform. *(v)* Ein Dienst ist auffindbar. Ein Dienst wird in einem speziellen Verzeichnisdienst registriert und ist dann für Nutzer über diesen Verzeichnisdienst auffindbar. *(vi)* Ein Dienst wird dynamisch zur Laufzeit gebunden. Nutzer brauchen keine Implementierung des Dienstes zur Entwicklungszeit.

Die oben genannten Eigenschaften stellen nach Bianco et al. einen idealen Dienst dar. Laut der Autoren werden in der Praxis einige dieser Eigenschaften vernachlässigt oder fehlen ganz, wie beispielsweise die letzten beiden Punkte aus der obigen Aufzählung. Die Begriffe *Netz*, *Schnittstelle* und *Programmiersprache* in der Definition von Bianco et al. verdeutlichen, dass es sich um eine Software handelt.

Im Service-Oriented Modelling Framework (SOMF) [38], einer Modellierungssprache für Geschäftsprozesse und Softwaresysteme, wird Dienst sinngemäß als Softwarekomponente definiert, die als ganzheitliche Einheit die Anforderungen des Geschäfts kapselt. Auch hier wird deutlich, dass es sich bei einem Dienst um *Software* handelt.

Unter Berücksichtigung der obigen Definitionen wird für diese Arbeit der Begriff *Dienst* folgendermaßen definiert:

DEFINITION 2.2 – Dienst*: Ein Dienst ist eine Software, die über das Netz verfügbar ist. Diese Software stellt eine definierte Funktionalität bereit und ist interoperabel. Ein Dienst stellt seine Funktionalität über eine offengelegte, definierte Schnittstelle zur Verfügung und liefert eine Beschreibung der Funktionalität, die er bereitstellt (Dienstbeschreibung).*

2.3.1 DIENSTANBIETER, DIENSTNUTZER, ANFRAGE UND ANTWORT

Bianco et al. führen neben dem Begriff *Dienst* die Begriffe *Dienstanbieter* (engl. Service Provider) und *Dienstnutzer* (engl. Service User oder Service Consumer) ein. Für Dienstanbieter und Dienstnutzer gelten nach Bianco et al. folgende Nebenbedingungen: *(i)* Dienstnutzer senden *Anfragen* zu Dienstanbietern. *(ii)* Wenn Dienstanbieter andere Dienste nutzen, sind sie gleichzeitig Dienstnutzer.

Obige Aufzählung enthält einen weiteren wichtigen Begriff: *Anfrage* (engl. *Request*). Das Pendant zur Anfrage ist die *Antwort* (engl. *Reply*). Anfrage und Antwort sind Nachrichten. Eine Antwort wird vom Dienstanbieter zum Dienstnutzer auf eine Anfrage des Dienstnutzers geschickt. Anfragen und Antworten werden in Form von *Nachrichten* ausgetauscht. Anfrage, Antwort, Dienstanbieter und Dienstnutzer sind zentrale Begriffe und daher werden sie hier, wie sie in dieser Arbeit verwendet werden, definiert:

DEFINITION 2.3 – Anfrage*: Eine Anfrage ist eine Nachricht. Sie wird vom Dienstnutzer zu einem Dienstanbieter geschickt.*

DEFINITION 2.4 – Antwort*: Eine Antwort ist eine Nachricht. Sie wird vom Dienstanbieter nach einer vorherigen, vom Dienstanbieter empfangenen Anfrage, zu einem Dienstnutzer geschickt.*

DEFINITION 2.5 – Dienstanbieter*: Ein Dienstanbieter ist eine Entität, die einen Dienst zur Verfügung stellt. Er empfängt Anfragen von Dienstnutzern. Nur auf eine Anfrage schickt der Dienstanbieter eine Antwort. Das Senden einer Antwort ist, in Abhängigkeit des erbrachten Dienstes, optional.*

DEFINITION 2.6 – Dienstnutzer*: Ein Dienstnutzer ist eine Entität, die einen Dienst nutzt, indem sie eine Anfrage zu dem Dienstanbieter schickt.*

2.3.2 ANWENDUNG UND ANWENDER

Die Begriffe *Anwendung* und *Anwender* werden hier kurz vorgestellt, um sie von den Begriffen Dienst und Dienstnutzer abzugrenzen. In Lexika [161, 69] wird Anwendung sinngemäß als Computerprogramm mit einem klar abgegrenzten Funktionsbereich definiert, welches den Anwender bei der Ausführung bestimmter Aufgaben unterstützt. Diese Definition ähnelt der obigen Definition des Begriffs Dienst und auch das ISO/OSI-Referenzmodell stützt diese Implikation, wie im nächsten Abschnitt erläutert wird.

Das ISO/OSI-Referenzmodell benutzt den Begriff Anwendung in der Anwendungsschicht (vgl. Abbildung 2.1(a)). Im ISO/OSI-Referenzmodell ist der Begriff Dienst wie die allgemeine Definition des Begriffs (siehe oben) zu verstehen. Die Anwendungsschicht stellt der Anwendung einen Dienst zur Verfügung und die Anwendung wiederum stellt dem Anwender einen Dienst zur Verfügung, welche ihn bei der Ausführung bestimmter Aufgaben unterstützt, wie in Abbildung 2.6 dargestellt.

Diese Arbeit sieht vor, dass Softwareentwickler in den von ihnen entwickelten Diensten auch andere Dienste nutzen. Durch das Nutzen anderer Dienste werden Softwareentwickler in ihren Aufgaben unterstützt. Somit sind auch sie Anwender (von anderen Diensten) und die anderen Dienste realisieren eine Anwendung. Ein Anwender ist damit im gewissen Sinne auch eine Entität eines Dienstnutzers nach Definition 2.6. In dieser Arbeit werden Anwender allerdings nicht als Dienstnutzer bezeichnet, sondern die Software, mit der sie einen Dienst nutzen.

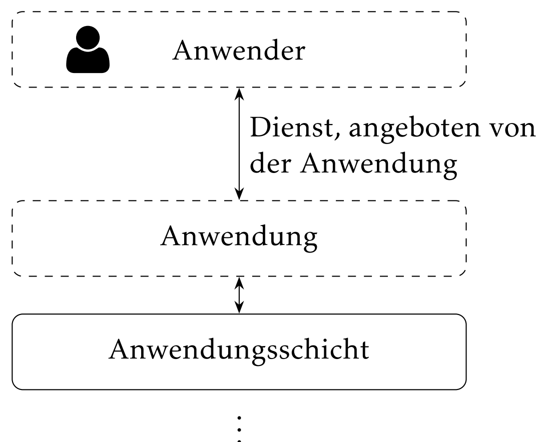


ABBILDUNG 2.6 – *Die Anwendung stellt dem Anwender einen Dienst zur Verfügung*

2.4 INHALTSZENTRISCHER ANSATZ

Der hier vorgestellte inhaltszentrische Ansatz CCN basiert auf dem Protokoll mit dem Namen CCNx [86]. CCNx liegt mittlerweile in der Version 1.0 [112] vor. Die im Rahmen der Arbeit entstandene Implementierung für drahtlose Sensorknoten und die darauf basierenden Untersuchungen beziehen sich auf die Version 0.7.0 von CCNx. CCNx 1.0 führt Erweiterungen des CCNx-Protokolls sowie eine Semantik für Namen ein. Die Betrachtungen in dieser Arbeit greifen diese Neuerungen bewusst nicht auf, da viele der Neuerungen in CCNx 1.0 inkompatibel zu den ressourcenbeschränkten Geräten sind, die in dieser Arbeit betrachtet werden. Das grundlegende Funktionsprinzip von CCN stellt die Basis für die Beiträge dieser Arbeit dar. Im Folgenden werden die wesentlichen Aspekte von CCN eingeführt, die zum Verständnis des inhaltszentrischen Ansatzes nötig sind. Wesentliche Aspekte von CCN sind die *Namen*, das *Knotenmodell* mit seinen Datenstukturen und das *Funktionsprinzip*.

2.4.1 CCN-NAMEN

In der Literatur zu CCN werden Namen an Beispielen erklärt. Diese Arbeit bedient sich bei der Definition von Namen und den dazugehörenden Begriffen mit der Backus-Naur-Form (BNF) eines formalen Ansatzes. Die BNF ist eine Metasprache zur Syntaxdarstellung und wurde von Peter Naur und John W. Backus in „Revised Report on the Algorithmic Language Algol 60“ [117] veröffentlicht. Die BNF benutzt Ableitungsregeln, um Zeichenketten zu beschreiben. Jede Zeile in der BNF ist eine Produktionsregel. Ausdrücke der BNF in spitzen Klammern $\langle \dots \rangle$ sind sogenannte Nichtterminalsymbole. Ein besonderes Nichtterminalsymbol ist das Endesymbol ε, das das Ende einer Zeichenkette markiert und bei der generierten Zeichenkette nicht ausgegeben wird. Aus den Nichtterminalsymbolen auf der linken Seite des Produktionsoperators „$\models$“ werden Zeichenketten aus Terminalsymbolen und Nichtterminalsymbolen auf der rechten Seite abgeleitet. Alternativen bei der Ableitung werden durch den Alternativenoperator „ | “ getrennt. Die Nichtterminalsymbole werden mit den Produktionsregeln solange

abgeleitet, bis die Zeichenkette keine Nichtterminalsymbole mehr enthält. CCN-Namen bestehen aus Komponenten, daher wird zunächst *Komponente* mit Hilfe der BNF in Definition 2.7 definiert:

DEFINITION 2.7 – *Komponente (und Prozentkodierung):*

$$\begin{aligned}
\langle\text{Komponente}\rangle &\models \; \texttt{/}\,\langle\text{s}\rangle \\
\langle\text{s}\rangle &\models \; \langle\text{c}\rangle\,\langle\text{s}_\varepsilon\rangle \mid \langle\text{Prozentkodierung}\rangle\,\langle\text{s}_\varepsilon\rangle \\
\langle\text{c}\rangle &\models \; \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \mid \texttt{0} \mid \ldots \mid \texttt{9} \mid \texttt{-} \mid \texttt{.} \mid \texttt{_} \mid \texttt{\~{}} \mid \texttt{?} \mid \texttt{\#} \\
\langle\text{Prozentkodierung}\rangle &\models \; \texttt{\%}\,\langle\text{h}\rangle\,\langle\text{h}\rangle \\
\langle\text{h}\rangle &\models \; \texttt{0} \mid \ldots \mid \texttt{9} \mid \texttt{a} \mid \ldots \mid \texttt{f} \mid \texttt{A} \mid \ldots \mid \texttt{F} \\
\langle\text{s}_\varepsilon\rangle &\models \; \langle\text{c}\rangle \mid \varepsilon
\end{aligned}$$

Jede Komponente beginnt mit einem Schrägstrich auf diesen dann eine Zeichenkette folgt. Komponenten sind ASCII-kodiert und daher wird jedes Zeichen mit einem Byte kodiert. ASCII-Steuerzeichen und Bytes, die kein ASCII-Zeichen kodieren, werden in *Prozentkodierung* dargestellt. Beispiel 2.1 zeigt Beispiele für Prozentkodierung.

BEISPIEL 2.1 – *Prozentkodierung:* (i) *Das ASCII-Steuerzeichen* `NUL` *ist nicht druckbar. Es wird prozentkodiert als* `%00` *dargestellt.* (ii) *Leerzeichen in Namen werden nicht als Leerzeichen, sondern als* `%20` *dargestellt.* (iii) *Der Schrägstrich wird als* `%2F` *dargestellt, wenn er nicht den Anfang einer Komponente markiert.* (iv) *Das Zeichen „ö" ist kein ASCII-Zeichen, daher wird es als* `%C3%B6` *dargestellt. Die UTF-8 Kodierung von ö ist* $C3B6_{16}$.

Der CCN-Name, in dieser Arbeit auch kurz *Name* genannt, wird wie die Komponente ebenfalls mit Hilfe der BNF in Definition 2.8 definiert. Darüber hinaus werden in der Definition noch die Begriffe *Präfix* und *Suffix* von Namen eingeführt.

DEFINITION 2.8 – *CCN-Name, Präfix und Suffix:*

$$\begin{aligned}
\langle\text{CCN-Name}\rangle &\models \; \langle\text{Root}\rangle \mid \langle\text{Präfix}\rangle\,\langle\text{Suffix}\rangle \mid \langle\text{Präfix}\rangle \mid \langle\text{Suffix}\rangle \\
\langle\text{Root}\rangle &\models \; \texttt{/} \\
\langle\text{Präfix}\rangle &\models \; \langle\text{Komponente}\rangle\,\langle\text{Präfix}\rangle \mid \langle\text{Komponente}\rangle \\
\langle\text{Suffix}\rangle &\models \; \langle\text{Komponente}\rangle\,\langle\text{Suffix}\rangle \mid \langle\text{Komponente}\rangle
\end{aligned}$$

Für das Nichtterminal ⟨Komponente⟩ *gilt Definition 2.7.*

Ein CCN-Name ist entweder der Root-Name, der nur aus dem Schrägstrich besteht, oder aus Präfix und Suffix. Aufeinanderfolgende Komponenten eines Namens, von links nach rechts, sind der Präfix und aufeinanderfolgende Komponenten, von rechts nach links, sind der Suffix. Betrachtet man alle Komponenten eines Namens, so ist der Name sein eigener Präfix und Suffix.

Auf Namen ist weiterhin eine Ordnungsrelation definiert. Diese Ordnungsrelation wird *Shortlex-Ordnung* genannt. Die Shortlex-Ordnung wird folgendermaßen definiert:

Definition 2.9 – *Shortlex-Ordnung ($<_{SL}$): Sei C die Menge aller Komponenten, die mit der BNF aus Definition 2.7 erzeugt werden, und seien $a, b \in C$. Weiterhin seien p_a, p_b die kürzesten Präfixe von a, b, so dass gilt $p_a \neq p_b$ und $|\cdot|_{Byte}$ die Länge eines CCN-Namens in Byte, dann ist $a <_{SL} b$, wenn* (i) $|p_a|_{Byte} < |p_b|_{Byte}$ *oder* (ii) $|p_a|_{Byte} = |p_b|_{Byte}$ *und p_a in der lexikographischen (alphabetischen) Ordnung vor p_b steht.*

Mit den Begriffen Komponente, Name, Präfix, Suffix und Shortlex-Ordnung werden im Folgenden das Knotenmodell und die Nachrichtenverarbeitung von CCN erklärt.

2.4.2 Knotenmodell

Jeder CCN-Knoten folgt dem CCN-Knotenmodell von Jacobson et al. [85]. Wesentliches Element des CCN-Knotenmodells ist der *CCN-Dämon* (kurz Dämon), der für die Weiterleitung der Nachrichten zuständig ist. Der Dämon enthält die drei Datenstrukturen: *Content Store* (CS), *Pending Interest Table* (PIT) und *Forwarding Information Base* (FIB), die weiter unten erklärt werden. CCN definiert zwei Nachrichtentypen: *Interest* und *Content Object*. Mit dem Interest werden Content Objects angefragt. Der Interest enthält einen Namen und Informationen, um die Ausbreitung des Interests zu steuern. Das Content Object enthält ebenfalls einen Namen, Steuerungsinformationen sowie die Daten (Content). Über die *Faces*, eine Verallgemeinerung von Interfaces, tauschen Anwendungen und Netzwerkschnittstellen Interests oder Content Objects aus. Ein Knoten nach dem CCN-Knotenmodell in Abbildung 2.7 enthält einen Dämon. Der Dämon wiederum enthält die Datenstrukturen Content Store, Pending Interest Table und Forwarding Information Base. Die Faces sind die Schnittstellen zum Nachrichtenaustausch zwischen Dämon und den Anwendungen beziehungsweise den Netzwerkschnittstellen. Über die Netzwerkschnittstellen tauscht der Knoten Interests und Content Objects (CO) mit anderen Knoten aus.

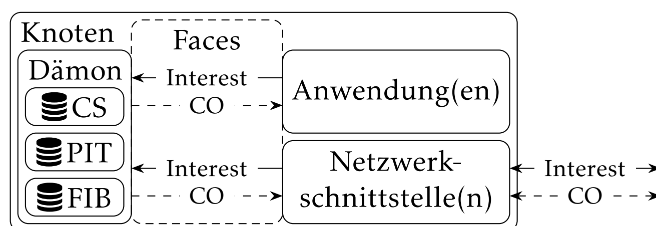
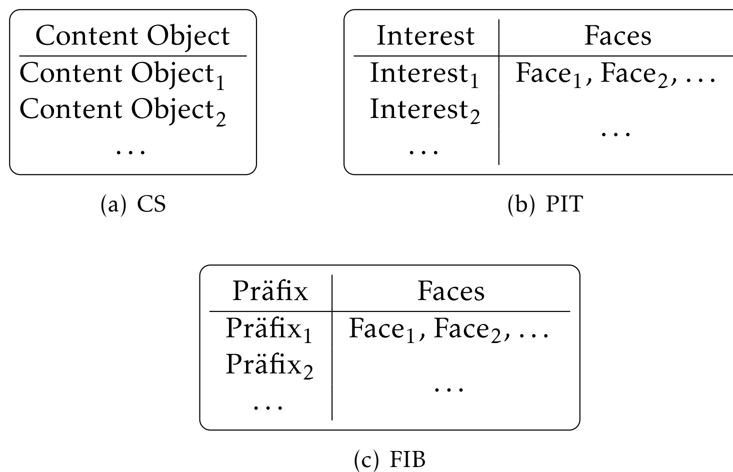


Abbildung 2.7 – *CCN-Knotenmodell*

2.4.3 CCN-Datenstrukturen

Im Knotenmodell in Abschnitt 2.4.2 wurden die drei Datenstrukturen von CCN eingeführt. Als Grundlage der Nachrichtenverarbeitung, die im folgenden Abschnitt 2.4.4 eingeführt wird, werden die Datenstrukturen hier detaillierter vorgestellt. Weiterhin stellt dieser Abschnitt auch eine Grundlage für Kapitel 4 dar, da dort Optimierungsstrategien für die Forwarding Information Base vorgeschlagen werden.

Die CCN-Datenstrukturen in Abbildung 2.8 sind wie Tabellen organisiert. Im Content Store (Abbildung 2.8(a)) werden Content Objects nach ihrem Namen in absteigender Shortlex-Ordnung gespeichert. Die Pending Interest Table (Abbildung 2.8(b)) speichert die Interests mit einer Referenz der Faces, von denen die Interests empfangen wurden. Die Forwarding Information Base (Abbildung 2.8(c)) speichert Namen mit Referenzen auf Faces. Die Namen in der Forwarding Information Base werden auch als *Präfixe* bezeichnet, da Präfixe von Interests mit den Präfixen in der Forwarding Information Base verglichen werden. Die Präfixe in der Forwarding Information Base sind absteigend nach der Shortlex-Ordnung sortiert.



(a) CS

(b) PIT

(c) FIB

Abbildung 2.8 – *Aufbau der CCN-Datenstrukturen*

2.4.4 NACHRICHTENVERARBEITUNG

In Abschnitt 2.4.2 wurde der Nachrichtenfluss von Interests und Content Objects bereits angedeutet. Wie die Verarbeitung im Dämon funktioniert, wird hier erklärt. Bei den Abbildungen in 2.9 und 2.10 werden die CCN-Datenstrukturen dargestellt. Rechts von den Datenstrukturen sind die Faces dargestellt. Die Pfeile zeigen den Nachrichtenfluss von Interests als durchgezogene Pfeile und Content Objects als gestrichelte Pfeile. Nummern an den Pfeilen markieren die Schritte. Der Nachrichtenfluss zu der kleinen Mülltonne links der Datenstrukturen symbolisiert Nachrichten, die verworfen werden.

Bei CCN wird ein Content Object nur nach einem Interest verschickt, der vorher empfangen wurde. Aus diesem Grunde wird zuerst die Verarbeitung eines Interest in Abbildung 2.9 betrachtet.

Von Face$_1$ in Abbildung 2.9 wird ein Interest am Dämon empfangen. In *Schritt 1* wird zuerst im Content Store nach einem passenden Content Object gesucht. Die Prüfung auf ein passendes Content Object im Content Store erfolgt in absteigender Shortlex-Ordnung der Content-Object-Namen. Ist der Name des Interests der Präfix eines Content-Object-Namens, so wird dieses Content Object im *Schritt 2(a)* als Antwort wieder an das Face zurückgeschickt, von dem der

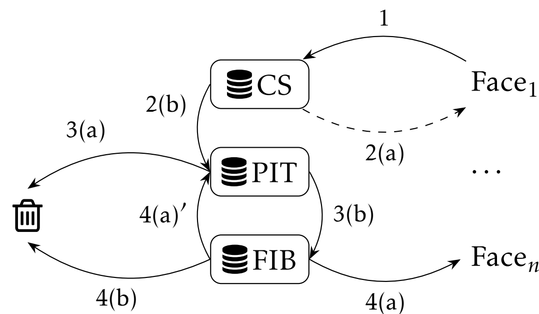


ABBILDUNG 2.9 – *Interest-Verarbeitung im Dämon*

Interest kam. (Der Vollständigkeit halber sei hier erwähnt, dass der Interest den Parameter *ChildSelector* hat, mit dem gesteuert wird, ob die Content Objects in absteigender oder aufsteigender Ordnung im Content Store verglichen werden; Standard ist die absteigende Ordnung.)

Gibt es kein passendes Content Object im Content Store, dann wird die Pending Interest Table in *Schritt 2(b)* durchsucht, ob es den Interest bereits gibt. Beim Vergleich in der Pending Interest Table wird auf Gleichheit der Namen geprüft. Der Interest wird in *Schritt 3(a)* verworfen, wenn der Interest bereits in der Pending Interest Table vorhanden ist. Allerdings wird in der Pending Interest Table dem bereits enthaltenen Interest das Face hinzugefügt, wenn das Face noch nicht hinzugefügt wurde.

War die Suche in der Pending Interest Table nicht erfolgreich, dann wird in der Forwarding Information Base im *Schritt 3(b)* nach einem passenden Eintrag gesucht. Bei der Suche nach einem passenden Eintrag wird wieder nach dem längsten gemeinsamen Präfix gesucht. Wenn es einen passenden Eintrag in der Forwarding Information Base gibt, dann wird der Interest an die korrespondierenden Faces des Präfix in *Schritt 4(a)* weitergeleitet und der Interest wird in *Schritt 4(a)′* in der Pending Interest Table gespeichert. Gibt es keinen passenden Eintrag in der Forwarding Information Base, dann wird der Interest in *Schritt 4(b)* verworfen. Bei der Weiterleitung von Interests mittels der Forwarding Information Base werden Interests nie an das Face zurückgeschickt, von dem sie empfangen wurden, um der Zirkulation von Interests im Netz vorzubeugen.

Angenommen, in Abbildung 2.9 von $Face_n$ kommt ein Content Object zurück, dann werden die Verarbeitungsschritte in Abbildung 2.10 ausgeführt. In *Schritt 1* wird zuerst im Content Store geprüft, ob das Content Object bereits vorhanden ist. Hierzu werden die Namen des empfangenen Content Object mit denen im Content Store auf Gleichheit geprüft. Wurde ein Content Object mit dem gleichen Namen gefunden, wird das Content Store in *Schritt 2(a)* verworfen.

Handelt es sich um ein neues Content Object, wird die Pending Interest Table in *Schritt 2(b)* nach einem passenden Interest durchsucht. Hierbei wird der Interest in der Pending Interest Table gewählt, der den längsten gemeinsamen Präfix mit dem Content Object hat. Gibt es einen passenden Interest in der

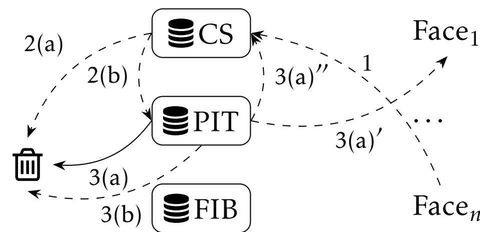


ABBILDUNG 2.10 – *Content-Object-Verarbeitung im Dämon*

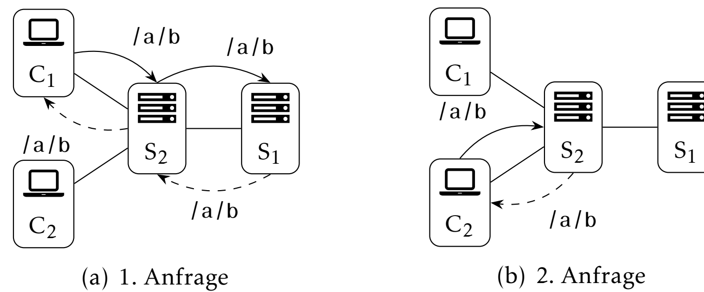
Pending Interest Table, wird in *Schritt 3(a)* das Content Object zu allen Faces weitergeleitet, von denen der Interest empfangen wurde. Weiterhin wird der Interest in *Schritt 3a′* aus der Pending Interest Table entfernt und das Content Object in *Schritt 3(a)″* im Content Store gespeichert. Gibt es keinen passenden Interest in der Pending Interest Table, so wird das Content Object in *Schritt 3(b)* ebenfalls verworfen. Über die Einträge in der Pending Interest Table werden die Content Objects zu den Netzknoten und Anwendungen geleitet, die den passenden Interest geschickt haben.

Content Objects werden nach einer, in den Steuerungsinformationen des Content Objects gespeicherten individuellen Lebenszeit, wieder aus dem Content Store entfernt. Ebenso werden Interests nach einer bestimmten Zeit aus der Pending Interest Table entfernt, sollte kein Content Object in Schritt 3(a) empfangen werden.

Bei der Verarbeitung von Nachrichten wird deutlich, dass Content (dt. Inhalt) adressiert wird und nicht wie bei IP spezielle Netzknoten oder Netzwerkschnittstellen. Daher wird dieser Ansatz als *inhaltszentrisch* bezeichnet. In Abgrenzung zum Begriff inhaltszentrisch wird der Ansatz von IP als *knotenzentrisch* bezeichnet.

In inhaltszentrischen Netzen werden Anfragen nach dem gleichen Inhalt nicht von der Quelle (Ursprung) des Inhalts, sondern aus dem Content Store eines Netzknotens zwischen der Quelle und dem Anfragenden bedient. Ein einfaches Beispiel von Anfragen bei CCN ist in Abbildung 2.11 dargestellt. Die Abbildung zeigt ein Netz mit vier Knoten, zwei Servern, S_1 und S_2, die Content Objects speichern und zwei Clients C_1 und C_2. Die Links zwischen den Knoten sind wieder als durchgezogene Linien dargestellt. Interests sind als durchgezogene Pfeile dargestellt, Content Objects als gestrichelte Pfeile.

In Abbildung 2.11(a) fragt C_1 den Content mit dem Namen `/a/b` an. Dazu schickt C_1 einen Interest mit dem Namen `/a/b` an S_2. S_2 ist nicht in der Lage den Interest zu beantworten und schickt diesen weiter an S_1. Die Weiterleitung des Interests erfolgt bei C_1 und bei S_2 jeweils anhand der Forwarding Information Base. S_1 beantwortet die Anfrage und schickt ein Content Object zurück. Das Content Object hat denselben Namen wie der Interest. Der Interest wird anhand der Einträge in den Pending-Interest-Table-Einträgen zurück zu C_1 gesendet.



(a) 1. Anfrage

(b) 2. Anfrage

Abbildung 2.11 – *Wiederholte Anfrage nach einem Content Object im Netz*

Erreicht S_2 wie in Abbildung 2.11(b) eine zweite Anfrage nach Content mit dem Namen /a/b von C_2, so wird die Anfrage direkt aus dem Content Store von S_2 bedient, da das Content Object bei der ersten Anfrage im Content Store von S_2 gespeichert wurde. Das Zwischenspeichern hat laut Jacobson et al. [85] Vorteile, dass der Weg, den eine Nachricht im Netz zurücklegt, kürzer wird und implizit eine Lastverteilung erfolgt, da die Quelle des Inhalts weniger Anfragen erhält.

2.5 ZUSAMMENFASSUNG

In diesem Grundlagenkapitel wurden die wichtigsten Grundlagen zum *Internet* und zum *Internet der Dinge* eingeführt. Wesentliche Technologien im Kontext des Internet der Dinge sind drahtlose Sensornetze, die heute knotenzentrisch entworfen werden. Gleiches gilt für aktuelle Technologien in der Heimautomation oder für das Message Queue Telemetry Transport (MQTT) Protokoll.

In Abschnitt 2.3 wurden die Begriffe Dienst, Dienstnutzer und Dienstanbieter definiert. Diese Begriffe werden insbesondere in Kapitel 3 bei der Einführung und Entwicklung der namenszentrischen Dienste benötigt.

In Abschnitt 2.4 wurden die Grundlagen des inhaltszentrischen Ansatzes (engl. Content-Centric Networking, kurz CCN) eingeführt. Wichtige Grundlagen des inhaltszentrischen Ansatzes sind die Namen, Datenstrukturen *Content Store* (CS), *Pending Interest Table* (PIT) und *Forwarding Information Base* (FIB) sowie die Verarbeitung der beiden Nachrichtentypen *Interest* und *Content Object*, da sie generell wichtig zum Verstehen der Arbeit sind. Die Datenstrukturen und die Verarbeitung der Nachrichten sind insbesondere in Kapitel 4 wichtig. Dort wird die Verarbeitung für Namen und Nachrichten auf ressourcenbeschränkten Geräten sowie Strategien zur Optimierung des Speichers der Forwarding Information Base vorgestellt. Das folgende Kapitel 3 führt in das Konzept der namenszentrischen Dienste ein.

KAPITEL 3

NAMENSZENTRISCHE DIENSTE

Auf Grundlage der Definition von Dienst in Kapitel 2 wird in diesem Kapitel das Konzept der namenszentrischen Dienste – einer Dienstarchitektur für das inhaltszentrische Internet der Dinge – entwickelt. Der Begriff *namenszentrisch* ist im Rahmen dieser Arbeit entstanden und verdeutlicht, dass Namen statt Inhalte im Fokus stehen. Das Konzept der namenszentrischen Dienste ist ein wissenschaftlicher Beitrag dieser Arbeit. Dazu werden in Abschnitt 3.1 verwandte Arbeiten vorgestellt, die im Konzept teilweise Verwendung finden und in einen neuen Kontext gesetzt werden.

Im Anschluss an die Vorstellung der verwandten Arbeiten werden *Herausforderungen* und *Potenziale* von inhalts-/namenszentrischen Ansätzen im Internet der Dinge in Abschnitt 3.2 herausgearbeitet. Insbesondere aus den Potenzialen wird in Abschnitt 3.3 schließlich das Konzept der namenszentrischen Dienste entwickelt.

3.1 VERWANDTE ARBEITEN

In diesem Abschnitt werden existierende Ansätze zur Realisierung von Diensten im knotenzentrischen Internet vorgestellt. Die hier entwickelten namenszentrischen Dienste orientieren sich konzeptionell an den existierenden Ansätzen, insbesondere an SOAP.

Neben REST ist die wichtigste Implementierung für serviceorientierte Architekturen (engl. Service-Oriented Architectures, kurz SOA) [100] aktuell das Simple Object Access Protocol (SOAP) [49]. Mit SOAP lassen sich Dienste über entfernte Methodenaufrufe (engl. Remote Procedure Call (RPC)) aufrufen. Der Dienstnutzer ruft bei SOAP eine Methode auf, indem er eine SOAP-Nachricht an den Dienstanbieter verschickt. Beim Empfang der Nachricht vom Dienstnutzer ermittelt der Dienstanbieter den Rückgabewert der Methode und schickt den

Rückgabewert in einer Nachricht zurück an den Dienstnutzer. Beispiele der Umsetzung von RPC mit SOAP finden sich in „Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI“ von Curbera et al. [59].

Die Nachrichten bei RPC sind bei SOAP in XML kodiert. XML [52] ist eine textbasierte Auszeichnungssprache für strukturierte Daten. Bei RPC folgen Methodenaufrufe und Rückgabewerte einer vorgegebenen XML-Struktur, einem sogenannten Schema [70, 162, 47]. Die vorgegebene Struktur der Schnittstelle erlaubt eine automatisierte werkzeugunterstützte Erstellung von Quellcode für die Kommunikation und somit eine einfachere Entwicklung.

Die Beschreibung der öffentlichen Schnittstelle erfolgt bei SOAP mit der Web Service Description Language (WSDL) [55]. WSDL ist die Beschreibung der Schnittstelle und ist XML kodiert. In WSDL werden die Art der Kommunikation, Methoden und Datentypen definiert.

Für die Auffindbarkeit von SOAP basierten Diensten wurde ein zentrales Verzeichnis mit dem Universal Description Discovery and Integration (UDDI) [39] vorgeschlagen, welches sich aber nicht durchgesetzt hat, wie Pedrinaci und Domingue in „Web Services are Dead. Long Live Internet Services“ [126] schreiben. Anstatt einer dynamischen Zuweisung über UDDI werden bei Webservices den Dienstnutzern die Dienstanbieter fest zugewiesen.

Mit dem Aufkommen von eingebetteten Systemen wurden Device Profile for Web Services (DPWS) [155] entwickelt, um SOAP basierte Dienste auch auf eingebetteten Systemen anzubieten. In DPWS gibt es mit dem Hosting Service einen dedizierten Dienst, der die Auffindbarkeit von Diensten ermöglicht und Dienstbeschreibungen für Dienste im Netz bereitstellt. Der Hosting Service wird zentral von einem Gerät bereitgestellt. Er realisiert eine Zuordnung von Diensten zu Geräten im Netz und ist somit inhärent knotenzentrisch. Da DPWS auf SOAP aufbaut und die XML kodierten Nachrichten damit recht groß sind, ist es fraglich, ob DPWS für drahtlose Sensornetze geeignet ist.

Wie SOAP oder DPWS sehen namenszentrische Dienste ebenfalls eine Dienstbeschreibung und eine Werkzeugunterstützung vor. Ein Vorteil von namenzentrischen Diensten ist ihre Auffindbarkeit, da die Dienste adressiert werden und nicht die Knoten, über die sie erreichbar sind. Weitere Gemeinsamkeiten und Unterschiede von namenszentrischen Diensten und den existierenden Ansätzen werden in Abschnitt 3.3 bei der Entwicklung des Konzepts der namenszentrischen Dienste erläutert.

Neben SOAP im knotenzentrischen Internet gibt es von Braun et al. mit „Service-Centric Networking“ [50] Ideen zur Realisierung von Diensten auf der Basis von CCNx. Im Gegensatz zu den hier vorgestellten namenszentrischen Diensten ist bei Service-Centric Networking eine Verwendung auf ressourcenbeschränkten Geräten im Internet der Dinge nicht vorgesehen. Weiterhin werden Konzepte wie Dienstbeschreibung und Werkzeugunterstützung in Service-Centric Networking ebenfalls nicht betrachtet.

3.2 INHALTSZENTRISCHE ANSÄTZE IM INTERNET DER DINGE

Bevor das Konzept der namenszentrischen Dienste eingeführt wird, wird hier die Frage beantwortet, warum inhaltszentrische Ansätze im Internet der Dinge wichtig sind. Im Rahmen der Arbeit werden zwei Annahmen getroffen, die die Frage unterschiedlich beantworten:

1. Wenn das zukünftige Internet inhaltszentrisch wird, dann wird auch das Internet der Dinge inhaltszentrisch.
2. Wenn inhaltszentrische Ansätze vorteilhaft für das Internet der Dinge sind, dann werden sich inhaltszentrische Ansätze im Internet der Dinge durchsetzen und eventuell das zukünftige Internet dahingehend beeinflussen, dass auch dieses inhaltszentrisch wird.

Die Annahmen sind in Abbildung 3.1 grafisch dargestellt. Stellt man sich das Internet der Dinge als Teilmenge des Internet vor, so sind die Annahmen in den Teilmengen entsprechend verortet. Die erste Annahme befindet sich in der Teilmenge *Internet*, die zweite Annahme in der Teilmenge *Internet der Dinge* (IoT). Bei jeder Annahme geht man davon aus, dass die jeweilige Teilmenge, in dem sich die Annahme befindet, dem inhaltszentrischen Ansatz folgt und dass sich die Eigenschaft, inhaltszentrisch zu sein, sich jeweils auf die andere Teilmenge überträgt, wenn es die andere schon ist. Die beiden Annahmen werden im Folgenden mit *Internet-getrieben* beziehungsweise *IoT-getrieben* bezeichnet.

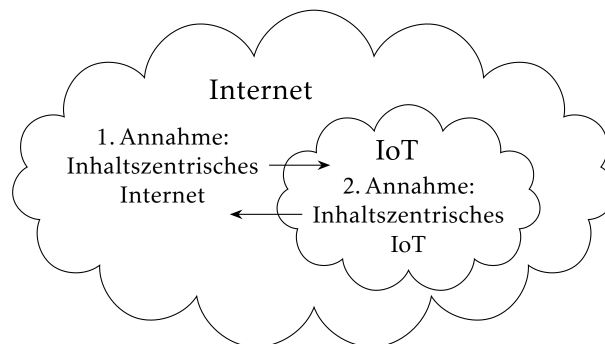


Abbildung 3.1 – *Annahmen für ein inhaltszentrisches Internet der Dinge*

Die Annahme Internet-getrieben folgt der pragmatischen Überlegung, dass das Internet eine homogene Architektur hat. Diese Annahme ist somit vergleichbar mit der Argumentation der 6LoWPAN-Befürworter, die mit 6LoWPAN das Ziel verfolgen, eine homogene Architektur des Internets sicherzustellen (vgl. Argumentation von Vasseur und Dunkels [165] in Abschnitt 2.2). Geräte im Internet der Dinge brauchen daher die technischen Voraussetzungen, um sich in das inhaltszentrische Internet zu integrieren (vgl. Punkt 2 von Definition 2.1).

Annahme IoT-getrieben folgt den visionären Überlegungen der Befürworter der inhaltszentrischen oder der Clean-Slate-Ansätze (vgl. Content Centric Networking von Jacobson et al. oder die Argumentation für Clean-Slate-Ansätze

bei Rexford und Dovrolis [137]). Ein Argument für diese Annahme ist, dass man insbesondere bei Sensornetzen an den Daten interessiert ist, und nicht an den Sensorknoten und daher inhaltszentrische Ansätze hier von Vorteil sind. Dieses Argument steht gegen den bisherigen Ansatz, das Internet der Dinge knotenzentrisch zu entwerfen.

Beide Annahmen werden in dieser Arbeit als gleichwertig angesehen. Argumente für und gegen die jeweilige Annahme werden in den folgenden Unterkapiteln herausgearbeitet, wo *Herausforderungen* und die *Potenziale* inhaltszentrischer Ansätze im Internet der Dinge kritisch diskutiert werden. Unabhängig von welcher Annahme man ausgeht, zeigt diese Arbeit, wie man mit dem inhaltszentrischen Ansatz Dienste im Internet der Dinge realisiert.

3.2.1 HERAUSFORDERUNGEN

Die Integration von ressourcenbeschränkten Geräten in das inhaltszentrische Internet der Dinge ist mit Herausforderungen verbunden. Um die Herausforderungen bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge herauszuarbeiten, wird zunächst der knotenzentrische Ansatz im Internet der Dinge betrachtet. Dabei werden bestimmte Aspekte identifiziert, die bei dem Vergleich des knotenzentrischen mit dem inhaltszentrischen Ansatz gegenübergestellt werden. Im Anschluss werden die Herausforderungen anhand der Aspekte, unter denen der knotenzentrische und der inhaltszentrische Ansatz verglichen wurden, herausgearbeitet.

Eine Herausforderung bei der Integration des knotenzentrischen Ansatzes ist, dass die Paketgröße in drahtlosen Ad-hoc-Netzen beschränkt ist. Ein Rahmen ist beispielsweise bei IEEE 802.15.4 maximal 127 Byte groß. IPv6 setzt voraus, dass eine minimale Paketgröße von 1280 Byte unterstützt wird (siehe RFCs von IPv6 [62] und 6LoWPAN [92]). Die 6LoWPAN Adaptionsschicht ermöglicht den Transport von 1280 Byte großen IPv6 Paketen über IEEE 802.15.4, indem sie große Pakete fragmentiert und Verfahren zur Header-Komprimierung implementiert.

Laut der Befürworter von 6LoWPAN hat der Einsatz von IPv6 trotz der Herausforderungen Potenziale [165]:

- Der knotenzentrische Ansatz, den auch 6LoWPAN verfolgt, ist den Anwendern (z. B. Entwicklern und Administratoren) bekannt.
- Die semantische Kompatibilität von 6LoWPAN zu IPv6 ermöglicht eine transparente Integration von drahtlosen Geräten in das Internet.
- Der knotenzentrische Ansatz ist neutral gegenüber den Anwendungen, d. h. er macht keine Annahmen über die Anwendungen und deren Kommunikation [165]. Dadurch ist der knotenzentrische Ansatz flexibel.

Herausforderung und Potenziale der Anwendung von IPv6 im Internet der Dinge geben Hinweise auf die zu betrachtenden Aspekte, die für eine Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge problematisch sind. Vergleicht man den knotenzentrischen mit dem inhaltszentrischen Ansatz, so

zeigt sich, dass diese *konzeptionell unterschiedlich* sind. Der konzeptionelle Unterschied zeigt sich insofern, dass der knotenzentrische Ansatz unabhängig von Anwendungen entwickelt wurde, wie Vasseur und Dunkels [165] schreiben. Der inhaltszentrische Ansatz hingegen wurde für die Verteilung von Inhalten im Internet konzipiert, wie diverse Arbeiten [86, 53, 148] aus diesem Kontext verdeutlichen.

Für die *Dienstentwicklung* gibt es bei den knotenzentrischen Ansätzen etablierte Lösungen, die auch im Internet der Dinge eingesetzt werden, wie beispielsweise in „A lightweight SOAP over CoAP Transport Binding for Resource Constraint Networks" von Moritz et al. [111] beschrieben. Bei den inhaltszentrischen Ansätzen gibt es so eine Lösung, insbesondere für die Anwendung im Internet der Dinge, bisher nicht. Die numerischen Adressen beim knotenzentrischen Ansatz werden beim inhaltszentrischen Ansatz durch Namen ersetzt. Zwar gibt es mit dem Domain Name System eine Methode, um IP-Adressen auf Namen abzubilden, aber in den Nachrichten werden beim knotenzentrischen Ansatz weiterhin IP-Adressen verwendet. Beim inhaltszentrischen Ansatz hingegen werden, wie in Abschnitt 2.4.2 dargestellt, auch in den Nachrichten Namen verwendet. Im Gegensatz zu den numerischen Adressen, die bei IPv4 32 Bit und bei IPv6 128 Bit groß sind, sind Namen in ihrer Größe theoretisch nicht beschränkt.

Ein weiterer Aspekt ist das Messaging Pattern. Der englische Begriff Messaging Pattern oder auch Message Exchange Pattern (MEP) bedeutet frei übersetzt „Nachrichtenaustauschmuster". Da der englische Begriff gebräuchlich ist, wird dieser in der Arbeit verwendet.

Der Aspekt des Messaging Pattern wird betrachtet, da sich hier der inhaltszentrischen vom knotenzentrischen Ansatz unterschiedet. So sieht der inhaltszentrische Ansatz ein striktes Anfrage/Antwort-Messaging-Pattern vor, da ein Content Object nur mit einem Interest angefragt wird, wie in Abschnitt 2.4.4 beschrieben. Der knotenzentrische Ansatz hingegen sieht kein bestimmtes Messaging Pattern vor.

Zusammenfassend ergeben sich folgende Aspekte, unter denen der inhaltszentrische Ansatz mit dem knotenzentrischen gegenübergestellt werden: *(i)* der *konzeptionelle Unterschied*, *(ii)* die Entwicklung von Diensten (*Dienstentwicklung*), *(iii)* die *Adressierung*, *(iv)* das *Messaging Pattern*. Bei der Herausarbeitung der Aspekte wurden auch die Unterschiede zwischen dem knotenzentrischen und dem inhaltszentrischen Ansatz deutlich, die in Tabelle 3.1 noch einmal zusammengefasst werden.

KONZEPTIONELLE UNTERSCHIEDE

Wie Eingangs in Abschnitt 3.2.1 dargestellt, ist der knotenzentrische Ansatz mit dem Ziel entwickelt worden, unabhängig von Anwendungen zu sein. Ebenso wurde eingangs erwähnt, dass der inhaltszentrische Ansatz für die Verteilung von Inhalten im Netz entwickelt worden ist, das eine spezielle Anwendung darstellt. Somit unterscheiden sich die Ziele des knotenzentrischen und des inhaltszentrischen Ansatzes grundlegend. Die Anwendungsunabhängigkeit des knotenzentrischen Ansatzes sowie die Verteilung von Inhalten beim inhaltszentrischen Ansatz werden im Folgenden an Beispielen erläutert.

Aspekt	Knotenzentrisch	Inhaltszentrisch
Konzeptionelle Unterschiede	Anwendungs-unabhängig	Verteilung von Inhalten
Dienstentwicklung	SOAP, REST, CoAP	–
Adressierung	DNS/IP-Adressen, URLs, Ports	Namen
Messaging Pattern	Beliebig	Anfrage/Antwort

Tabelle 3.1 – *Vergleich der Aspekte des knotenzentrischen mit dem inhaltszentrischen Ansatz*

Ist ein Anwender im knotenzentrischen Internet an Inhalten interessiert, schickt er beispielsweise über den Browser eine HTTP-Anfrage an den Server. In dem Uniform Ressource Locator (URL) [42] der HTTP-Anfrage befindet sich implizit die Adresse des Servers. Der Anwender kommuniziert daher durch die URL mit einem bestimmten Knoten, dem Server, im Internet auf dem die Inhalte abrufbar sind.

Sind weitere Anwender im knotenzentrischen Internet am selben Inhalt interessiert, so kontaktieren sie alle diesen Server. Knoten, wie Router im Inneren des knotenzentrischen Internet, leiten dieselben Inhalte bei wiederholten Anfragen auch wiederholt weiter. Hier zeigt sich die Anwendungsunabhängigkeit des knotenzentrischen Ansatzes, da Nutzeranfragen nach Inhalten in dem obigen Beispiel genauso behandelt werden, wie jede andere Kommunikation im Netz. Ein Beispiel für eine andere Kommunikation im Internet ist eine Ende-zu-Ende-Kommunikation bei Internettelefonie. Diese unterscheidet sich von der Kommunikation zur Abfrage von Inhalten dadurch, dass die Daten, die bei der Internettelefonie entstehen, nur für die beiden Endpunkte der Verbindung von Interesse sind und dass die Daten nur eine kurze Lebensdauer haben. Die Gemeinsamkeit aller Arten von Kommunikation im Internet ist das Ende-zu-Ende-Paradigma. Beim Ende-zu-Ende-Paradigma im Internet werden Daten von einem Ende zum anderen Ende geschickt, was nach Vasseur und Dunkels [165] alle elementaren Bedürfnisse der Kommunikation im Internet erfüllt. Dass beim Ende-zu-Ende-Paradigma im Internet, insbesondere bei IP, keine Annahmen darüber gemacht werden, wie lange Daten aktuell sind und ob sie vielleicht für andere interessant sind, zeigt die Anwendungsunabhängigkeit des knotenzentrischen Ansatzes.

Das inhaltszentrische Internet folgt einem grundlegend anderen Ansatz. Anfragen adressieren Daten beziehungsweise Inhalte und nicht Knoten. Interessiert sich ein Anwender im inhaltszentrischen Internet für einen bestimmten Inhalt, so schickt er eine Anfrage mit einem Uniform Resource Identifier (URI) [42], das heißt, einem eindeutigen Namen, der den gewünschten Inhalt eindeutig identifiziert. Knoten, die die Anfrage empfangen und eine Kopie des angefragten Inhalts im Content Store zwischenspeichern, schicken den Inhalt zu dem anfragenden Knoten zurück (vgl. Wiederholte Anfrage nach einem Content Object im Netz in Abbildung 2.11 auf Seite 32).

Der Vorteil des inhaltszentrischen Ansatzes ist laut Jacobson et al. [86], dass der Inhalt für den Anfragenden schneller verfügbar ist, da der von einem näherliegenden Knoten aus dem Cache bezogen wird und nicht von dem Server, von dem der Inhalt ursprünglich kam. An dieser Stelle sei darauf hingewiesen, dass Caching kein Alleinstellungsmerkmal des inhaltszentrischen Ansatzes ist. Caching von Inhalten wird auch im knotenzentrischen Netzen eingesetzt. HTTP-Proxies sind beispielsweise in der Lage, Inhalte in Caches zwischenzuspeichern, auch mit dem Ziel, Inhalte schneller verfügbar zu machen. Beim inhaltszentrischen Ansatz ist Caching durch den Content Store ein elementarer Bestandteil und die Caches sind im gesamten Internet verteilt, insbesondere auf Routern. Für den knotenzentrischen Ansatz hingegen ist Caching mit HTTP-Proxies optional. HTTP-Proxies werden darüber hinaus nur in privaten Netzen, vor allem beim Übergang vom lokalen zum Netz des Internet Service Providers eingesetzt, und nicht auf Routern im Internet.

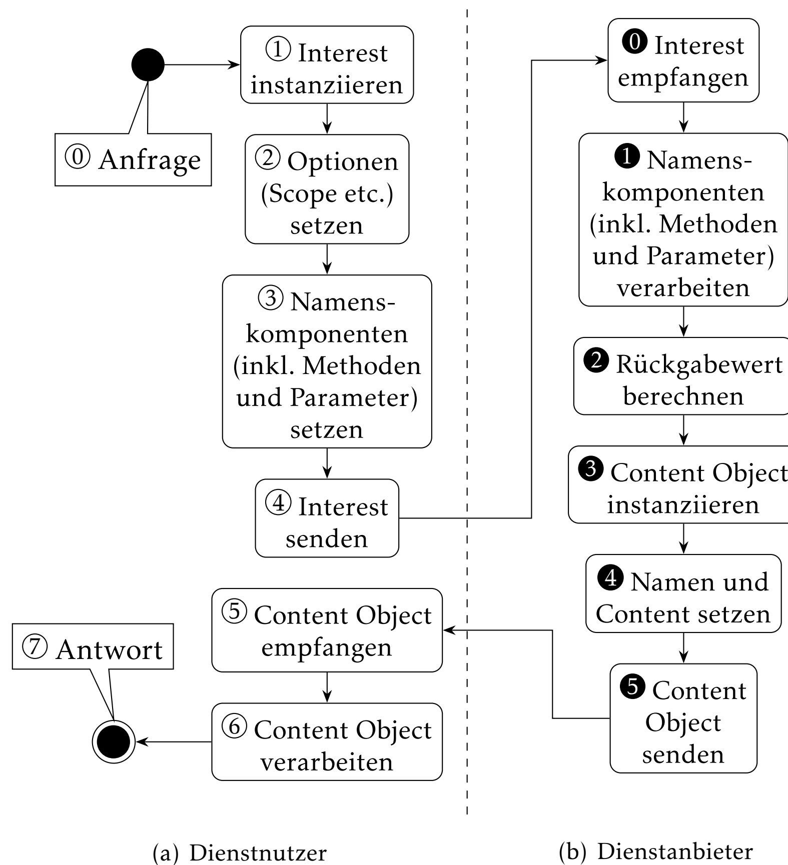
Ob der inhaltszentrische Ansatz immer vorteilhaft ist, ist insbesondere für das Internet der Dinge, umstritten. Baccelli et al. [35] bemerken, dass Inhalte im Internet der Dinge, wie beispielsweise Sensordaten, stets aktuell zu halten sind. Die Aktualität der Inhalte im Internet der Dinge steht mit dem Caching des inhaltszentrischen Ansatzes in Konflikt. Weiterhin bemerken Baccelli et al., dass es im Internet der Dinge einen Bedarf an spontaner Kommunikation gibt, wie beispielsweise der Ansteuerung von Aktuatoren. Laut Baccelli et al. lässt sich spontane Kommunikation besser mit dem knotenzentrischen Ende-zu-Ende-Paradigma umsetzen, was wiederum gegen die Annahme eines inhaltszentrischen Internet der Dinge wie in Abbildung 3.1 spricht.

Wie diese Arbeit sehen diverse andere Arbeiten [33, 166, 127, 60] im Internet der Dinge für die spontane Kommunikation Dienste vor und gehen daher davon aus, dass Dienste im Internet der Dinge eine bedeutende Rolle spielen. Mit Diensten lassen sich Anwendungen im Internet der Dinge realisieren, wie das Auslesen von Sensoren oder die Ansteuerung von Aktuatoren.

Leider steht die Entwicklung von Diensten beim inhaltszentrischen Ansatz bisher nicht im Fokus, sondern Verteilung von Inhalten im Netz. Eine Herausforderung bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge besteht somit darin, die einfache Entwicklung von Diensten ähnlich zu den existierenden Lösungen bei den knotenzentrischen Ansätzen zu ermöglichen (vgl. Verwandte Arbeiten in Abschnitt 3.1). Die Entwicklung von Diensten wird im folgenden Abschnitt betrachtet.

DIENSTENTWICKLUNG

Die Entwicklung von Diensten mit dem inhaltszentrischen Ansatz erfordert von den Entwicklern, denen die Entwicklung von Diensten mit dem knotenzentrischen Ansatz geläufig ist, ein Umdenken und eine Einarbeitung, wie in „Tool Chain for Application Development with Name-Centric Services“ [4] (Teubler, Hellbrück) beschrieben. Das Aktivitätsdiagramm in Abbildung 3.2, das die wesentlichen Aktivitäten bei der Entwicklung eines inhaltszentrischen Dienstes zeigt, verdeutlicht, dass ein Umdenken und eine Einarbeitung notwendig sind.



(a) Dienstnutzer (b) Dienstanbieter

ABBILDUNG 3.2 – *Aktivitäten bei der Entwicklung von Diensten mit CCN zur Realisierung eines entfernten Methodenaufrufs*

Folgende Aktivitäten entstehen bei einem entfernten Methodenaufruf (Initialzustand 0 in Abbildung 3.2(a)) durch einen Dienstnutzer: Erzeugen einer Instanz eines Interests (Aktivität 1). Dann werden Name und CCN-spezifische Optionen des Interests gesetzt (Aktivität 2) und dann wird die Namenskomponente der Anfrage (Methode) manuell zusammengestellt (Aktivität 3). Die manuelle Zusammenstellung von Namen ist fehleranfällig aufgrund möglicher Tippfehler oder falscher Reihenfolge der Parameter beim Methodenaufruf. Ist der Interest zusammengestellt, wird er gesendet (Aktivität 4).

Wird der Interest von einem Dienstanbieter empfangen (Aktivität 0 in Abbildung 3.2(b)), werden die folgenden Aktivitäten ausgeführt: Der Name des Interests wird verarbeitet, das heißt die Methode und die Parameter werden extrahiert (Aktivität 1). Dann wird das Ergebnis der Anfrage berechnet (Aktivität 2). Im Anschluss wird ein Content Object für den Rückgabewert der Anfrage erstellt (Aktivität 3) und der Name und der Rückgabewert im Content Object gesetzt (Aktivität 4). Dann wird das Content Object gesendet (Aktivität 5).

Der Dienstnutzer empfängt das Content Object (Aktivität 5). Methodenname, Parameter und der Rückgabewert werden verarbeitet (Aktivität 6), um den Rückgabewert der Anfrage zuzuordnen (Endzustand 7).

Wie durch die Beschreibung verdeutlicht, ist die manuelle Entwicklung von Diensten sehr aufwändig und in den einzelnen Schritten können Fehler auftreten. Wegen der strukturellen Ähnlichkeit der Aktivitäten in Abbildung 3.2 sind Entwickler geneigt, Teile des Quellcodes für unterschiedliche Dienste zu kopieren und wiederzuverwenden (engl. Copy-and-Paste). Dieses Copy-and-Paste ist problematisch, da es zu subtilen Fehlern führt, weil in den jeweiligen Diensten kleine aber notwendige Änderungen am kopierten Quellcode übersehen werden. Li et al. [98] sowie Jablonski und Hou [84] bestätigen die Problematik von Copy-and-Paste-Fehlern. Die in dieser Arbeit vorgeschlagene Werkzeugunterstützung in Kapitel 5 beugt Copy-and-Paste-Fehlern bei der Implementierung von Dienstnutzern und Dienstanbietern vor und entlastet den Entwickler von Aufgaben, die mit der eigentlichen Implementierung des Dienstes nichts zu tun haben.

ADRESSIERUNG

Namen in inhaltszentrischen Ansätzen weisen den Inhalten eine Bedeutung oder *Semantik* zu. Beispielsweise erwartet man unter dem Namen `/com/pethome/cat` Inhalte zu Katzen, zum Beispiel Bilder oder Videos. Die Eigenschaft eines Namens, für Menschen verständlich zu sein, wird „sprechend“ genannt. Der Name in dem obigen Beispiel ist sprechend, da den meisten (internetaffinen) Menschen klar ist, welcher Inhalt damit gemeint ist. Aus diesem Grunde wurde für den knotenzentrischen Ansatz die Uniform Ressource Locator (URL) [43] geschaffen. Eine URL identifiziert den Ort einer Ressource in Form eines Namens. Die URL enthält auch den Hostnamen des Knotens, wo die Ressource zu finden ist. Mit dem Domain Name System werden Hostnamen auf IP-Adressen abgebildet, da es für Menschen leichter ist, sich Namen anstatt IP-Adressen zu merken.

Namen werden in den lokalen Datenstrukturen des Knotens Pending Interest Table (PIT), Content Store (CS) und Forwarding Information Base (FIB) gespeichert (vgl. Knotenmodell von CCN in Abbildung 2.7 auf Seite 28). Der limitierte Speicher von Sensorknoten stellt für die Speicherung von langen Namen eine Herausforderung dar. Namen sind auch Bestandteil von Nachrichten in inhaltszentrischen Netzen. Die beschränkten Rahmengrößen der Sicherungsschicht (vgl. Referenzmodelle für Netzwerkprotokolle in Abbildung 2.1) der verwendeten drahtlosen Übertragungstechnologien im Internet der Dinge stellen ebenfalls eine Herausforderung dar, da die Nachrichten aufgrund ihrer Größe nicht in einen Rahmen der IEEE 802.15.4 Sicherungsschicht passen. Passen Nachrichten nicht in die Rahmen der Sicherungsschicht, werden sie fragmentiert.

Lange Namen führen zur daher Fragmentierung von Paketen auf der Sicherungsschicht. Bei der Fragmentierung gibt es bei inhaltszentrischen Ansätzen im Vergleich zu knotenzentrischen Ansätzen eine einschränkende Besonderheit: Paketfragmente werden bei jedem empfangenden Knoten zwischengespeichert, um die Fragmente so weit zusammenzusetzen, dass der Name vollständig ist, um so eine Weiterleitungsentscheidung zu treffen, wie Weigel et al. in „Route-Over Forwarding Techniques in a 6LoWPAN “ [171] schreiben. Das Zwischenspeichern der Fragmente benötigt wiederum Speicher. Im Gegensatz zum inhaltszentrischen Ansatz unterliegen fragmentierte 6LoWPAN Pakete dieser Einschränkung nicht. Bei „Mesh-Under“ Routing von 6LoWPAN werden alle Fragmente anhand

der Routinginformationen des ersten Pakets weitergeleitet, wie in „6LoWPAN Demystified“ von Jonas Olsson [122] beschrieben.

Zusammengefasst stellen sprechende, lange Namen eine Herausforderung für ressourcenbeschränkte Geräte im Internet der Dinge dar, da sie Speicher in den Datenstrukturen und Platz in Nachrichten verbrauchen. Weiterhin stehen sprechende Namen im Widerspruch zum Internet der Dinge, da dort Maschinen miteinander kommunizieren (vgl. Definition 2.1 auf Seite 14).

Eine weitere Herausforderung ist die Verarbeitung von Namen. Beim inhaltszentrischen Ansatz werden fortlaufend Nachrichten, und damit Namen, auf den Knoten verarbeitet (vgl. Nachrichtenverarbeitung in Abbildung 2.9 auf Seite 30). Dabei werden die Namen der Pakete mit den Namen der Inhalte im Content Store, in der Pending Interest Table und in der Forwarding Information Base verglichen. In der Pending Interest Table werden die Namen auf Gleichheit geprüft, im Content Store und in der Forwarding Information Base wird ein Vergleich bezüglich der Shortlex-Ordnung von Namen vorgenommen. Diese Vergleiche effizient auf ressourcenbeschränkten Systemen auszuführen, stellt eine Herausforderung dar. Beim knotenzentrischen 6LoWPAN hingegen werden unter Nutzung der Header-Komprimierung in Abhängigkeit des Routing-Protokolls und des Zielnetzes entweder 6 Byte lange IEEE 802.15.4 MAC-Adressen oder, im äußersten Fall, 16 Byte lange IPv6-Adressen verglichen, wie in „6LoWPAN Demystified“ [122] dargestellt.

MESSAGING PATTERN

Im Folgenden wird der vierte Aspekt, die Messaging Pattern, aus Tabelle 3.1 auf Seite 38 behandelt. Zur Erinnerung, der inhaltszentrische Ansatz unterstützt nur das Anfrage/Antwort-Messaging-Pattern. Um zu zeigen, welche Messaging Pattern es bei der Realisierung von Diensten mit dem knotenzentrischen Ansatz gibt, werden die Messaging Pattern des Simple Object Access Protocol (SOAP) vorgestellt.

Das Simple Object Access Protocol, mit dem verteilte Dienstarchitekturen umgesetzt werden, definiert insgesamt acht Messaging Pattern: *(a)* In-Only, *(b)* Out-Only, *(c)* In-Out, *(d)* Out-In, *(e)* Robust In-Only, *(f)* Robust Out-Only, *(g)* In-Optional-Out, *(h)* Out-Optional-In.

In den Sequenzdiagrammen in Abbildung 3.3 sind die Messaging Pattern grafisch dargestellt. Sequenzdiagramme zeigen ausgehend von den Instanzen, in Abbildung 3.3 sind das Dienstnutzer und Dienstanbieter, eine vertikal verlaufende gepunktete Linie, die Lebenslinie. Die Lebenslinie zeigt an, wie lange die Instanz gültig ist. Die Rechtecke auf den Lebenslinien sind Aktivierungsbalken. Aktivierungsbalken zeigen die Zeit an, in dem eine Instanz über den Kontrollfluss verfügt, und aktiv am Nachrichtenaustausch beteiligt ist. Die Pfeile zeigen Richtung des Nachrichtenaustauschs zwischen Dienstanbieter und Dienstnutzer. Ein gestrichelter Pfeil ist entweder eine beliebige Nachricht oder eine Antwortnachricht, die auf eine Anfragenachricht geschickt wird. Eine Anfragenachricht wird mit einem durchgezogenen Pfeil dargestellt. Die in Abbildung 3.3 dargestellten Messaging Pattern werden im Folgenden kurz besprochen.

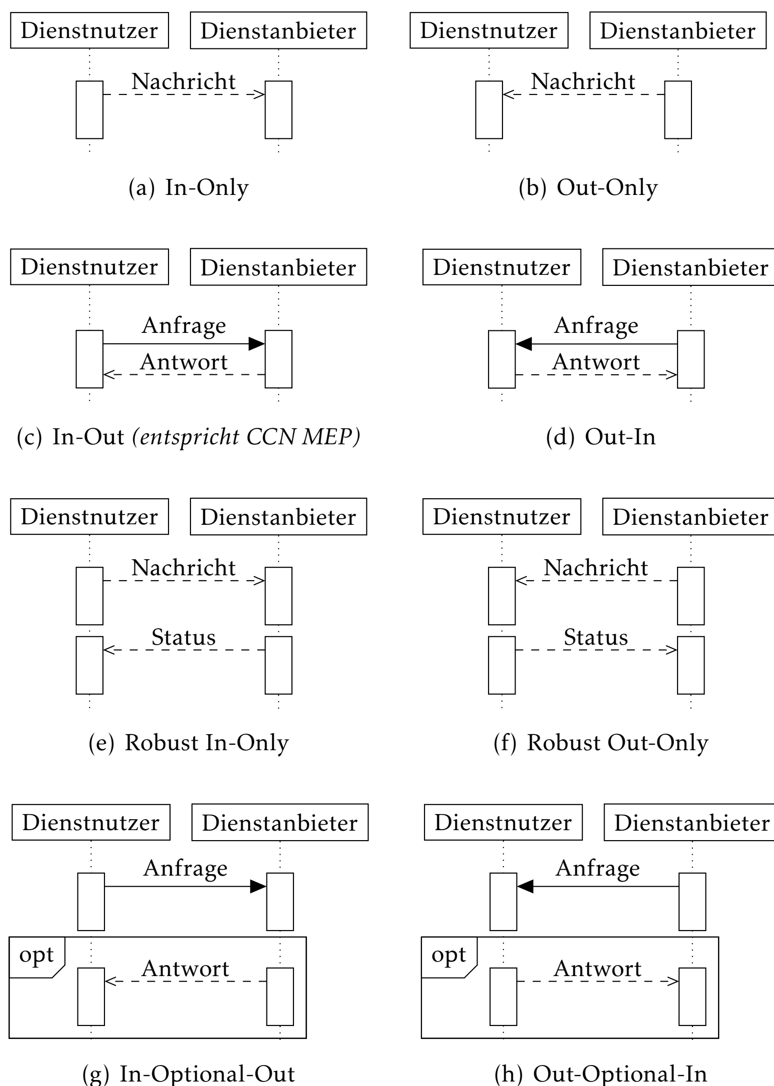


Abbildung 3.3 – *SOAP Messaging Pattern*

Beim „In-Only“ Messaging Pattern in Abbildung 3.3(a) schickt der Dienstnutzer eine Nachricht an den Dienstanbieter. Der Dienstanbieter verarbeitet die Nachricht, schickt aber keine Nachricht als Antwort zurück. Schickt der Dienstanbieter wie in Abbildung 3.3(b) eine Nachricht zum Dienstnutzer, ohne von diesem eine Antwort zu erhalten, handelt es sich um ein „Out-Only“ Messaging Pattern.

Beim „In-Out“ Messaging Pattern in Abbildung 3.3(c) schickt der Dienstanbieter eine Anfrage an den Dienstnutzer. Der Dienstanbieter schickt auf die Anfrage eine Antwort. Die Daten werden in der Antwort transportiert. Damit entspricht das In-Out Messaging Pattern auch dem Messaging Pattern des inhaltszentrischen Ansatzes, wobei die Anfrage der Interest ist, die Antwort das Content Object. Auch beim inhaltszentrischen Ansatz enthält das Content Object die Daten. Beim „Out-In“ Messaging Pattern schickt der Dienstanbieter eine Anfrage und erhält vom Dienstnutzer eine Antwort.

Von den Messaging Pattern „In-Only“ in Abbildung 3.3(a) und „Out-Only“ in Abbildung 3.3(b) gibt es die Varianten „Robust In-Only“ in Abbildung 3.3(e) und „Robust Out-Only“ in Abbildung 3.3(f). Bei den Messaging Pattern mit „Robust“ im Namen bestätigt der Empfänger der Nachricht den Empfang mit einer Statusnachricht.

Weiterhin gibt es von den Messaging Pattern „In-Out“ in 3.3(c) und „Out-In“ in 3.3(d) noch die Varianten „In-Optional-Out“ in Abbildung 3.3(g) und „Out-Optional-In“ in Abbildung 3.3(h). Bei den Messaging Pattern mit „Optional“ im Namen ist die Antwortnachricht optional.

Von den acht Messaging Pattern in Abbildung 3.3 wird vom inhaltszentrischen Ansatz nur das In-Out Pattern in Abbildung 3.3(c) realisiert. Die anderen Messaging Pattern werden im Internet der Dinge aber ebenso benötigt. Beispielsweise benötigt man für ein Publish/Subscribe-Verfahren, wie es beispielsweise von MQTT umgesetzt wird, zwei Messaging Pattern: *(i)* Der Dienstnutzer registriert sich am Dienstanbieter mit dem Ziel, dass der Dienstanbieter im Falle eines Ereignisses eine Nachricht schickt, um den Dienstnutzer über das Ereignis zu informieren. Das entspricht einem *In-Only* Messaging Pattern. *(ii)* Im Falle eines Ereignisses schickt der Dienstanbieter eine Nachricht an den Dienstnutzer. Hierbei handelt es sich um ein *Out-Only* Messaging Pattern.

Da der inhaltszentrische Ansatz nur das In-Out Messaging Pattern (vgl. Abbildung 3.3(c)) unterstützt, ist die Entwicklung von Diensten mit dem inhaltszentrischen Ansatz eine Herausforderung. Insbesondere, dass Daten nur mit einem Interest angefragt werden, stellt ein Problem für die Realisierung von Diensten im Internet der Dinge dar. Geht man davon aus, dass Daten nur in Content Objects verschickt werden, so ist erhöht sich auch das Nachrichtenaufkommen, da Content Objects stets proaktiv mit Interests angefragt werden. Weiterhin ist es nicht möglich, aufgrund von Ereignissen, ein Content Object ohne einen vorangegangenen Interest zu verschicken.

ZUSAMMENFASSUNG HERAUSFORDERUNGEN

Aus den vier Aspekten wurde jeweils eine Herausforderung abgeleitet. Im Folgenden werden in der Arbeit Maßnahmen zum Umgang mit den Herausforderungen vorgeschlagen. Daher werden die Herausforderungen hier zusammengefasst:

1. Der inhaltszentrische Ansatz ist für die Verteilung von Inhalten im Internet konzipiert [86], nicht für die Entwicklung von Diensten im Internet der Dinge. Der *konzeptionelle Unterschied* ist eine Herausforderung bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge.
2. Gegenüber dem vertrauten knotenzentrischen Paradigma ist das *Entwickeln von Diensten* mit dem inhaltszentrischen Paradigma eine Herausforderung (vgl. Aktivitäten in Abbildung 3.2).
3. Lange und sprechende *Namen* des inhaltszentrischen Ansatzes sind für ressourcenbeschränkte Geräte im Internet der Dinge aufgrund der Verarbeitung und ihres Ressourcenanspruchs bzgl. Speicher und Nachrichten eine Herausforderung.

4. Das *Messaging Pattern* des inhaltszentrischen Ansatzes sieht eine starre Abfolge von Nachrichten Anfragen (Interest) und Antwort (Content) vor. Das strikte Messaging Pattern ist eine Herausforderung für die Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge, da im Internet der Dinge verschiedene Messaging Pattern vorgesehen sind [103].

3.2.2 POTENZIALE

Nach den Herausforderungen werden in diesem Unterabschnitt die Potenziale des inhaltszentrischen Ansatzes im Internet der Dinge identifiziert. Die Gliederung orientiert sich an den vier oben genannten Herausforderungen, um den Herausforderungen so die Potenziale gegenüberzustellen. Die letzten beiden Punkte der Gliederung, Inhalte und Messaging Pattern, werden zusammengefasst, da sie nur zusammen ein Potenzial darstellen.

INHALTSZENTRISCHER ANSATZ

Der inhaltszentrische Ansatz ist für die Verteilung von Daten im Netz und nicht für die Entwicklung von Diensten entworfen worden, wie aus der ersten Herausforderung, dem *konzeptionellen Unterschied* zwischen inhaltszentrischen und knotenzentrischen Ansatz hervorgeht. Dienste spielen, wie in Kapitel 1 erwähnt, aber eine bedeutende Rolle im Internet der Dinge. Betrachtet man den inhaltszentrischen Ansatz genauer, so birgt seine Anwendung im Internet der Dinge auch eine Reihe von Potenzialen.

Ein Potenzial ist das flexible Knotenmodell des inhaltszentrischen Ansatzes (vgl. Knotenmodell von CCN in Abbildung 2.7 auf Seite 28). Die Flexibilität zeigt sich beispielsweise daran, dass das Knotenmodell skalierbar ist. Entfernt man zum Beispiel den Content Store auf einigen oder von allen Knoten in einem drahtlosen Netz, so funktioniert die Datenübertragung weiterhin. Eine mögliche Konfiguration in einem inhaltszentrischen, drahtlosen Sensornetz mit heterogenen Geräten wäre, dass Knoten mit mehr Speicher mit einem Content Store ausgestattet sind und Knoten mit wenig Speicher keinen oder einen kleineren Content Store haben.

Ein weiteres Potenzial ist, dass im Gegensatz zu dem knotenzentrischen Ansatz die Daten nicht mehrere Schichten durchlaufen (vgl. Schichtenmodell in Abbildung 2.3 auf Seite 18), bis sie verschickt werden. Beim inhaltszentrischen Ansatz erzeugen Anwendungen Nachrichten und geben diese über Faces an den Dämon weiter, der die Weiterleitung der Nachrichten über weitere Faces übernimmt. Damit ist das gesamte System insgesamt flexibler. Diese Flexibilität wird in Kapitel 7 Basisdienste deutlich, wo Dienste beispielsweise flexibel netzwerkspezifische Aufgaben übernehmen.

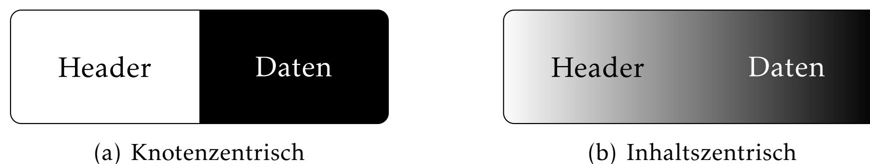
NAMEN

Wie in den Herausforderungen in Abschnitt 3.2.1 dargestellt, stellen insbesondere lange Namen für ressourcenbeschränkte Geräte im Internet der Dinge eine Herausforderung dar. Bei inhaltszentrischen Ansätzen sind Namen prinzipiell frei wählbar. In Netzen mit beschränkten Paketgrößen auf der Sicherungsschicht

empfiehlt es sich daher, kürzere Namen zu verwenden. Die Verwendung kürzerer Namen wird auch von Baccelli et al. [35] vorgeschlagen. Details zur Vergabe von Namen werden in Kapitel 6 diskutiert.

Namen sind aber nicht nur eine Herausforderung. Sie sind auch ein Potenzial für den Einsatz des inhaltszentrischen Ansatzes im Internet der Dinge, denn sie sind flexibler als die IP-Adressen des knotenzentrischen Ansatzes. Bevor konkrete Beispiele gezeigt werden, wie Namen dafür sorgen, dass inhaltszentrische Ansätze flexibler sind, werden zunächst Nachrichtenstrukturen von knotenzentrischen und inhaltszentrischen Ansätzen in Abbildung 3.4 gegenübergestellt. Die Nachrichtenstrukturen sind eine stark vereinfachte Sicht auf die Nachrichten, die in nachrichtenorientierten Netzen, wie dem Internet, ausgetauscht werden. Eine Nachricht besteht nach dieser einfachen Struktur aus einem *Header* und den *Daten*.

Abbildung 3.4(a) zeigt die knotenzentrische Nachrichtenstruktur mit einer strikten Trennung zwischen dem Header und den Daten. Der Header in Paketen bei knotenzentrischen Ansätzen enthält in der Regel ein Adresspaar, bestehend aus einer Quell- und einer Zieladresse. Die Angabe von Ziel- und Quelladresse folgt dem Ende-zu-Ende-Paradigma von knotenzentrischen Ansätzen. Bei knotenzentrischen Ansätzen gilt, dass Adressen nur Knoten (oder Netzwerkschnittstellen) adressieren. Diese Adressen sind nur für die Auslieferung der Pakete relevant und haben für die Anwendung keine Bedeutung. Dies gilt auch für die Adressen der Vermittlungsschicht beziehungsweise der IP-Adressen der Internetschicht (vgl. Referenzmodelle Abbildung 2.1), auf die sich die folgenden Vergleiche beziehen. Die IP-Adressen der Internetschicht werden deswegen als Vergleich herangezogen, da ihr Gültigkeitsbereich ähnlich dem der Namen im inhaltszentrischen Internet ist. Tatsächlich ist der Gültigkeitsbereich der Namen größer, da zum Beispiel im Namen auch Anwendungen adressiert werden. Beim knotenzentrischen Ansatz werden Anwendungen über die Port-Nummern der Transportschicht adressiert.



(a) Knotenzentrisch (b) Inhaltszentrisch

ABBILDUNG 3.4 – *Nachrichtenstrukturen*

Bei der inhaltszentrischen Nachrichtenstruktur in Abbildung 3.4(b) ist diese Trennung zwischen Adresse und Daten *unscharf*, da der Name im Paket Informationen zur Adressierung als auch Informationen, im Sinne von Daten, transportiert. Der Name wird bei inhaltszentrischen Ansätzen von der Anwendung gesetzt, wie aus dem Teil-Aktivitätsdiagramm in Abbildung 3.2(a) auf Seite 40 (Aktivität 3), der Entwicklung von Diensten mit dem inhaltszentrischen Ansatz, hervorgeht. Daher ist es für die Anwendung auch möglich, Informationen in den Namen zu kodieren.

Ein Beispiel für Informationen oder Daten, die im Namen transportiert werden, sind *Kommandos*. Kommandos sind eine Namenskonvention für Komponenten und wurden mit den ersten Versionen von CCNx eingeführt. In CCNx 1.0 sind Kommandos durch sogenannte Segment Marker ersetzt worden. Die hier vorgeschlagene Implementierung für namenszentrische Dienste nutzt eine zu den Kommandos ähnliche Konvention für die *Dienstmethoden*.

Da die Darstellung von Dienstmethoden in dieser Implementierung für die Anwendung auf ressourcenbeschränkten Geräten entwickelt wurde und aufgrund der daraus resultierenden kompakten Darstellung nicht sehr anschaulich ist, wird in Abbildung 3.5 ein anschauliches Beispiel für ein CCNx-Kommando gezeigt. Für ressourcenbeschränkte Geräte im Internet der Dinge ist die Methode ein Ersatz für das Kommando. Ein Kommando beginnt mit einem Command Marker, dann folgen der Namensraum der Operation, die Operation sowie die Parameter der Operation.

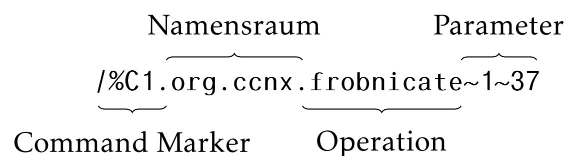


Abbildung 3.5 – *Beispiel für ein CCNx-Kommando*

Gemäß der Nachrichtenstruktur aus Abbildung 3.4(b), entspricht der Präfix einer Nachricht dem Header mit den Adressinformationen, der Suffix entspricht den Daten. Wie oben erwähnt bezieht sich der Vergleich auf die Vermittlungsschicht. Abbildung 3.6 zeigt die Struktur eines beispielhaften Interests mit einem Namen. Der Präfix `/de/fh-luebeck` entspricht dem „Header-Anteil" und der Suffix `/%C1.org.ccnx.frobnicate~1~37` dem „Datenanteil".

`/de/fh-luebeck/foo/%C1.org.ccnx.frobnicate~1~37`

Abbildung 3.6 – *Präfix (Header) und Suffix (Daten) einer inhaltszentrischen Nachricht*

Dabei ist der Übergang zwischen Header beziehungsweise Adressen und Daten fließend und flexibel durch die Anwendung festgelegt, wie in Abbildung 3.6 dargestellt. Eine Komponente wie in Abbildung 3.5 bildet in der Regel jedoch den Suffix eines Namens, da sonst FIB-Einträge für alle möglichen Kombinationen von Parametern nötig wären, um den längsten gemeinsamen Präfix zu ermitteln.

Namen sind beliebig wählbar und so ist es auch möglich, den Präfix mit einer Bedeutung oder Semantik zu versehen, um so kontextabhängige Adressierungen zu realisieren. Der Präfix enthält dann beispielsweise Lokationsinformationen, die inhaltszentrisch vernetzte Roboter adressieren. Eine Lokationsinformation repräsentiert dabei die physikalische Position des Roboters. Diese Art der Adressierung ist nützlich, um Roboter an einem bestimmten Ort zu adressieren.

Bei Reinigungsrobotern ist es beispielsweise unerheblich, welcher individuelle Roboter an einem bestimmten Ort eine Reinigung durchführt, sondern dass ein Roboter vor Ort adressierbar ist.

Bei knotenzentrischen Protokollen wird eine kontextabhängige Adressierung auf Anwendungsebene implementiert. Beim obigen Beispiel mit den Robotern kommuniziert die Anwendung mittels Multicast-Adressen auf der Netzwerkschicht. Die Kontextinformationen würden bei einer knotenzentrischen Anwendung in den Daten transportiert werden. Problematisch bei einer kontextabhängigen Adressierung bei einem knotenzentrischen Ansatz ist, dass ein Routing auf der Netzwerkschicht nicht möglich ist, da die benötigte Kontextinformation nur auf der Anwendungsschicht vorhanden ist.

Eine weitere Strategie, um eine kontextabhängige Adressierung in knotenzentrischen Netzen zu realisieren, ist das Locator/Identifier Separation Protocol (LISP) [72, 142]. Der Haupteinsatzzweck von LISP ist es, die Mobilität von IP-adressierten Knoten zu ermöglichen, indem die Identität von dem Ort des Knotens (Lokation) getrennt wird. Bewegt sich beispielsweise ein Knoten aus einem IP-Netz in ein anderes, so ändert sich in der Regel auch seine IP-Adresse. Die IP-Adresse identifiziert den Knoten selbst und liefert auch seine Lokation in einem Netzwerk. Bei LISP haben Knoten eine feste IP-Adresse unter der sie erreichbar sind. In den Netzen, die LISP unterstützen, gibt es spezielle Router, die die Kommunikation zwischen LISP Netzen tunneln.

Okada et al. [121] weisen geografischen Positionen IP-Multicast-Adressen zu. Diese Multicast-Adressen werden wiederum LISP Routern zugewiesen. Verallgemeinert man das Verfahren von Okada et al., lassen sich beliebige kontextabhängige Adressierungen in IP-basierten Netzen implementieren.

Die im Rahmen dieser Arbeit entwickelten namenszentrischen Dienste realisieren die Konzepte *Identität* und *Lokation* getrennt voneinander über Dienste direkt auf den Knoten. Im Gegensatz zu IP und LISP gibt es bei den namenszentrischen Diensten keine Infrastruktur, die diese Trennung mittels zustandsbehafteten Netzwerkkomponenten vornimmt. Das heißt nicht, dass es keine Infrastruktur bei den namenszentrischen Diensten gibt. Gateways, die in Kapitel 4 vorgestellt werden, sind eine Infrastrukturkomponente, die zwischen drahtgebundenen zu drahtlosen Netzen vermittelt.

Zusammengefasst sind Namen flexibler einsetzbar als die IP-Adressen im knotenzentrischen Internet. Mit IP werden nur die Knoten adressiert, daher ist eine kontextabhängige Adressierung in knotenzentrischen Netzen nur recht umständlich mit speziellen Infrastrukturkomponenten (z. B. Routern) umzusetzen, wie die Beispiele von LISP und die Adressierung von geografischen Positionen von Okada et al. zeigen. Mit Namen ist es einfach möglich, geografische Koordinaten zu kodieren und somit Knoten oder Dienste an bestimmten Orten zu adressieren.

INHALTE UND MESSAGING PATTERN

Die identifizierten Herausforderungen wie Kurzlebigkeit von Inhalten und dem Messaging Pattern des inhaltszentrischen Ansatzes sind in der Umsetzung zu be-

trachten. Ein großer Teil des Netzwerkverkehrs in drahtlosen Sensornetzen wird das Senden von Messwerten zu einer Datensenke sein, wie in RFC 6550 [173] angenommen. Neue Funktionalitäten in Sensornetzen, beispielsweise neue Dienste, werden durch Softwareupdates ergänzt. Hier ist das Caching des inhaltszentrischen Ansatzes nützlich, da Softwareupdates direkt von Knoten in der Nachbarschaft bezogen werden, die das Update bereits erhalten haben. Bei dem Ausbringen von Updates hat der inhaltszentrische Ansatz mit dem Caching und dem Messaging Pattern Vorteile in einem inhaltszentrischen Internet der Dinge.

Bei dem knotenzentrischen Ansatz hingegen baut jeder Knoten eine Ende-zu-Ende-Verbindung zu dem Host auf, der die Daten für das Update bereitstellt. Hier wird der Vorteil von Caching deutlich und daher implementieren auch knotenzentrische Ansätze Methoden zum Caching von Inhalten. Im Kontext der drahtlosen Sensornetze implementiert CoAP [151] als knotenzentrisches Anwendungsprotokoll einen Caching-Mechanismus. Daher sei an dieser Stelle darauf hingewiesen, dass Caching kein Alleinstellungsmerkmal von inhaltszentrischen Ansätzen ist.

Auch kurzlebige Sensorwerte profitieren vom Caching, wie Baccelli et al. [35] bemerken. Dabei werden Sensorwerte nach dem für inhaltszentrische Ansätze üblichen Messaging Pattern ausgetauscht, also mit einem Interest wird der Sensorwert in einem Content Object angefordert. Für das inhaltszentrische Messaging Pattern aus Anfrage und Antwort wurden Anwendungsbeispiele in der Gebäudeautomation identifiziert (siehe hierzu RFC 5867 „Building Automation Routing Requirements in Low-Power and Lossy Networks“ [103]). Als Vorteil des Messaging Pattern aus Anfrage und Antwort wird eine einheitliche und konstante Paketbelastung im Netzwerk genannt.

Wie oben dargestellt sind Interests in der Lage, in ihrem Namen allgemeine Informationen, wie Sensorwerte, zu kodieren. Die Verwendung von Interests, um damit Sensorwerte zu übertragen, wirkt auf den ersten Blick zweckentfremdet. Um diese Zweckentfremdung aufzulösen, bietet sich eine Interpretation des Interests dahingehend an, dass es sich dabei um eine Anfrage an potenzielle Interessenten von Sensorwerten handelt. Wenn keine Antwort auf die Interests erwartet wird, wird die Gültigkeit der Interests entsprechend niedrig gewählt. Mit Interests und der Variation ihrer Gültigkeit ist es möglich, beliebige Messaging Pattern mit dem inhaltszentrischen Ansatz im Internet der Dinge zu realisieren.

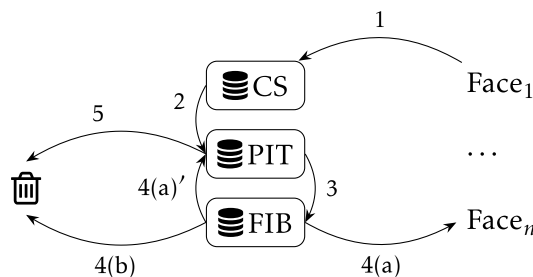


ABBILDUNG 3.7 – *Verarbeitung von kurzlebigen Interests*

Kurzlebige Interests, also Interests mit einer kurzen Gültigkeit, werden genauso verarbeitet wie Interests mit einer langen Gültigkeit (vgl. Grundlagen der Nachrichtenverarbeitung beim Inhaltszentrischen Ansatz in Abschnitt 2.4.4). Die Verarbeitungsschritte von kurzlebigen Interests sind in Abbildung 3.7 dargestellt. In *Schritt 1* wird der Content Store geprüft, ob es ein passendes Content Object gibt. Nun ist es so, dass bei kurzlebigen Interests in der Regel kein Content Object erwartet wird, daher ist es unwahrscheinlich im Content Store ein Content Object zu haben, welches zu dem Interest passt. In *Schritt 2* wird die Pending Interest Table geprüft, ob dort ein Interest mit demselben Namen vorhanden ist. Auch das ist eher unwahrscheinlich, da davon ausgegangen wird, dass Interests mit dem gleichen Namen auch kurzlebig sind und bereits wieder aus der Pending Interest Table entfernt wurden. In *Schritt 3* wird der Interest gegen die Einträge in der Forwarding Information Base geprüft und entweder in *Schritt 4(a)* zu einem Face weitergeleitet. Wenn kein passender Eintrag gefunden wurde, wird der Interest in *Schritt 4(b)* verworfen. Weiterhin wird der Interest in *Schritt 4(a)′* in die Pending Interest Table eingetragen. Da der Interest allerdings kurzlebig ist, wird der Interest in *Schritt 5* nach einer kurzen Zeit wieder aus der Pending Interest Table entfernt.

3.3 KONZEPT

Dieser Abschnitt führt das Konzept der namenszentrischen Dienste gemäß Definition 2.2 ein. Zuerst wird das allgemeine Konzept von namenszentrischen Diensten als Software vorgestellt. Es folgt die Schnittstelle der namenszentrischen Dienste und der Dienstbeschreibung.

Der Begriff *namenszentrisch* wird in dieser Arbeit gewählt, da die Adressierung von Diensten mit Namen im Vordergrund steht. In ähnlicher Form findet sich diese Bezeichnung auch in der Literatur, beispielsweise bei den Veröffentlichungen „Networking Named Data“ [86] oder „Named Data Networking (NDN) Project“ [178].

3.3.1 DIENSTE

Dienste sind nach Definition 2.2 auf Seite 24 Software, die eine definierte Funktionalität über das Netz bereitstellen. Abbildung 3.8 zeigt einen Dienstanbieter (vgl. Definition 2.5 auf Seite 25) und einen Dienstnutzer (vgl. Definition 2.6), die über das Netz kommunizieren. Dienstanbieter und Dienstnutzer, dargestellt als Kreise, sind mit dem Netz, dargestellt als Wolke, verbunden. Anfragen und Antworten zwischen Dienstanbieter und Dienstnutzer, dargestellt als gestrichelter Doppelpfeil, werden über das Netz ausgetauscht.

Ersetzt man das Netz in Abbildung 3.8 durch einen CCN-Dämon (vgl. Abbildung 2.7 auf Seite 28) und stellt sich die Dienstanbieter und Dienstnutzer als CCN-Anwendungen (Software) auf einem Knoten vor, erhält man das *Knotenmodell für namenszentrische Dienste* in Abbildung 3.9. CCN-Anwendungen, die Dienstanbieter und Dienstnutzer implementieren, werden im Knotenmodell für namenszentrische Dienste als *Dienstinstanz* (engl. Service Instance, $S_1, \ldots, S_n$)

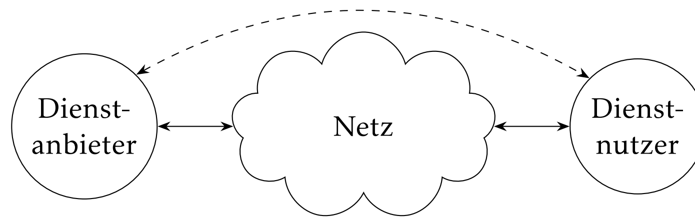


ABBILDUNG 3.8 – *Abstraktes Konzept von Dienstanbieter und Dienstnutzer*

bezeichnet. Die Verbindungslinien mit den Pfeilspitzen an beiden Enden zwischen dem Dämon und den Dienstinstanzen in Abbildung 3.9 sind die Faces ($F_1, \ldots, F_n$). Dienstinstanzen erzeugen Nachrichten und schicken sie über die Faces an den Dämon. Nachrichten sind Interests und Content Objects. Ein Interest, der einen entfernten Methodenaufruf enthält, wird in dieser Arbeit als *Anfrage* bezeichnet. Ein Content Object, was den Rückgabewert einer Anfrage des entfernten Methodenaufrufs enthält, wird als *Antwort* bezeichnet.

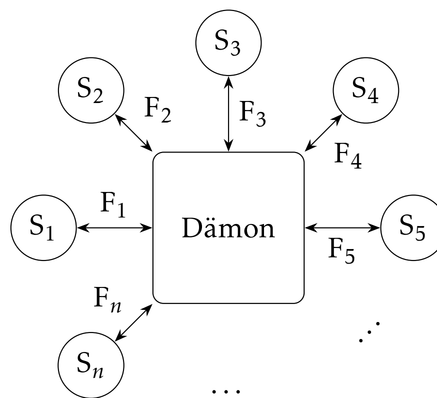


ABBILDUNG 3.9 – *Knotenmodell für namenszentrische Dienste*

Das Knotenmodell Abbildung 3.9 unterschiedet sich in der Darstellung des CCN-Knotenmodells dahingehend, dass die Datenstrukturen des Dämons nicht explizit dargestellt sind. Diese Datenstrukturen im Dämon gibt es nach wie vor und die Nachrichten werden wie in Abschnitt 2.4.4 beschrieben verarbeitet. Die Dienstinstanzen kommunizieren über sogenannte entfernte Methodenaufrufe miteinander. Durch die entfernten Methodenaufrufe sind die Dienste lose aneinander gekoppelt. Die Kopplung ist deshalb lose, da Dienstanbieter und Dienstnutzer transparent austauschbar sind. Um diese lose Kopplung zu illustrieren, werden die Dienstinstanzen im Knotenmodell der namenszentrischen Dienste um den Dämon herum dargestellt.

Die lose Kopplung macht namenszentrische Dienste sehr flexibel einsetzbar. Diese Flexibilität zeigt sich daran, dass Dienstnutzer über Knotengrenzen hinweg mit Dienstanbietern transparent durch andere Dienstanbieter kommunizieren, die als Funktionalität den Transport von Nachrichten über die Knotengrenzen hinweg anbieten, wie in Abbildung 3.10 dargestellt.

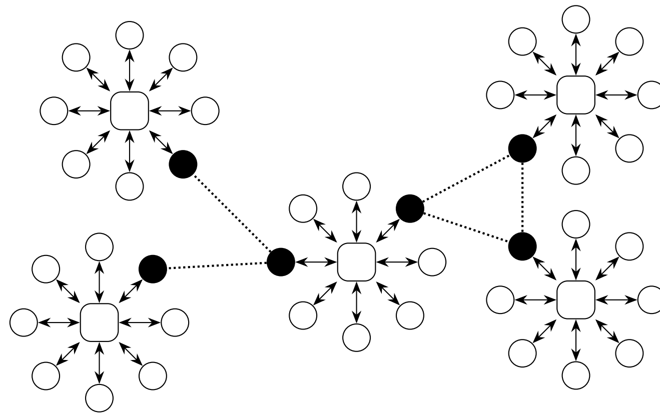


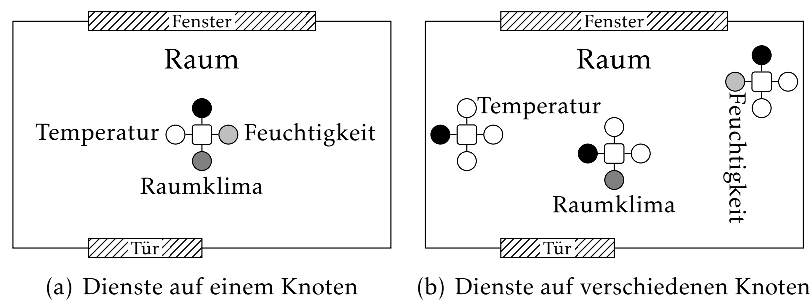
ABBILDUNG 3.10 – *Netz aus namenszentrischen Diensten über mehrere Knoten verteilt*

Dienste, die Nachrichten über Knotengrenzen transportieren sind in Abbildung 3.10 farblich hervorgehoben. Sie haben Zugriff auf die Kommunikationshardware des Knotens, wie beispielsweise die Funkschnittstelle oder die serielle Schnittstelle. Abbildung 3.10 zeigt darüber hinaus einen wichtigen Aspekt der namenszentrischen Dienste: Dienstanbieter und Dienstnutzer kommunizieren miteinander und nicht die Knoten. Im Gegensatz zu CCNx gibt es in dem Knotenmodell für namenszentrische Dienste keine dedizierten Netzwerk-Faces (vgl. Abbildung 2.7 auf Seite 28).

Dienste, die Nachrichten zu anderen Knoten weiterleiten und die lose Kopplung der Dienste erlauben, dass nicht jeder Knoten jeden Dienst anbieten muss und so Funktionalität im Netz verteilt werden kann. In einem Sensornetz gibt es beispielsweise einen Dienstanbieter, der als Zeitgeber periodisch bei den Dienstnutzern eine Aktion auslöst, wie das Einlesen von Sensorwerten. Diese Anwendung wurde in „Name-Centric Service Architecture for Cyber-Physical Systems (Short Paper)“ von Hellbrück et al. [9] vorgestellt. Entweder ist jeder Knoten mit diesem Zeitgeber ausgerüstet, oder ein Knoten in der Nachbarschaft bietet diesen Zeitgeberdienst an. Für den Zeitgeber Dienstanbieter und für die Dienstnutzer ist es transparent, ob sie sich auf demselben oder auf unterschiedlichen Knoten befinden.

Lose gekoppelte Dienste sind in vielen Anwendungsszenarien nützlich, beispielsweise in der Gebäudeautomation. Abbildung 3.11 zeigt ein Szenario in zwei alternativen Hardwareausstattungen aus der Gebäudeautomation. In dem Szenario bieten Dienste für Temperatur, Feuchtigkeit und Lüftungsregelung den höherwertigen Dienst der Regelung des Raumklimas an.

In dem Beispiel in Abbildung 3.11(a) gibt es einen Sensorknoten, der folgende Dienste ausführt: *(i)* Temperaturmessung, *(ii)* Feuchtigkeitsmessung, *(iii)* Regelung für das Raumklima (z. B. Öffnen und Schließen des Fensters) und *(iv)* Kommunikation. Der Dienst für die Regelung des Raumklimas ist Dienstnutzer der anderen Dienste, die Temperatur und Feuchtigkeit bereitstellen. Das Öffnen und Schließen des Fensters wird in Abhängigkeit der gemessenen Temperatur und Luftfeuchtigkeit reguliert.



(a) Dienste auf einem Knoten (b) Dienste auf verschiedenen Knoten

ABBILDUNG 3.11 – *Beispiel für lose gekoppelte namenszentrische Dienste in der Gebäudeautomation*

In Abbildung 3.11(b) werden die gleichen Dienste für Temperatur- und Feuchtigkeitsmessung sowie der Regelung für das Raumklima ausgeführt, nur dass sie anstatt auf einem Knoten auf drei unterschiedlichen Knoten ausgeführt werden. Jeder Knoten hat einen Dienst für die Kommunikation.

Für die Dienstanbieter und Dienstnutzer ist es dabei transparent, ob sie auf einem oder auf unterschiedlichen Knoten ausgeführt werden. Transparent bedeutet in diesem Fall, dass eine Änderung der Konfiguration an den einzelnen Diensten nicht notwendig ist, denn die Dienste sind untereinander immer noch durch ihren Namen erreichbar, egal auf welchem Knoten sie ausgeführt werden. Wenn der Dienstnutzer, die Regelung des Raumklimas, eine Anfrage mit dem Namen `/temp` schickt, wird dieser vom Dämon auf dem Knoten verarbeitet. Im ersten Fall in Abbildung 3.11(a) wird die Anfrage mit dem Namen `/temp` bei der Verarbeitung im Dämon an den Dienst zur Temperaturmessung weitergeleitet. Das liegt daran, dass es in der Forwarding Information Base einen Eintrag mit dem Präfix `/temp` gibt, der Anfragen zu dem Face weiterleitet, hinter dem sich die Dienstinstanz mit dem Dienst für die Temperaturmessung befindet.

Im zweiten Fall in Abbildung 3.11(b) gibt es auf dem Knoten kein Präfix `/temp` in der Forwarding Information Base. Die Anfrage wird daher zum Kommunikationsdienst weitergeleitet und über die Funkschnittstelle versendet, da dieser mit dem Root-Namen in der Forwarding Information Base registriert ist und der Root-Name jeden Präfix matcht, weil er in jedem Namen enthalten ist. Wenn die Anfrage am Knoten empfangen wird, der die Dienst für die Temperaturmessung ausführt, wird die Anfrage dort zum Dienst für Temperaturmessung weitergeleitet.

In beiden Fällen wird die Antwort wieder an den Dienstnutzer zurückgeschickt, da über die Einträge in der Pending Interest Table der Rückweg für die Antwort gegeben ist. Kommunikationsdienste gehören zu den Basisdiensten, die in Kapitel 7 eingeführt und erklärt werden.

Namenszentrische Dienste sehen einen modularen Aufbau vor und lassen sich damit bedarfsgerecht an verschieden Szenarien anpassen. Damit unterscheiden sich namenszentrische Dienste von den knotenzentrischen Ansätzen, da es dort Bestrebungen gibt, wonach 6LoWPAN, CoAP und RPL feste Bestandteile des Netzwerkstacks sein sollen, wie Clausen et al. [57] schreiben. Das heißt, dass

RPL auch dann eingesetzt wird, wenn es eigentlich nicht nötig wäre. Durch den unnötigen Einsatz werden unter Umständen mehr Ressourcen, wie Speicher und Energie, als nötig verbraucht.

3.3.2 DIENSTMETHODE

Das Ergebnis einer Anforderungsanalyse, die dieser Arbeit vorausging, war die Umsetzung der namenszentrischen Dienste mit entfernten Methodenaufrufen wie bei SOAP (vgl. Verwandte Arbeiten in Abschnitt 3.1). Methoden sind wie in Abbildung 3.6 auf Seite 47 sind im Namen einer Anfrage kodiert. Der Begriff *Dienstmethode* wird im Folgenden für den entfernten Methodenaufruf verwendet. In Abgrenzung dazu wird der Begriff Methode für Methoden im Sinne von Programmiersprachen verwendet.

Eine Dienstmethode wird am Dienstnutzer mit einer Methode in der jeweiligen Programmiersprache aufgerufen. Das heißt, dass der Aufruf der Methode eine Nachricht erzeugt, in der ein (entfernter) Methodenaufruf kodiert ist. Ein Methodenaufruf folgt einer Syntax, die im wesentlichen für alle Methoden und in allen Programmiersprachen ähnlich ist. Die Syntax eines Methodenaufrufs in Pseudocode ist in Abbildung 3.12 dargestellt. Sie zeigt die Elemente, die jeder Methodenaufruf enthält. Jedes Element findet sich auch in einer Dienstmethode und wird entsprechend in der Nachricht kodiert.

$$\underbrace{\texttt{retType}\ \underbrace{\texttt{methodName}}_{\text{Methodenname}}\underbrace{\texttt{(paramType}_1\ \texttt{param}_1\texttt{, paramType}_2\ \texttt{param}_2\texttt{, ...)}}_{\text{Parameter}}}_{\text{Signatur}}$$

ABBILDUNG 3.12 – *Aufrufsyntax einer Dienstmethode*

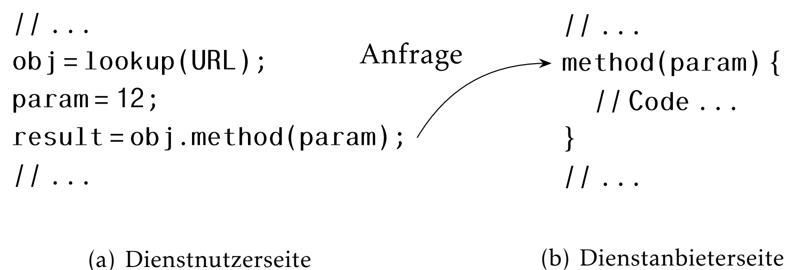
Methoden haben einen Rückgabedatentyp (`returnType`), einen Namen (`methodName`) und eine Parameterliste ($\texttt{param}_n$) mit ihren Datentypen ($\texttt{paramType}_n$). Die Größe von Nachrichten ist bei ressourcenbeschränkten Geräten im Internet der Dinge ein entscheidender Faktor. Aus diesem Grunde wird bei der Parameterliste bei Methoden der namenszentrischen Dienste auf optionale Parameter oder das Überladen von Methoden, bekannt aus objektorientierten Programmiersprachen, verzichtet. Der Verzicht auf optionale Parameter und das Überladen von Methoden verhindert Mehrdeutigkeiten, denn zur Auflösung der Mehrdeutigkeiten würden zusätzliche Informationen gebraucht werden und das Resultat wären dann größere Nachrichten.

Rückgabedatentyp, Methodenname und die formale Parameterliste bilden zusammen die sogenannte *Signatur*. Die Signatur beschreibt die Schnittstelle einer Methode vollständig. Der Begriff Datentyp (engl. Data Type) wird in dieser Arbeit analog zu dem gleichlautenden Begriff aus den Programmiersprachen verwendet [147].

Der Rückgabetyp beschreibt den Datentyp des *Rückgabewertes* und der Rückgabewert wird in der Antwort auf den Dienstmethodenaufruf verschickt. Der

Rückgabetyp ist bei Dienstmethoden optional. Ein fehlender Rückgabetyp ist vergleichbar mit dem Rückgabetyp `void` in der Programmiersprache C [25]. Wenn Dienstmethoden keinen Rückgabetyp haben, dann hat ihr Aufruf in der Regel einen *Seiteneffekt* [147]. Bei einem Seiteneffekt wird beispielsweise der Zustand des Dienstanbieters, der die Anfrage empfängt, verändert. Ein Beispiel für einen Dienstmethodenaufruf mit einem Seiteneffekt, wäre das Einschalten einer Leuchtdiode an einem Knoten. Hier ändert sich der Zustand von „Leuchtdiode aus" zu „Leuchtdiode an".

Namenszentrische Dienstmethodenaufrufe unterscheiden sich kaum von den knotenzentrischen Dienstaufrufen. Der Dienstmethodenaufruf in einem knotenzentrischen Netz aus der Sicht des Entwicklers ist als Pseudocode in Abbildung 3.13 dargestellt. Ein Dienst in einem knotenzentrischen Netz wird in Abgrenzung zu dem namenszentrischen Dienst im Folgenden als *knotenzentrischer Dienst* bezeichnet.



(a) Dienstnutzerseite (b) Dienstanbieterseite

ABBILDUNG 3.13 – *Entfernter Methodenaufruf aus Entwicklersicht (knotenzentrisch)*

Auf der Dienstnutzerseite in Abbildung 3.13(a) wird zuerst ein Objekt für den sogenannten Endpunkt erzeugt, das von der `lookup()` Methode zurückgegeben wird. Der Methode `lookup()` wird als Parameter die URL des Endpunkts übergeben. Der Aufruf der `lookup()`-Methode wird auch als *Bindung* an den Endpunkt bezeichnet. Ein Teil der URL ist entweder eine IP-Adresse oder ein Domainname. Ein Domainname wird durch das Domain Name System wieder auf eine IP-Adresse abgebildet. Der Aufruf der Dienstmethode `method()` erzeugt eine Nachricht, die an den Dienstanbieter adressiert ist, der im Parameter von `lookup()` spezifiziert wurde. Die Dienstmethode hat einen Parameter, der den Wert 12 hat.

Wird die Nachricht mit dem Methodenaufruf am Dienstanbieter empfangen, wird die Implementierung der Dienstmethode in Abbildung 3.13(b) aufgerufen. Der Rückgabewert der Dienstmethode wird mit einer Nachricht an den Dienstnutzer geschickt. Die Methode `method()` blockiert auf der Dienstnutzerseite solange, bis der Rückgabewert empfangen wurde (oder ein Fehler auftritt). Der auf der Dienstnutzerseite empfangene Rückgabewert wird dann von `method()` zurückgegeben und der Variablen `result` zugewiesen.

Bei namenszentrischen Diensten in Abbildung 3.14 erfolgt die Bindung nicht an einen Endpunkt, der als Objekt repräsentiert wird, sondern ein Präfix wird an einen Dienstmethodenaufruf gebunden. Dazu wird der Dienstmethode ein

Präfix sowie die Parameter übergeben, wie in Abbildung 3.13(a) dargestellt. Der Aufruf der Dienstmethode `method()` auf der Dienstnutzerseite erzeugt eine Anfrage, wobei sich der Name aus dem Präfix `/pre/fix` sowie dem kodierten Methodenaufruf zusammensetzt. Das Zusammensetzen des Namens geschieht auf der Dienstnutzerseite in der Dienstmethode. Das setzten des Präfix beim Methodenaufruf ist notwendig, da man bei namenszentrischen Diensten nicht mit einem bestimmten Endpunkt kommuniziert, wie es bei knotenzentrischen Diensten der Fall ist. Hätten namenszentrische Dienste keinen Einfluss auf den Präfix beim Dienstmethodenaufruf, wären Dienste, die eine kontextabhängige Adressierung implementieren, nicht möglich.

```
// ...
prefix="/pre/fix";
param=12;
result=method(prefix,param);
// ...
```

(a) Dienstnutzerseite

Interest →

```
// ...
method(prefix,param){
    //Code ...
}
// ...
```

(b) Dienstanbieterseite

Abbildung 3.14 – *Entfernter Methodenaufruf aus Entwicklersicht (namenszentrisch)*

Beim Empfang der Anfrage wird auf der Dienstanbieterseite die Dienstmethode in Abbildung 3.14(b) aufgerufen. Der Dienstmethode auf der Dienstanbieterseite wird zusätzlich zu den Parametern der Präfix übergeben. Der Präfix ist für Dienste notwendig, die beispielsweise eine kontextabhängige Adressierung implementieren. Braucht der Entwickler der Dienstmethode den Präfix nicht, so wird der Präfix einfach ignoriert.

PROXIES

Das Senden eines Interests auf der Dienstnutzerseite beim Dienstmethodenaufruf und der Aufruf einer Methode auf der Dienstanbieterseite wie in Abbildung 3.14 wird mit sogenannten *Proxies* realisiert. Proxies sind Module oder Klassen, die stellvertretend für die entfernte Schnittstelle stehen. Auf der Dienstanbieterseite gibt es den *Dienstanbieter-Proxy* und auf der Dienstnutzerseite den *Dienstnutzer-Proxy*.

Dienstanbieter- und Dienstnutzer-Proxy sind ein bekanntes Konzept aus den verteilten Systemen und werden bei RPC und SOAP [143] eingesetzt. Das Sequenzdiagramm in Abbildung 3.15 zeigt die Methodenaufrufe und den Nachrichtenaustausch zwischen den Proxies auf Dienstnutzer- und Dienstanbieterseite.

In Abbildung 3.15 nutzt der Dienstnutzer eine entfernte Methode (`method()`). Der Dienstnutzer ruft dazu eine Methode am Dienstnutzer-Proxy auf. Die Methode am Dienstnutzer-Proxy hat bis auf das Informationsobjekt für den Interest (vgl. Abbildung 3.14) die gleiche Signatur wie die korrespondierende Methode auf der Dienstanbieterseite. Beim Methodenaufruf erzeugt der Dienstnutzer-

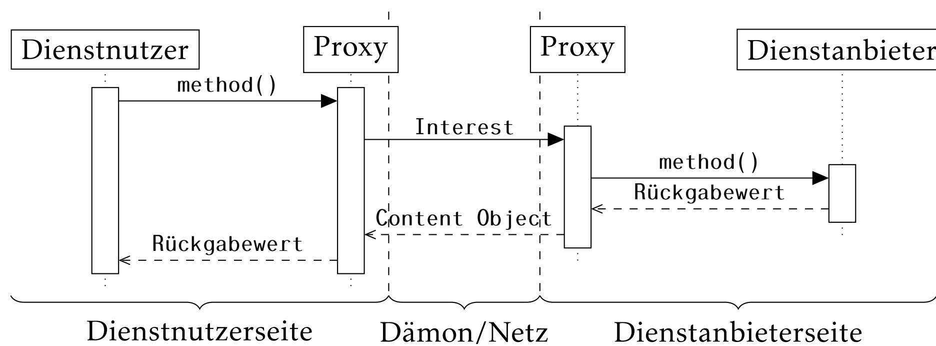


Abbildung 3.15 – *Methodenaufrufe und Nachrichtenaustausch zwischen Dienstanbieter, Dienstnutzer und Proxies*

Proxy eine entsprechende Anfrage die den Aufruf der Dienstmethode enthält. Der Dienstnutzer-Proxy verschickt den Interest mit CCN.

Auf der Seite des Dienstanbieters (links) wird der Interest vom Dienstanbieter-Proxy empfangen. Der Dienstanbieter-Proxy entnimmt dem Interest die Information über die aufzurufende Dienstmethode. Dann ruft der Proxy die entsprechende Methode des Dienstanbieters auf, woraufhin der Dienstanbieter die Methode ausführt und den Rückgabewert beziehungsweise die Antwort ermittelt. Hat die Methode des Dienstanbieters die Antwort ermittelt, erzeugt der Dienstanbieter-Proxy ein Content Object mit der Antwort.

SEGMENTIERUNG

Eine weitere Aufgabe der Proxies ist die transparente *Segmentierung* der Antwort, wenn diese aufgrund ihrer Größe nicht in ein Content Object passt, wie in Abbildung 3.16 dargestellt. Durch den Dienstanbieter-Proxy wird zuerst ein Teil der Antwort in einem Content Object verschickt. Abbildung 3.16 zeigt die Namen der Interests und der Content Objects an den Pfeilen, die den Nachrichtenaustausch zwischen den Proxies darstellen. Dass es sich nur um ein Segment der Nachricht handelt, wird durch den Suffix `/s1-2` angegeben. Der Suffix identifiziert das erste Segment von insgesamt zwei Segmenten.

Auf der Seite des Dienstnutzers fordert der Dienstnutzer-Proxy nach dem Empfang des ersten Segments das zweite Segment durch eine erneute Anfrage an. Der Dienstanbieter-Proxy erzeugt beim Empfang der Anfrage ein Content Object, das das zweite Segment der Antwort enthält. Der Dienstnutzer-Proxy fügt die Segmente wieder zusammen und übergibt die Antwort an den Dienstnutzer.

In Abhängigkeit der Implementierung der namenszentrischen Dienste, ist der Aufruf der Dienstmethode des Dienstnutzer-Proxies entweder *synchron* oder *asynchron* [54]. Der Unterschied zwischen einem synchronen und einem asynchronen Aufruf einer Dienstmethode ist in Abbildung 3.17 dargestellt.

Bei einem synchronen Aufruf in Abbildung 3.17(a) der Dienstmethode blockiert die Methode des Dienstanbieter-Proxies solange, bis die Antwort vollständig empfangen wurde. Ein synchroner Aufruf wird durch eine gefüllte Pfeilspitze

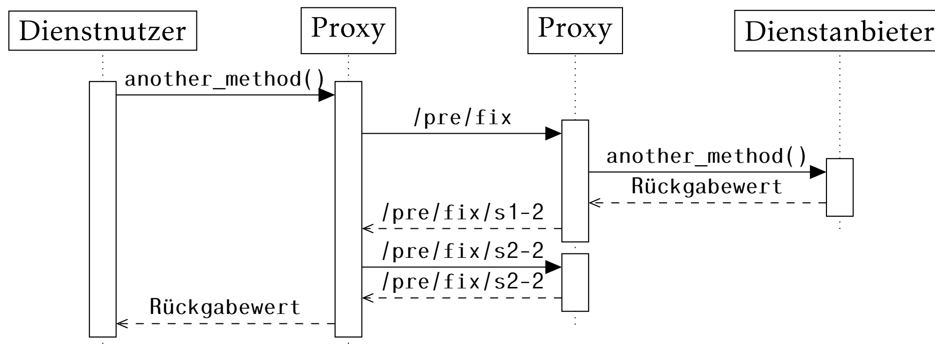
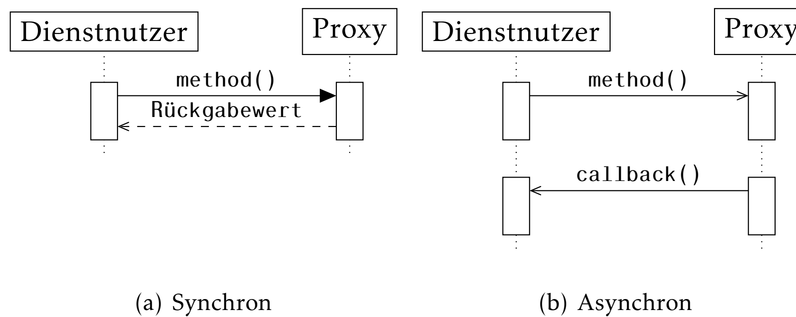


Abbildung 3.16 – *Segmentierung von großen Rückgabewerten durch die Proxies*



(a) Synchron (b) Asynchron

Abbildung 3.17 – *Aufruf von Dienstmethoden am Dienstnutzer*

dargestellt. Die Antwort ist bei einem synchronen Aufruf der Methode des Dienstanbieter-Proxies auch der Rückgabewert der Methode selbst.

Bei einem asynchronen Aufruf in Abbildung 3.17(b) kehrt die Methode zurück, nachdem sie die Anfrage erzeugt und verschickt hat. Ein asynchroner Aufruf wird durch eine einfache Pfeilspitze dargestellt. Die Antwort eines asynchronen Methodenaufrufs wird mittels einer Callback-Methode [146] übergeben. In den Abbildungen 3.15 und 3.16 ist der Methodenaufruf auf der Dienstanbieterseite synchron.

Das Message Exchange Pattern in den Abbildungen 3.15 und 3.16 ist ebenfalls synchron. Synchron heißt in diesem Falle nicht, dass der Dämon solange blockiert, bis ein Content Object empfangen wurde. Jedem versendeten Content Object ist ein Interest vorausgegangen und daher erfolgt eine Synchronisierung durch die Nachrichtenübertragung bei CCN. Das Message Exchange Pattern bei CCN ist synchron (vgl. Abbildung 3.3 auf Seite 43).

DEFAULT-METHODE

Neben den Dienstmethoden gibt es bei den namenzentrischen Diensten die Standard-Dienstmethode (engl. Default Service Method), kurz *Default-Methode*. Sie wird immer dann aufgerufen, wenn ein Dienst einen Interest erhält, der von keiner Dienstmethode verarbeitet wird. Die Default-Methode wird optional von

namenszentrischen Diensten implementiert und hat keinen Rückgabewert. Wird die Default-Methode aufgerufen, erhält sie den Interest, der nicht verarbeitet wurde.

Die Default-Methode wird von den Basisdiensten in Kapitel 7 benutzt. Sie dient bei Kommunikationsdiensten dazu, Interests zu manipulieren und die manipulierten Interests weiterzuleiten. Eine weitere Anwendung der Default-Methode ist eine Fehlerbehandlung. Beispielsweise ist es mit der Default-Methode möglich, Interests zu zählen, die nicht verarbeitet wurden.

Abbildung 3.18 zeigt den Ablauf in einem Flussdiagramm, der beim Empfang eines Interests am Dienstanbieter ausgeführt wird. Das Flussdiagramm zeigt insbesondere, wie entschieden wird, ob die Default-Methode aufgerufen wird, oder nicht. Die fett gedruckten Pfeile im Flussdiagramm zeigen die möglichen Pfade, die zum Aufruf der Default-Methode führen. Ein Flussdiagramm besteht aus Tätigkeiten, *Operationen* genannt, die als rechteckige Blöcke dargestellt werden, und aus *Entscheidungen*, dargestellt als Rauten.

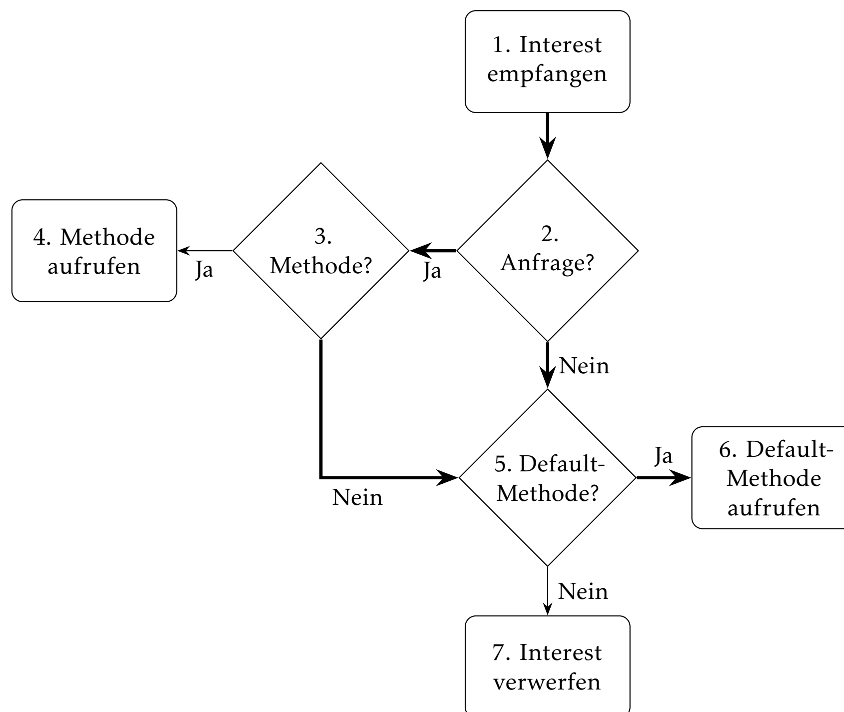


ABBILDUNG 3.18 – *Ablauf zum Aufruf der Default-Methode*

Der Dienstanbieter-Proxy empfängt einen Interest (*Operation 1*). Handelt es sich um eine Anfrage (*Entscheidung 2*), weil in dem empfangenen Interest ein Methodenaufruf kodiert ist, prüft der Dienstanbieter-Proxy, ob der Dienstanbieter die Methode implementiert (*Entscheidung 3*). Wenn ja, wird die Dienstmethode am Dienstanbieter aufgerufen (*Operation 4*). Enthält der Interest keine Methode oder wird die Methode vom Dienstanbieter nicht implementiert, wird geprüft, ob der Dienstanbieter die Default-Methode implementiert (*Entscheidung 5*). Wenn ja, wird die Default-Methode aufgerufen (*Operation 6*). Die Implementierung der

Default-Methode am Dienstanbieter bekommt den Interest übergeben. Wird die Default-Methode nicht implementiert, wird der Interest vom Dienstanbieter verworfen (*Operation 7*).

3.3.3 DIENSTBESCHREIBUNG

Ein Dienst liefert, nach Definition 2.2, eine Beschreibung der Funktionalitäten, die er bereitstellt. Die Funktionalitäten werden über eine Schnittstelle angesprochen, bei namenszentrischen Diensten sind das die Dienstmethoden. Die Beschreibung der Methoden eines Dienstes finden sich in einem Dokument beziehungsweise einer Datei, der sogenannten *Dienstbeschreibung*. Eine Dienstbeschreibung beschreibt die Methoden mit ihrer Signatur (vgl. Abbildung 3.12) sowie die verwendeten Datentypen. Datentypen sind vergleichbar mit den Strukturdatentypen der Programmiersprache C [25] und bestehen wie diese aus primitiven Datentypen oder wiederum aus strukturierten Datentypen. Weiterhin enthält die Dienstbeschreibung Zusatzinformationen/Metadaten, wie beispielsweise die Version eines Dienstes.

Die Dienstbeschreibung dient als Eingabe für das Werkzeug zur automatischen Generierung von Dienstanbieter- und Dienstnutzer-Proxy sowie von *Code-Gerüsten* (engl. Stub Code, kurz *Stub*), welche die Methodenrümpfe enthalten. Der Nutzer startet das Werkzeug mit der Dienstbeschreibung und einer Konfiguration (beispielsweise Kommandozeilenparametern). Daraufhin erstellt das Werkzeug in Abhängigkeit von der Konfiguration Quellcode-Dateien für den Dienstanbieter oder den Dienstnutzer. Abbildung 3.19 zeigt den Vorgang der werkzeugunterstützten Codegenerierung. Eine detaillierte Beschreibung der werkzeugunterstützten Codegenerierung wird in Kapitel 5 vorgenommen.

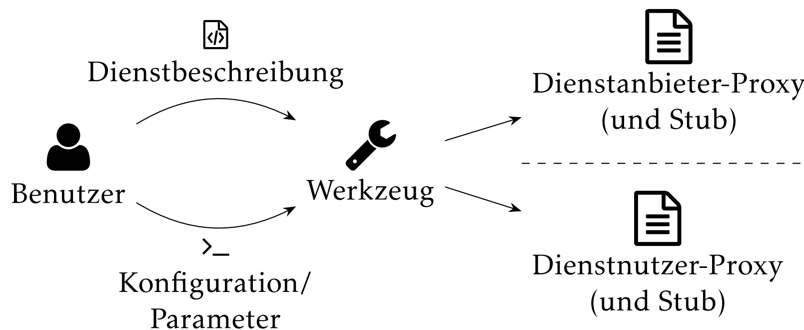


ABBILDUNG 3.19 – *Werkzeugunterstützte Codegenerierung*

3.4 ZUSAMMENFASSUNG

In diesem Kapitel wurden die Herausforderungen und Potenziale bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge diskutiert und das Konzept der namenszentrischen Dienste eingeführt. Insgesamt wurden vier Herausforderungen bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge identifiziert: *(i)* der *konzeptionelle Unterschied* zwischen knotenzentrischen und inhaltszentrischen Ansatz, *(ii)* die *Entwicklung* von *Diensten* mit

dem inhaltszentrischen Ansatz, *(iii)* die *Adressierung* mit *Namen*, insbesondere mit langen, sprechenden Namen sowie *(iv)* das starre In-Out *Messaging Pattern* des inhaltszentrischen Ansatzes. Den Herausforderungen stehen bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge auch Potenziale gegenüber: *(i)* Das *flexible Knotenmodell* skaliert und erlaubt die Entwicklung von lose gekoppelten Diensten. *(ii)* Die Adressierung mit *Namen* ist flexibel. So lässt sich beispielsweise eine kontextabhängige Adressierung realisieren, die mit knotenzentrischen Ansätzen deutlich schwieriger umzusetzen ist. *(iii)* Das Internet der Dinge profitiert auch von den Caching-Mechanismen des inhaltszentrischen Ansatzes, da es neben kurzlebigen Daten ebenso langlebige Daten (z. B. Konfigurationsdaten) gibt, die vom Caching profitieren. *(iv)* Einfache Messaging Pattern wie In-Only (vgl. Abbildung 3.3(a) auf Seite 43) lassen sich mit kurzlebigen Interests umsetzen. Geringe Datenmengen, wie Sensorwerte, werden dabei im Namen des Interests transportiert.

Die Potenziale des inhaltszentrischen Ansatzes im Internet der Dinge sind die Motivation für das Konzept der *namenzentrischen Dienste*. Das Konzept der namenzentrischen Dienste sieht vor, dass die gesamte Funktionalität auf Geräten im Internet der Dinge mit Diensten realisiert wird. Dazu zählen auch Basisfunktionalität wie die Kommunikation. So sind in dem erarbeiteten Konzept beispielsweise Dienste vorgesehen, die den Nachrichtenaustausch über Knotengrenzen realisieren (vgl. Abbildung 3.10 auf Seite 52).

Namenszentrische Dienste wenden dabei existierende etablierte Verfahren der serviceorientierten Architekturen an. So gibt es bei den namenszentrischen Diensten den entfernten Methodenaufruf der Dienstmethode. Der entfernte Methodenaufruf wird wie bei Middleware-Ansätzen (z. B. SOAP, RPC) üblich, über sogenannte Proxies realisiert. Weiterhin definieren die namenszentrischen Dienste eine sogenannte *Default-Methode*. Die Default-Methode wird von Dienstanbietern optional implementiert und verarbeitet alle Interests, die nicht von einer Methode auf der Dienstanbieterseite verarbeitet wurden. Verwendung findet die Default-Methode bei den in Rahmen der Arbeit entwickelten Basisdiensten, die in Kapitel 7 vorgestellt werden.

Wie auch bei SOAP-Webservices gibt des für namenszentrische Dienste eine Dienstbeschreibung und eine Werkzeugunterstützung. Mit der Werkzeugunterstützung wird aus der Dienstbeschreibung Quellcode für die Proxies sowie Stub-Code für Dienstanbieter und Dienstnutzer generiert. Kapitel 5 stellt die Details zur Dienstbeschreibung sowie zur Werkzeugunterstützung vor.

Im folgenden Kapitel 4 wird die Architektur des inhalts-/namenszentrischen Internet der Dinge vorgestellt, die im Rahmen der Arbeit entwickelt wurde. Die Architektur dient als Grundlage für die Implementierung der namenszentrischen Dienste auf ressourcenbeschränkten Geräten, die ebenfalls im folgenden Kapitel eingeführt wird.

KAPITEL 4

ARCHITEKTUR UND IMPLEMENTIERUNG

EIN wesentlicher Beitrag der Architektur ist die Identifikation der *Komponenten* des inhalts-/namenszentrischen Internet der Dinge und der *Schnittstellen* zwischen diesen Komponenten. Mit der Architektur der Softwarekomponenten erfolgt der Übergang vom Konzept zur Implementierung von CCN für ressourcenbeschränkte Geräte im inhalts-/namenszentrischen Internet der Dinge. Der erste Beitrag der Implementierung ist die Umsetzung der Datenstrukturen für CCN-IoT. Neue Datenstrukturen von CCN-IoT sind die Buffer (dt. Puffer) zur Speicherung von Namen und Nachrichten, die für die Nachrichtenverarbeitung relevant sind. Die Buffer wurden in „Architecture and Message Processing for Name-Centric Services in Wireless Sensor Networks" [3] vorgestellt. In dieser Veröffentlichung wurde auch die Nachrichtenverarbeitung evaluiert. Der zweite Beitrag der Implementierung umfasst die Optimierungsstrategien der CCN-Datenstruktur Forwarding Information Base (FIB) sowie deren Evaluation. Die Optimierungsstrategien der FIB und die Evaluation wurden in „Memory Efficient FIB for Content-Centric Networking" [1] veröffentlicht.

Das Kapitel gliedert sich wie folgt: in Abschnitt 4.1 werden verwandte Arbeiten vorgestellt. Abschnitt 4.2 stellt die System- und die Softwarearchitektur vor. Die Softwarearchitektur bildet die Überleitung zur Implementierung in Abschnitt 4.3. In diesem Abschnitt werden die Datenstruktur Buffer, die Nachrichtenverarbeitung und die Implementierung der CCN-Datenstrukturen vorgestellt. Die Speicheroptimierung der Forwarding Information Base findet sich am Ende von Abschnitt 4.3.

4.1 VERWANDTE ARBEITEN

Neben der in dieser Arbeit umgesetzten Implementierung gibt es weitere Implementierungen von CCNx, auch für ressourcenbeschränkte Geräte. Zu nennen

sind hier CCN-lite [145] von Scherb et al., eine CCNx Implementierung für Contiki [140] von Saadallah et al. und CCN-WSN [135] von Zhong et al. CCN-lite ist eine vereinfachte CCN-Implementierung, ohne die Sicherheitsmechanismen von CCN, die hauptsächlich zum Studium des Protokollverhaltens entwickelt wurde. Die CCNx-Implementierung von Contiki ist, ähnlich wie CCN-lite, eine Machbarkeitsstudie und durch das darunterliegende Contiki-Betriebssystem ressourcenintensiver als die Implementierung in dieser Arbeit. CCN-WSN basiert nicht auf einem Betriebssystem und ist gezielt für inhaltszentrische Ansätze in drahtlosen Sensornetzen entworfen worden.

Die Implementierung, die im Rahmen dieser Arbeit entstanden ist, basiert auf CCN-WSN. CCN-WSN weist Schwächen auf, die bei der Umsetzung der namenszentrischen Dienste behoben werden mussten. Eine Schwäche von CCN-WSN ist die Verarbeitung von Namen. So sind Prozentkodierung (vgl. Definition 2.7 und Beispiel 2.1) sowie die Ordnung auf Namen (vgl. Definition 2.9) im Entwurf von CCN-WSN nicht berücksichtigt.

CCN wurde daher unter dem Namen *CCN-IoT* für die drahtlosen Sensorknoten noch einmal von Grund auf neu entwickelt. In Abgrenzung zu CCN-WSN wird die neue Entwicklung CCN-IoT genannt. Wie CCN-WSN wird auch die neue Entwicklung als Teil der C++ Wiselib Template-Bibliothek von Baumgartner et al. [37] implementiert. Die Wiselib wurde bereits auf viele unterschiedliche Hardwareplattformen portiert und somit ist die Implementierung von CCN-IoT auch auf diesen Hardwareplattformen ohne eine komplizierte Portierung lauffähig.

Ein Merkmal von CCN-IoT ist das flexible Nachrichtenformat, das eine Mischung aus Feldern fester Länge und variabler Länge vorsieht. Die Felder variabler Länge sind ähnlich dem Nachrichtenformat von ASN.1 [93, 64] aufgebaut. Das Nachrichtenformat von ASN.1 ist eine Type-Length-Value-Kodierung (TLV, dt. Typ-Länge-Wert). Unabhängig von CCN-IoT wurde mit CCNx 1.0 [112] parallel eine TLV-Kodierung für Nachrichten eingeführt. Das Nachrichtenformat von CCN-IoT ist im Gegensatz zu dem von CCNx 1.0 an drahtlose Sensornetze angepasst.

Zusammengefasst ist CCN-IoT eine Lösung, ressourcenbeschränkte Geräte, wie drahtlose Sensorknoten, Teil des inhalts-/namenszentrischen Internet der Dinge werden zu lassen. Bisherige Implementierungen von CCN vernachlässigen Geräte wie batteriebetriebene Sensorknoten oder sie sind für die Realisierung von namenszentrischen Diensten nicht geeignet. Ebenso ist es bei den verwandten Arbeiten nicht direkt ersichtlich, wie die ressourcenbeschränkten Geräte Teil des Internets werden. In Abschnitt 4.2.1 Systemarchitektur werden Gateways vorgeschlagen, die drahtgebundene und drahtlose Netze miteinander verbinden und so die ressourcenbeschränkten Geräte in das Internet integrieren.

4.2 ARCHITEKTUR

Dieser Abschnitt führt die Architektur der namenszentrischen Dienste für das Internet der Dinge ein und gliedert sich in *Systemarchitektur* und *Softwarearchitektur*. Die Systemarchitektur führt die Komponenten und die Schnittstellen

zwischen den Komponenten ein, die Softwarearchitektur präsentiert den internen Aufbau der Softwarekomponenten und beschreibt, wie Komponenten über die Schnittstellen interagieren.

4.2.1 SYSTEMARCHITEKTUR

In diesem Abschnitt wird die Systemarchitektur des inhalts-/namenszentrischen Internet der Dinge eingeführt. Die Systemarchitektur beschreibt die *Hard-* und die *Softwarekomponenten*.

Hardwarekomponenten im Internet der Dinge sind zum Beispiel die Geräte, die die Dinge repräsentieren. Das sind in dieser Arbeit ressourcenbeschränkte Geräte, die drahtlose Ad-hoc-Netze aufbauen und darüber kommunizieren. Da das Internet der Dinge in dieser Arbeit als eine Teilmenge des Internets betrachtet wird, gibt es im Internet der Dinge weitere Netzwerkkomponenten, wie Router, sowie Endgeräte, wie Server und PCs. Im Internet kommen breitbandige Übertragungstechnologien zum Einsatz, die in der Regel drahtgebunden sind. Daher werden im Folgenden breitbandige Netze als drahtgebundene Netze bezeichnet.

Der Übergang zwischen drahtgebundenen und drahtlosen Netzen wird mit Hardwarekomponenten realisiert, die in dieser Arbeit als *Gateways* bezeichnet werden. Gateways haben eine Funkschnittstelle, mit der sie sich zu einem drahtlosen Ad-hoc-Netz verbinden. Darüber hinaus sind sie mit dem drahtgebundenen Internet verbunden. Gateways vermitteln zwischen dem drahtgebundenen und dem drahtlosen Netz, indem sie Pakete aus dem einen Netz in das andere weiterleiten und dabei eine Paketkonvertierung vornehmen. In dieser Arbeit wird davon ausgegangen, dass Gateways über mehr Ressourcen (Rechenleistung und Speicher) verfügen, als die ressourcenbeschränkten Geräte im drahtlosen Ad-hoc-Netz.

Im Kontext der namenszentrischen Dienste sind es die ressourcenbeschränkten Geräte, Server, PCs und Gateways die die Dienstinstanzen ausführen. Dienstinstanzen, Proxies und die CCN-Implementierungen sind die *Softwarekomponenten*, die bei der Systemarchitektur betrachtet werden.

Hard- und Softwarekomponenten des inhalts-/namenszentrischen Internet der Dinge werden in Abbildung 4.1 gezeigt. In der Abbildung sind oben zuerst die Hardwarekomponenten, dann unten die Softwarekomponenten dargestellt. Von links beginnend stellen Sensorknoten stellvertretend die drahtlosen Ad-hoc-Netze im Internet der Dinge dar. Rechts von den Sensorknoten befindet sich ein Gateway (als „GW“ in Abbildung 4.1 markiert), das den Übergang vom drahtlosen Ad-hoc-Netz in das drahtgebundene Internet realisiert und so das Ad-hoc-Netz in das Internet integriert. Das Sensornetz und das Gateway liegen am Rande des inhalts-/namenszentrischen Internets, da durch Sensornetze kein allgemeiner Netzverkehr geleitet wird. Daher sind Gateways in inhalts-/ namenszentrischen Netzen vergleichbar mit 6LoWPAN Edge-Routern [122].

Das drahtgebundene Internet ist als Wolke dargestellt. Im Internet gibt es Netzwerkkomponenten, wie Router, sowie Endgeräte, wie Server und PCs. Der Laptop und der Server ganz rechts stellen dedizierte Geräte im Internet dar, die namenszentrische Dienste ausführen. Unterhalb Hardwarekomponenten sind

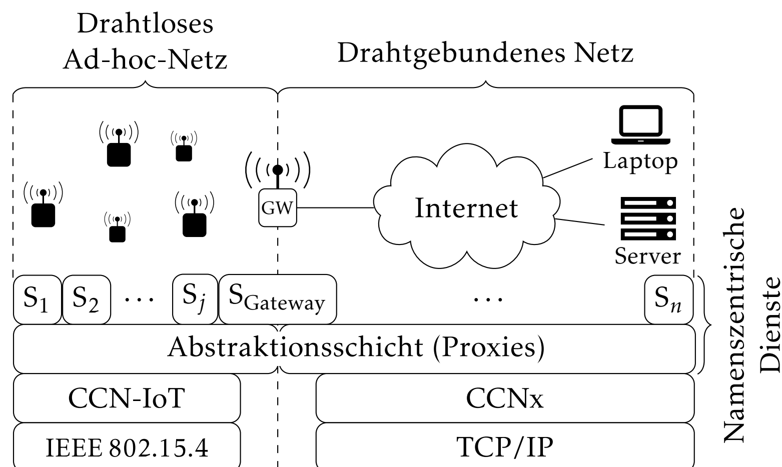


Abbildung 4.1 – *Überblick über die Gesamtarchitektur: namenszentrische Dienste und Hardwarekomponenten*

in Abbildung 4.1 die Softwarekomponenten als Protokollstapel dargestellt. Die oberen beiden Schichten bilden die namenszentrischen Dienste, wobei sich auf der obersten Schicht die Dienstinstanzen befinden. Dienstinstanzen S_1 bis S_j werden auf den Sensorknoten und $S_{Gateway}$, der Gateway-Dienst, wird auf dem Gateway ausgeführt. Der Gateway-Dienst ist ein Basisdienst und implementiert die Paketkonvertierung. Er wird in Kapitel 7 behandelt. Dienstinstanz S_n wird im drahtgebundenen Internet ausgeführt.

Wie in Abschnitt 3.3.2 dargestellt, kommunizieren namenszentrische Dienste nicht direkt über CCN, sondern nutzen eine Abstraktionsschicht, repräsentiert durch die Proxies. Dienste und Proxies sind in Abbildung 4.1 als namenszentrische Dienste zusammengefasst. Die CCN-Schicht setzt im drahtlosen Netz direkt auf Protokolle wie IEEE 802.15.4 auf. Im drahtgebundenen Netz wird CCN als logisches Netz über TCP/IP betrieben.

Die Softwarekomponenten sind nun in der Architektur soweit eingeführt. In der Darstellung der Architektur fehlen allerdings die Schnittstellen zwischen den Komponenten. In dem Konzept in Abschnitt 3.3 werden Dienstmethoden und Faces als Schnittstellen eingeführt. Im Folgenden wird erklärt, wo sich Dienstmethoden und Faces in der Architektur befinden.

Betrachtet man den Protokollstapel in Abbildung 4.1, befinden sich in vertikaler Richtung die Zugänge zu den Diensten der jeweiligen Schichten. Diese dienen auch als Programmierschnittstellen (engl. Application Programming Interfaces, kurz API). In horizontaler Richtung nehmen die Dienstinstanzen die Rollen von *Dienstanbieter* und *Dienstnutzer* ein. Auf den beiden Schichten, die den namenszentrischen Diensten zugerechnet werden, realisieren die Dienstmethoden die Service-Schnittstellen zwischen den Dienstinstanzen. Die Dienstmethoden sind – wie RPC bei SOAP – das Protokoll, über das namenszentrische Dienste kommunizieren (vgl. Dienst und Protokoll im ISO/OSI-Referenzmodell in Abbildung 2.5 auf Seite 23).

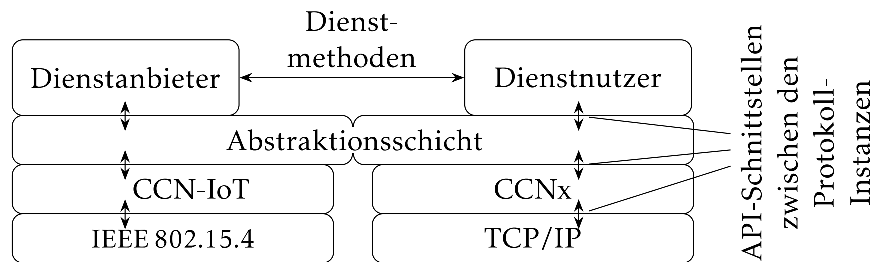


Abbildung 4.2 – *Schnittstellen zwischen Diensten und Protokollen*

Abbildung 4.2 zeigt die Schnittstellen zwischen den Softwarekomponenten in einem Netzwerkstack als Doppelpfeile. In vertikaler Richtung sind, wie oben beschrieben, die Programmierschnittstellen, und in horizontaler Richtung sind die Dienstmethoden. Ein Dienstnutzer benötigt eine Schnittstelle, die von einem Dienstanbieter angeboten wird. Die Unified Modeling Language (UML) 2.0 [139] unterscheidet daher in Komponentendiagrammen zwischen *angebotener* und *benötigter* Schnittstelle. Im Folgenden werden UML Komponentendiagramme benutzt, um die Schnittstellen genauer darzustellen.

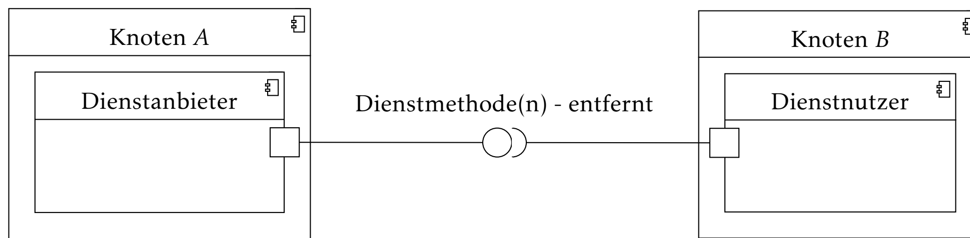
Im Komponentendiagramm wird die angebotene Schnittstelle als Kreis und die benötigte Schnittstelle als Halbkreis zwischen der Verbindung zwischen den Komponenten dargestellt. Kleine Rechtecke an der Verbindung der Komponente zur Schnittstelle werden *Ports* genannt und zeigen an, dass die Schnittstelle aus mehreren Dienstmethoden besteht.

Das Komponentendiagramm in Abbildung 4.3 zeigt mit der *Dienstmethode* die Schnittstelle zwischen Dienstanbieter und Dienstnutzer. Dienstanbieter und Dienstnutzer sind Softwarekomponenten, die auf Hardwarekomponenten, den *Knoten* ausgeführt werden. Die Darstellung in Abbildung 4.3 gilt für Sensorknoten, PCs oder Server. Nach dem Konzept ist es unerheblich, ob Dienstanbieter und Dienstnutzer auf dem selben Knoten oder auf unterschiedlichen Knoten ausgeführt werden.

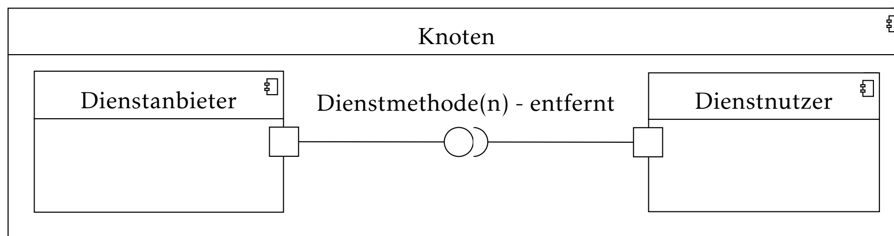
Ruft also ein Dienstnutzer eine Methode am Dienstanbieter auf, bedeutet das nicht, dass die Dienste, wie in Abbildung 4.3(a) notwendigerweise auf unterschiedlichen Knoten ausgeführt werden. Auch wenn die Dienste, wie in Abbildung 4.3(b) auf einem Knoten ausgeführt werden, spricht man von einem entfernten Aufruf. In Abbildung 4.3 wird noch einmal deutlich, dass nur Dienste miteinander kommunizieren und nicht Knoten, was den Ansatz der namenszentrischen Dienste von knotenzentrischen Ansätzen abgrenzt.

In vertikaler Richtung, zwischen der CCN-Schicht und den namenszentrischen Diensten in Abbildung 4.2 sind die Schnittstellen die *Faces*, die das Komponentendiagramm in Abbildung 4.4 dargestellt. Die CCN-Schicht wird durch die Softwarekomponente CCN-Dämon repräsentiert. Dienstinstanzen sind mit dem Dämon über die Face-Schnittstelle verbunden, worüber sie Interests und Content Objects senden und empfangen.

Eine Dienstinstanz besteht aus dem Dienstanbieter und gegebenenfalls aus einem Dienstnutzer und den Proxies. Der Dienstanbieter bietet API-Schnittstellen



(a) Dienstanbieter-/nutzer auf unterschiedlichen Knoten



(b) Dienstanbieter-/nutzer auf einem Knoten

Abbildung 4.3 – *Dienstmethode als Schnittstelle zum entfernten Aufruf zwischen dem Dienstanbieter und Dienstnutzer*

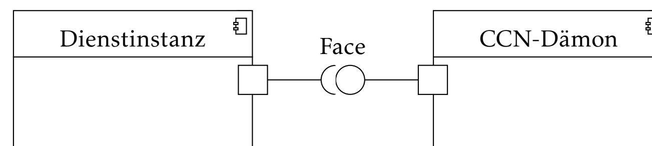


Abbildung 4.4 – *Face-Schnittstelle zwischen Dienstinstanz und CCN-Dämon*

zum Aufruf der Dienstmethoden an, die vom Dienstanbieter-Proxy aufgerufen werden, wenn sie eine Anfrage verarbeitet haben. Der Dienstanbieter-Proxy serialisiert die Rückgabewerte von Methodenaufrufen und er ruft die lokalen Dienstmethoden auf, die der Dienstanbieter bereitstellt. Der Dienstanbieter-Proxy benötigt den Dienstanbieter, da er die Programmlogik des Dienstes enthält.

Interests, die an den Dienstanbieter gerichtet sind, aber keiner Dienstmethode zugeordnet werden können, werden von der Default-Methode des Dienstanbieters verarbeitet, sofern diese implementiert ist (vgl. Default-Methode in Kapitel 3 auf Seite 58). Da Dienstnutzer die Default-Methode nicht bewusst aufrufen, ist sie in Abbildung 4.3 nicht dargestellt.

Der Dienstnutzer nutzt entfernte Dienste, indem er die lokalen Dienstmethoden des Dienstnutzer-Proxies aufruft. Der Aufruf einer Dienstmethode hat zur Folge, dass der Dienstnutzer-Proxy einen Interest erstellt und verschickt. Rückgabewerte von entfernten Methodenaufrufen werden vom Dienstnutzer-Proxy deserialisiert und dem Dienstnutzer übergeben.

Abbildung 4.5 fasst Dienstanbieter, Dienstnutzer, die Proxies und die API-Schnittstellen in einem Komponentendiagramm zusammen. Dienstanbieter, Dienstnutzer und die Proxies sind Komponenten, Dienstmethoden sind die Schnittstellen. Der Dienstanbieter bietet Schnittstellen für den Dienstanbieter-Proxy an, da die Kreise der Schnittstelle im Komponentendiagramm in Abbildung 4.5(a) auf der Seite des Dienstanbieters sind. Unter den Dienstmethoden in Abbildung 4.5(a) befindet sich die API-Schnittstelle der Default-Methode. Auf der Dienstnutzerseite in Abbildung 4.5(b) bietet der Dienstnutzer-Proxy die Schnittstellen zum Aufruf der Dienstmethoden an.

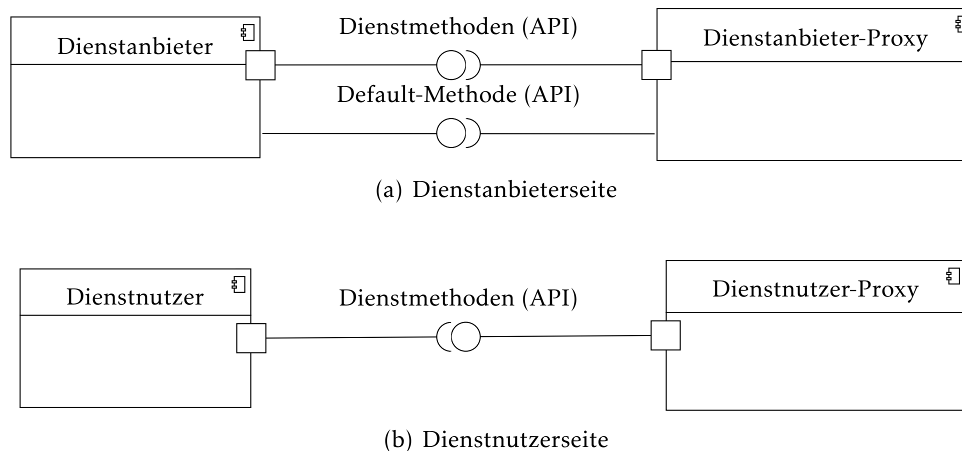


ABBILDUNG 4.5 – *Komponenten und API-Schnittstellen zwischen Dienstanbieter und Dienstnutzer und den Proxies*

Einige Dienste brauchen andere Dienste, um ihre Funktion bereitzustellen. Das heißt, dass Dienstanbieter und Dienstnutzer wie in Abbildung 4.6 in einer Komponente vereint sind. Die Schnittstellen in Abbildung 4.6 sind ohne Bezeichnung abgebildet, sie entsprechen den Schnittstellen aus Abbildung 4.5.

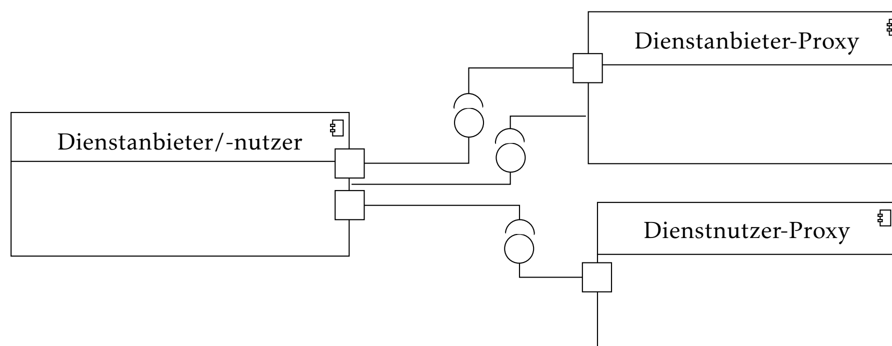


ABBILDUNG 4.6 – *Dienstanbieter und Dienstnutzer in einer Komponente vereint*

Zusammengefasst laufen die Softwarekomponenten Dienstinstanz und CCN-Dämon auf einem Knoten. Ein Knoten ist nach der Architektur des inhaltszentrischen Internet der Dinge ein ressourcenbeschränktes Gerät in einem draht-

losen Ad-hoc-Netz oder ein Gerät im drahtgebundenen Internet, welches über mehr Ressourcen als Geräte in einem drahtlosen Ad-hoc-Netz verfügt. Eine Dienstinstanz besteht wiederum aus Softwarekomponenten wie Dienstanbieter und, wenn der Dienstanbieter andere Dienste für seine Funktion braucht, auch Dienstnutzer sowie deren Proxies. Die Proxies übernehmen anstelle von Dienstanbieter und Dienstnutzer die Erstellung, Verarbeitung sowie das Senden und Empfangen von Interests und Content Objects. Proxies entsprechen im CCN-Knotenmodell den Anwendungen. Damit eine Anwendung Interests und Content Objects über den Dämon senden und empfangen kann, ist sie über die Face-Schnittstelle mit dem Dämon verbunden. Die Regeln für die Nachrichtenverarbeitung in CCN, beschrieben in den Grundlagen in Abschnitt 2.4.4, verdeutlichen, warum eine Anwendung und damit ein Proxy immer mit einem Face mit dem Dämon verbunden ist: *(i)* Sendet beispielsweise ein Dienstnutzer-Proxy einen Interest, dann wird das Content Object, das als Antwort am Dämon empfangen wird, durch den Eintrag in der Pending Interest Table an den Dienstnutzer-Proxy weitergeleitet. *(ii)* Ein Dienstanbieter registriert einen Dienstanbieter-Proxy mit einem bestimmten Präfix in der Forwarding Information Base, damit dieser Interests empfängt und verarbeitet, für ihn bestimmt sind.

Ein Beispiel für Softwarekomponenten auf einem Knoten zeigt Abbildung 4.7. Die Hardwarekomponente *Knoten* führt die *Dienstinstanz* und den *CCN-Dämon* aus. Die Dienstinstanz in dem Beispiel besteht aus einer Softwarekomponente, die Dienstanbieter-/nutzer kombiniert und mit *DA/DN* (für Dienstanbieter/ Dienstnutzer) gekennzeichnet ist. Dienstanbieter- und Dienstnutzer-Proxy sind jeweils eigenständige Softwarekomponenten, da sie durch die Werkzeugunterstützung auch einzeln generiert werden. Sie sind über die Face-Schnittstelle mit dem *CCN-Dämon* verbunden, der eine weitere Softwarekomponente auf dem Knoten ist.

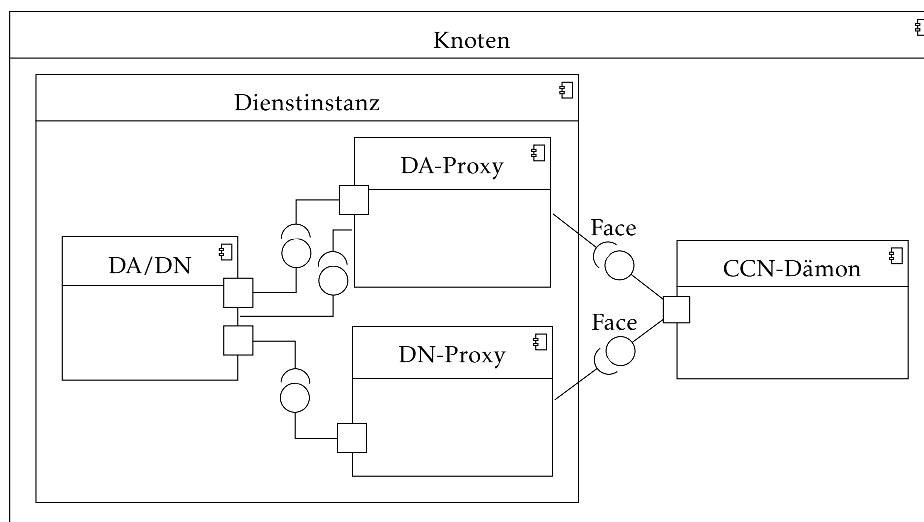


Abbildung 4.7 – *Beispiel einer Knotenarchitektur mit den Softwarekomponenten*

4.2.2 SOFTWAREARCHITEKTUR

Nach der Diskussion der Systemarchitektur im vorangegangen Abschnitt wird nun der Aufbau der Softwarekomponenten der Systemarchitektur betrachtet. Insbesondere werden hier die Beziehungen und Interaktionen zwischen den Komponenten beziehungsweise zwischen den Klassen vorgestellt, aus denen die Komponenten aufgebaut sind.

Im Rahmen der Arbeit wurde eine CCN-Implementierung für ressourcenbeschränkte Geräte entwickelt, die den Anforderungen der namenszentrischen Dienste genügt. Diese Implementierung wird CCN-IoT genannt. Die Darstellung der Softwarearchitektur erfolgt am Beispiel von CCN-IoT, da in Abschnitt 4.3 anhand von CCN-IoT gezeigt wird, wie mit Herausforderungen durch lange, sprechende Namen umgegangen wird (vgl. Herausforderungen in Abschnitt 3.2.1).

Eine wesentliche Anforderung an CCN-IoT war die Einfachheit, denn zusätzliche Komplexität wäre beim Umgang mit den Herausforderungen, die sich bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge ergeben, kontraproduktiv. Ein Merkmal von Einfachheit ist die Kapselung von Information. Daher verwaltet der Dämon die Datenstrukturen Content Store, Pending Interest Table und Forwarding Information Base selbst und lässt einen Zugriff von außen, das heißt durch die Anwendungen, nur begrenzt durch API-Schnittstellen zu. Die wichtigste Schnittstelle zwischen Dämon und den Anwendungen ist das *Face*. Bei ressourcenbeschränkten Geräten bietet sich ein asynchroner Methodenaufruf beim Empfang von Nachrichten an, da synchrones, blockierendes Warten auf Nachrichten auf ressourcenbeschränkten Geräten häufig nicht möglich ist oder zu viel Ressourcen verbraucht. Um daher asynchrone Methodenaufrufe zu realisieren, bieten sich Interfaces an. Eine Anwendung implementiert ein Interface und damit eine Callback-Methode, die beim Nachrichtenempfang durch den Dämon aufgerufen wird. Über eine API-Schnittstelle versenden Anwendungen Nachrichten zum Dämon.

In Abbildung 4.8 sind die wesentlichen Klassen der Implementierung von CCN-IoT dargestellt. Mit dem *Face* werden in CCN-IoT die Callback-Methoden sowie die API-Schnittstelle zum Empfangen und Senden von Nachrichten implementiert. Teil des Dämons sind die Klassen für den Content Store (CS), der Pending Interest Table (PIT) und der Forwarding Information Base (FIB). Klassen, die zu CCN gehören, wird der Namensraum *ccn* vorangestellt, wobei Namensraum und Klassenname der C++-Notation folgend mit einem doppelten Doppelpunkt getrennt werden. Die Klassen Dämon, Content Store, Pending Interest Table und Forwarding Information Base bilden eine *Komposition*, dargestellt durch die ausgefüllte Raute. Bei einer Komposition sind die Datenstrukturen von der Existenz des Dämons abhängig, daher wird die Raute auf der Seite des Dämons dargestellt.

Die Beziehung zwischen dem Dämon und den Faces wird durch eine *Aggregation* beschrieben, dargestellt mit einer Raute auf der Seite des Dämons. Die Bindung der Aggregation ist dabei nicht so stark wie die Bindung der Komposition, da es auch CCN-Anwendungen oder Dienste gibt, die nicht am Dämon registriert sind, weil sie beispielsweise temporär inaktiv sind. Wie in Kapitel 7 „Basisdienste“

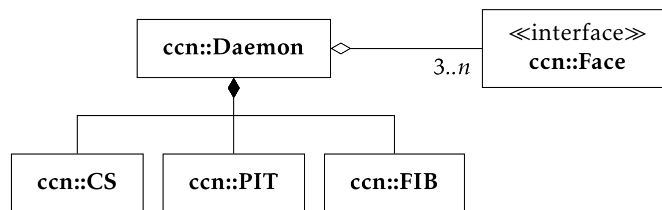


ABBILDUNG 4.8 – *Wesentliche Klassen von CCN-IoT*

gezeigt wird, ist der Dienst für die drahtlose Kommunikation über zwei Faces mit dem Dämon verbunden. Weiterhin gibt es mindestens noch einen Dienst, der den Kommunikationsdienst benutzt. Aus diesem Grunde gibt es immer mindestens drei Faces.

Die Proxies sind Anwendungen, daher implementieren sie das Interface Face, wie in Abbildung 4.9 an einem beispielhaften Dienst (ExampleService) dargestellt. Der Namensraum für die Proxies ist *ncs* (engl. für Name-Centric Services).

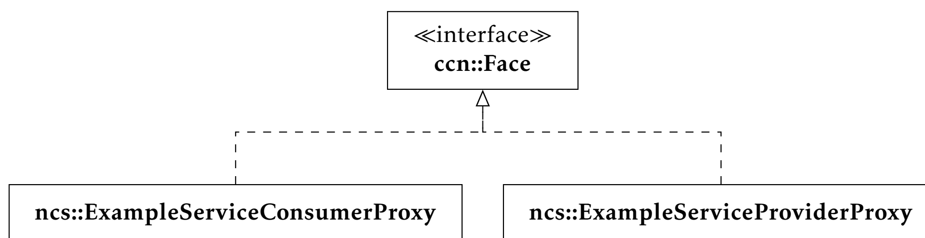


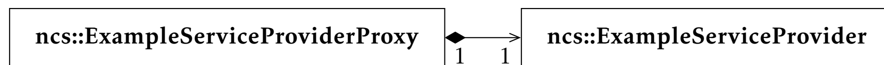
ABBILDUNG 4.9 – *Beziehung zwischen Face, Dienstanbieter- und Dienstnutzer-Proxy*

Strenggenommen wird das Face in der CCN-IoT-Implementierung nicht als C++-Interface implementiert, da die Wiselib virtuelle Methoden aus Kompatibilitätsgründen nicht erlaubt. Der Konstruktor von Face ist nur für die abgeleiteten Klassen sichtbar und die virtuellen Methoden werden über Methodenzeiger im Face realisiert, was letztlich ein C++-Interface nachbildet.

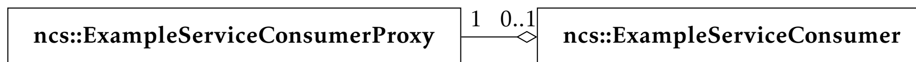
Dienstanbieter- und Dienstnutzer-Klassen interagieren mit den Proxy-Klassen. Bei der Interaktion stellt sich die Frage, welche Klassen andere Klassen wie benutzen. Wie in den vorherigen Abbildungen werden mit Kompositionen oder Aggregationen die Beziehungen zwischen den Klassen ausgedrückt. Bei der Ermittlung der Beziehungstypen wird beispielsweise analysiert, ob eine andere Klasse eine andere Klasse zum funktionieren braucht. Die Betrachtung der Beziehungstypen ist wichtig, weil damit die Instanziierung- und Initialisierungsreihenfolge der Instanzen der Klassen vorgegeben wird. Das Verständnis für die Beziehungen zwischen den Klassen sowie der Instanziierung und Initialisierung von Dienstanbieter-, Dienstnutzer- und Proxy-Instanzen ist wiederum wichtig für die Entwicklung und Implementierung der Werkzeugunterstützung.

Betrachtet man die Beziehung zwischen Dienstanbieter und Dienstanbieter-Proxy, zeigt sich, dass ein Dienstanbieter-Proxy nicht ohne einen Dienstanbieter existieren kann, da er ohne einen Dienstanbieter nicht in der Lage ist, eine Anfrage zu verarbeiten. Ein Dienstnutzer-Proxy hingegen kann ohne einen

Dienstnutzer existieren, da der Dienstnutzer-Proxy von sich aus keine Anfragen stellt. Damit ergeben sich für Dienstanbieter und Dienstnutzer sowie deren Proxies die in Abbildung 4.10 dargestellten Beziehungen.



(a) Dienstanbieterseite



(b) Dienstnutzerseite

ABBILDUNG 4.10 – *Beziehung zwischen Dienstanbieter, -nutzer und den Proxies*

Auf der Dienstanbieterseite in Abbildung 4.10(a) gibt es zu jeder Instanz eines Dienstanbieter-Proxy auch eine Instanz eines Dienstanbieters. Die Beziehung zwischen Proxy und Dienstanbieter ist als gerichtete Komposition dargestellt. Da die Raute auf der Seite des Proxies ist, hat der Proxy die Instanz des Dienstanbieters. Der Richtungspfeil sagt aus, dass der Proxy nur die Methoden des Dienstanbieters aufruft, nicht umgekehrt. Die Beziehung zwischen Proxy und Dienstanbieter zeigt mit einer ausgefüllten Raute eine starke Beziehung, da der Proxy ohne den Dienstanbieter nicht funktioniert.

Auf der Dienstnutzerseite in Abbildung 4.10(b) hat der Dienstnutzer die Instanz des Proxies, wie durch die Raute dargestellt. Die Bindung ist hier nicht so ausgeprägt wie auf der Dienstanbieterseite, da ein Dienstnutzer-Proxy auch ohne Dienstnutzer existieren kann. Dienstnutzer und Proxy greifen gegenseitig aufeinander zu, daher ist kein Richtungspfeil in der Beziehung in Abbildung 4.10(b) angegeben. Der Dienstnutzer ruft die Dienstmethoden beim Proxy auf, der Proxy gibt über Callback-Methoden die Rückgabewerte der entfernten Dienstmethodenaufrufe zurück.

Zusammengefasst lässt sich die Asymmetrie in den Beziehungen in Abbildung 4.10 auch kurz so erklären: Einen Dienst anbieten ohne ihn zu implementieren, stellt ein Problem dar, daher ist der Dienstanbieter notwendig. Einen angebotenen Dienst nicht zu nutzen, stellt dagegen kein Problem dar.

Bisher wurden nur die Beziehungen zwischen den Klassen betrachtet. Im Folgenden werden Instanziierungs- und Initialisierungsreihenfolge betrachtet. Die Instanziierungsreihenfolge auf Dienstanbieterseite geht dabei aus der Arbeitsweise des Dienstanbieter-Proxies hervor: Ein Dienstanbieter-Proxy bekommt über die Face-Schnittstelle einen Interest. Enthält dieser Interest einen Dienstmethodenaufruf, wird der Dienstmethodenaufruf deserialisiert und die entsprechende Dienstmethode der API-Schnittstelle des Dienstanbieters aufgerufen (vgl. Komponenten und API-Schnittstellen in Abbildung 4.5(a)).

Um die Methoden beim Dienstanbieter aufzurufen, braucht der Proxy eine Instanz des Dienstanbieters. Die Instanz des Dienstanbieters könnte man dem Proxy übergeben, das ist aber zu umständlich. Stattdessen erzeugt der Proxy die

Instanz des Dienstanbieters. Das funktioniert, weil der Stub-Code des Dienstanbieters, also das Gerüst der Klasse mit leeren Methodenrümpfen, durch die Werkzeugunterstützung generiert wird. Die Werkzeugunterstützung ist so geschrieben, dass Dienstanbieter und Dienstanbieter-Proxy exakt aufeinander abgestimmt werden. Ein Binden der Instanzen von Dienstanbieter und Proxy zur Laufzeit entfällt damit.

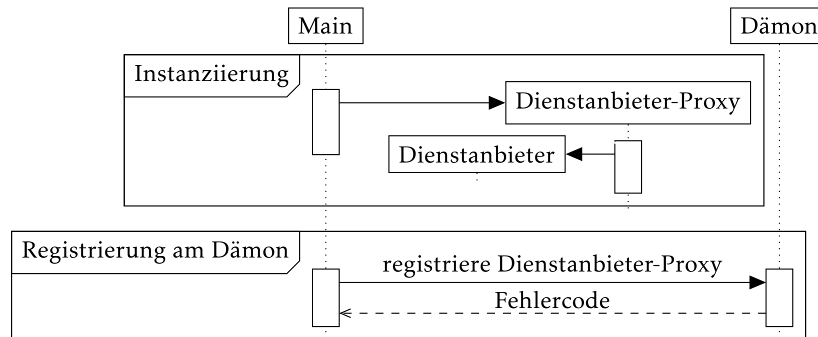
Auf der Dienstnutzerseite ist die Arbeitsweise wie folgt: Wird eine Dienstmethode der API-Schnittstelle am Dienstnutzer-Proxy aufgerufen, wird eine Anfrage – ein Interest mit einem serialisierten Dienstmethodenaufruf – erzeugt. Rückgabewerte der Dienstmethoden werden von den Dienstnutzer-Proxies aus den Antworten deserialisiert und über Callback-Methoden dem Dienstnutzer übergeben. Dazu hat sich der Dienstnutzer vorher am Proxy registriert. (In der Darstellung der Komponenten und API-Schnittstellen in Abbildung 4.5(b) sind Schnittstellen zur Registrierung der Callback-Methoden des Dienstnutzers am Proxy nicht dargestellt, da es sich hier um ein Implementierungsdetail von CCN-IoT handelt.)

Der Benutzer der API-Schnittstellen des Dienstnutzer-Proxies ist möglicherweise eine komplexe Anwendung, die noch weitere Dienste nutzt und damit weitere Dienstnutzer-Proxies. Aufgrund der Komplexität des Dienstnutzers ist es daher nicht möglich, Stub-Code für den Dienstnutzer zu erstellen. Aus diesem Grunde ist der Dienstnutzer eine Anwendung, die manuell erstellt wird und die genutzten Dienste werden über Dienstnutzer-Proxies eingebunden. Das heißt weiterhin, dass der Dienstnutzer die Dienstnutzer-Proxies instantiiert und initialisiert.

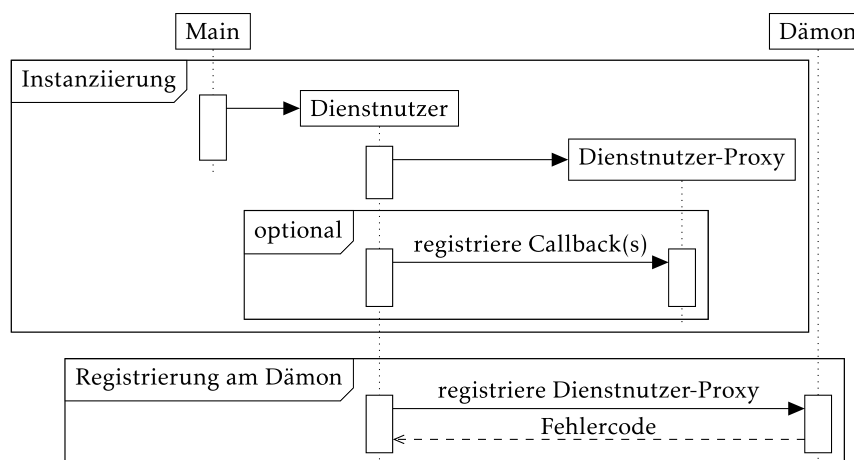
Aus den Arbeitsweisen von Dienstanbieter, Dienstnutzer und den Proxies ergeben sich damit für die Instanziierung und Initialisierung die zeitlichen Abläufe, die im Sequenzdiagramm in Abbildung 4.11 dargestellt sind. Die Instanziierung startet an einem Hauptprozess, der in den folgenden Sequenzdiagrammen „Main“ genannt wird. Kästen mit einer Beschriftung in der oberen linken Ecke annotieren Abschnitte im Sequenzdiagramm. Bei den Abschnitten mit der Annotation „Registrierung am Dämon“ handelt es sich um eine Initialisierung, da damit die Instanz dem Dämon bekannt gemacht wird.

In Abbildung 4.11(a) zeigt die *Instanziierung* den Ablauf der Erzeugung von Dienstanbieter und Dienstanbieter-Proxy und der Hauptprozess erzeugt den Dienstanbieter-Proxy. Der Dienstanbieter-Proxy erzeugt, wie oben beschrieben, den Dienstanbieter.

In Abbildung 4.11(b) erzeugt der Hauptprozess den Dienstnutzer. Jeder Dienstnutzer wiederum erzeugt einen Dienstnutzer-Proxy, da er von diesem die Dienstmethoden aufruft, die den entfernten Aufruf ausführen. Optional registriert der Dienstnutzer am Proxy Callback-Methoden, die beim Empfang der Antworten beim Dienstnutzer aufgerufen werden. Die Antworten sind die Rückgabewerte der entfernten Dienstmethoden. Diese Sequenz ist deswegen optional, da sie nur für Implementierungen ausgeführt wird, deren Dienstmethodenaufruf asynchron erfolgt (wie bei CCN-IoT) und auch dann nur für Dienstmethoden, die einen Rückgabewert haben.



(a) Dienstanbieterseite



(b) Dienstnutzerseite

Abbildung 4.11 – *Sequenzdiagramm zur Erzeugung der Softwarekomponenten und deren Registrierung am Dämon*

Auf der Dienstanbieter- als auch auf der Dienstnutzerseite wird der Proxy am Dämon registriert. Die Registrierung nimmt dabei immer der Ersteller des jeweiligen Proxies vor. Der Dämon liefert bei der Registrierung einen Rückgabewert, der anzeigt, ob die Registrierung erfolgreich war.

Zusammenfassend wurden mit der Systemarchitektur die Hard- und Softwarekomponenten sowie die Schnittstellen zwischen den Komponenten eingeführt. Die Softwarearchitektur betrachtete den Aufbau der Softwarekomponenten im Detail, wobei der Fokus bei der obigen Darstellung auf den Klassen der Softwarekomponenten und der Arbeitsweise der Klassen untereinander liegt, weil das wichtig für die Entwicklung der Werkzeugunterstützung ist, die in Kapitel 5 eingeführt wird. Im folgenden Abschnitt wird die Darstellung und Verarbeitung von Namen und Nachrichten in CCN-IoT sowie Strategien zur Optimierung des Speichers der Forwarding Information Base vorgestellt. Die Darstellung und Verarbeitung von Namen und Nachrichten und die Optimierungsstrategien der Forwarding Information Base sind Maßnahmen, um mit der in Abschnitt 3.2.1 identifizierten Herausforderung der langen, sprechenden Namen umzugehen.

Die Maßnahmen zum Umgang mit dieser Herausforderung sind im Kontext der Implementierung angesiedelt, daher werden sie im folgendem Abschnitt dargestellt.

4.3 IMPLEMENTIERUNG

Dieser Abschnitt behandelt die Implementierung von CCN-IoT und stellt zwei Maßnahmen zum Umgang mit langen, sprechenden Namen bei der Anwendung des inhalts-/namenszentrischen Ansatzes im Internet der Dinge vor. Eine Maßnahme ist die Darstellung und Verarbeitung von Namen und Nachrichten in CCN-IoT. Dazu wird zuerst erläutert, warum sich die Darstellung und Verarbeitung von Nachrichten für inhalts-/namenszentrische Ansätze von denen für knotenzentrische Ansätze unterscheidet. Eine weitere Maßnahme sind Strategien zur Optimierung des Speichers der Forwarding Information Base. Hierzu wird zuerst der Aufbau der Forwarding Information Base von CCN-IoT erläutert und anschließend die Optimierungen damit verglichen. Beide Maßnahmen wurden im Rahmen der Arbeit implementiert und evaluiert.

4.3.1 PUFFER FÜR NAMEN UND NACHRICHTEN

Ein Problem bei der Umsetzung des inhaltszentrischen Ansatzes im Internet der Dinge ist die effiziente Darstellung und Kodierung von Namen und Nachrichten. Namen sind Teil von Nachrichten und insbesondere in drahtlosen Ad-hoc-Netzen gilt es, Nachrichten möglichst effizient zu kodieren, da ein IEEE 802.15.4-Rahmen nur 127 Byte aufnehmen kann, wie in den Herausforderungen in Abschnitt 3.2.1 dargestellt. Für die effiziente Kodierung von Namen und Nachrichten wurde in CCN-IoT eine Datenstruktur mit dem Namen *Buffer* (dt. Puffer) entwickelt. Der Buffer ist die Basis für Namen, Nachrichten und Dienstmethoden. Er löst das Problem der variablen Länge von Namen und Nachrichten beim inhaltszentrischen Ansatz und gibt das Nachrichtenformat von CCN-IoT vor, das im folgenden Abschnitt motiviert und einführt wird.

FORMAT FÜR NAMEN UND NACHRICHTEN

In diesem Abschnitt werden die Begriffe *Format*, *Felder fester* und *variabler Länge* von Nachrichten zunächst an einem Beispiel eingeführt. Nachrichten werden in Programmiersprachen üblicherweise als Klassen oder Strukturdatentypen repräsentiert. Quelltext 4.1 zeigt einen Strukturdatentyp in der Programmiersprache C, der eine einfache knotenzentrische Nachricht repräsentiert.

Diese Nachricht ist wie die knotenzentrische Nachricht in Abbildung 3.4(a) auf Seite 46 aufgebaut. Der Header der Nachricht besteht aus vier Variablen: *(i)* Nachrichtentyp (`msg_type`), *(ii)* Zieladresse (`dst_addr`), *(iii)* Quelladresse (`src_addr`), *(iv)* Nutzlastgröße (`payload_size`). Dem Header folgen die Daten (`payload`) der Nachricht. Die ersten vier Felder sind ein Byte (`uint8_t`) beziehungsweise vier Byte (`uint32_t`) groß.

Die Nutzerdaten werden in dem Beispiel in Quelltext 4.1 als Bytearray mit einer im Vorwege definierten Größe (`MAX_PAYLOAD_SZ`) dargestellt. Die Definition

```
1 struct NodeCentricMessage
2 {
3  uint8_t msg_type; /**< Message Type */
4  uint32_t dst_addr; /**< Destination Address */
5  uint32_t src_addr; /**< Source Address */
6  uint8_t payload_size; /**< Current payload size */
7  uint8_t payload[MAX_PAYLOAD_SZ]; /**< Payload/User Data */
8 };
```

Quelltext 4.1 – *Strukturdatentyp für eine einfache knotenzentrische Nachricht in C*

der Größe im Vorwege ist beispielsweise bei eingebetteten Systemen üblich, da diese häufig nicht über die Möglichkeit einer dynamischen Speicherallokation verfügen. Daher wird für jede Nachricht ein Maximalwert für die Größe der Nutzerdaten statisch alloziert. Die tatsächlich genutzte Anzahl an Bytes der Nutzerdaten wird in der Variablen für die Nutzlastgröße gespeichert.

Bevor die Nachricht verschickt wird, wird sie *serialisiert*. Bei der Serialisierung werden die Felder hintereinander – Byte für Byte – in ein Array geschrieben und es kommen nur die tatsächlich genutzten Bytes der Nutzerdaten in die Nachricht. Die Serialisierung ist nötig, da die Treiber für Kommunikationshardware in der Regel nur Bytearrays für die Übertragung akzeptieren.

Abbildung 4.12 zeigt die Struktur der serialisierten Nachricht aus Quelltext 4.1. Die Variablen aus der Datenstruktur in Quelltext 4.1 sind die Felder der Nachricht. Die ersten drei Felder werden dem Header zugeordnet, die Nutzlastgröße und die Nutzlast den Daten. Der Header umfasst die Felder fester Größe, die Daten sind ein Feld variabler Größe.

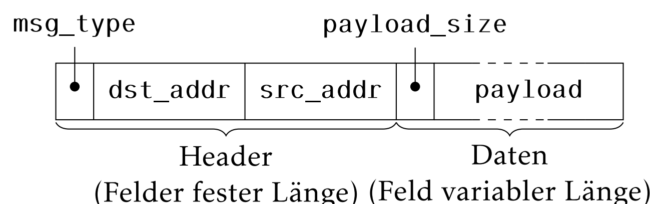


Abbildung 4.12 – *Struktur der serialisierten Nachricht aus Quelltext 4.1*

Nachrichten in inhaltszentrischen Netzen werden ebenfalls serialisiert. Die Repräsentation als Datenstruktur wie in Quelltext 4.1 ist allerdings nicht effizient, wie das folgende Beispiel verdeutlicht. Das Beispiel in Abbildung 4.13 zeigt zwei Nachrichten mit getrenntem Speicherplatz für Namen und Daten; es werden also zwei Arrays pro Nachricht benutzt. Der mit gültigen Daten belegte Speicherplatz ist in hellgrau für den Namen und in dunkelgrau für die Daten hervorgehoben. Der allokierte, aber nicht genutzte Speicherplatz ist farblich nicht hervorgehoben. Die Nachrichten in dem Beispiel enthalten jeweils ein Segment zusammengehörender Daten. Bei beiden Nachrichten wird der maximale Speicherplatz für den Namen und bei der zweiten Nachricht der allokierte maximale Speicherplatz für die Daten nicht voll ausgeschöpft.

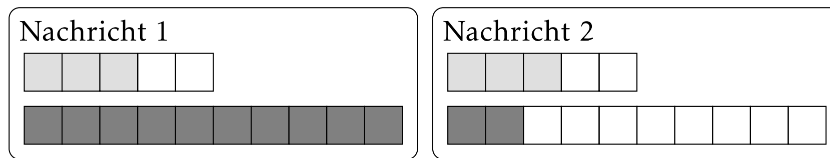


Abbildung 4.13 – *Speicherplatzverbrauch bei getrennter Speicherung von Name und Daten*

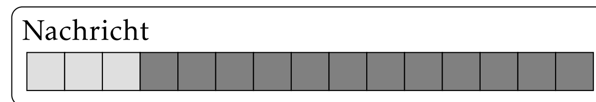


Abbildung 4.14 – *Speicherplatzverbrauch, wenn nur ein Array für Namen und Daten verwendet wird*

Eine bessere Ausnutzung des allokierten Speichers wird erreicht, indem nur ein Array, das den Namen und die Daten aufnimmt, verwendet wird. Abbildung 4.14 zeigt, dass nun die gleiche Menge an Daten in eine Nachricht passt. Der Speicherplatz wird im Gegensatz zu dem Beispiel in Abbildung 4.13 besser ausgenutzt.

CCN-IoT folgt dem Beispiel aus Abbildung 4.14 und serialisiert Namen und Nachrichten in einem Bytearray das Teil der Datenstruktur Buffer ist. Eine Repräsentation als Datenstruktur, wo einzelne Felder wie in Quelltext 4.1 als Variablen repräsentiert werden, gibt es bei CCN-IoT nicht. Alle Felder werden im Array gespeichert und werden über die Methoden des Buffers beziehungsweise der davon abgeleiteten Klassen angesprochen.

Das Bytearray des Buffers gliedert sich in drei logische Abschnitte: Im ersten Abschnitt werden die Felder fester Länge gespeichert. Der zweite Abschnitt enthält die Felder variabler Länge. Am Ende des Buffers markiert der dritte Abschnitt den Rest des allokierten Speicherbereichs, der keine gültigen Daten enthält und in dem sich die Felder variabler Länge ausbreiten. Der ungenutzte Bereich ist stets mit Nullen beschrieben. Abbildung 4.15 zeigt die drei logischen Abschnitte des Buffers. Die Pfeile in Abbildung 4.15 markieren die bereits zur Entwicklungszeit gesetzten festen Positionen der Abschnitte.

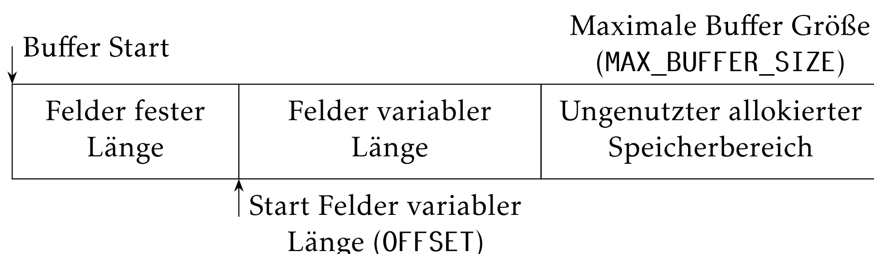


Abbildung 4.15 – *Aufbau des Buffers*

Felder fester Länge enthalten Informationen wie Ziel- und Quell-Adresse der Sicherungsschicht sowie den Nachrichtentyp. Die Implementierung von CCN-IoT

benutzt auf der Sicherungsschicht ein proprietäres Format, welches im Rahmen mehr Platz für Daten oberhalb der Sicherungsschicht lässt. Abbildung 4.16 zeigt den Anfang eines Rahmens mit den Feldern fester Länge.

Position:	0	1	2	3	4
Buffer:	0xFF	0xFF	0x00	0x64	0x1A

Abbildung 4.16 – *Beispiel für Felder fester Länge*

Die Positionen der einzelnen Felder fester Länge sind bekannt, da sie an festgelegten Indizes stehen. In Beispiel in Abbildung 4.16 ist die Zieladresse 0xFFFF (Broadcast-Adresse) an der Position Null. Die Quelladresse ist 0x0064 beginnt in diesem Beispiel an Position zwei. Der Nachrichtentyp ist 0x1A und steht an Position vier.

Felder variabler Länge werden nicht über einen Index, sondern über das *Typ-Längen-Feld* (engl. Type-Length-Value, kurz TLV) [93, 64] identifiziert, welches am Anfang eines Feldes variabler Länge steht. Der Typ-Teil des Typ-Längen-Feldes weist den darauf folgenden Datenbytes eine Semantik zu und der Längen-Teil gibt die Anzahl der darauf folgenden Datenbytes an. CCN-IoT verwendet für ein Typ-Längen-Feld nur ein Byte. Abbildung 4.17 zeigt ein Beispiel für Felder variabler Länge.

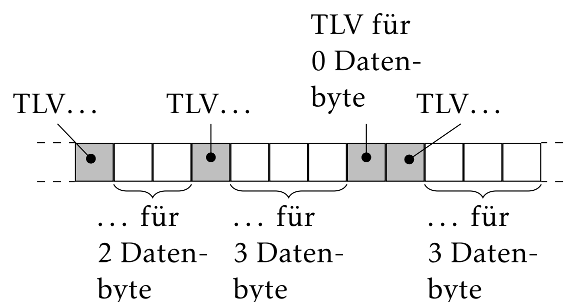


Abbildung 4.17 – *Felder variabler Länge*

In der Datenstruktur Buffer werden für die Typ-Längen-Felder vier Typen definiert: *(i)* Flag, *(ii)* Ganzzahltyp (Integer), *(iii)* Array und *(iv)* Namenskomponente. Das ein Byte große Typ-Längen-Feld wird in drei höherwertige und fünf niederwertige Bits geteilt. Für Typ-Längen-Felder des Buffers gilt: *(i)* Bei Flags sind die höherwertigen drei Bits immer Null, bei Ganzzahltypen im Bereich von 001_2 bis 100_2; die höherwertigen drei Bits kodieren die Anzahl der nach dem Typ-Längen-Feld folgenden Bytes und die niederwertigen fünf Bits kodieren den Typ der nach dem Typ-Längen-Feld folgt. *(ii)* Bei Arrays und Namenskomponenten sind die höherwertigen drei Bits im Bereich 101_2 bis 110_2; die höherwertigen drei Bits kodieren den Typ der folgenden Bytes; die niederwertigen fünf Bits kodieren die Anzahl der folgenden Bytes.

Bei Flags beginnt der Wertebereich für Typen (niederwertige fünf Bits) bei Eins und nicht bei Null, wie bei Ganzzahltypen, da Typ-Längen-Felder die gleich

Null sind, das Ende des genutzten Buffers markieren. Die Kodierung von Flags und Ganzzahltypen ermöglicht somit insgesamt 30 beziehungsweise 31 Typen. Flags und Ganzzahltypen spannen jeweils einen getrennten *Gültigkeitsbereich* auf, da es keine Ganzzahltypen der Länge Null gibt. Der Wertebereich für Flags und Ganzzahltypen überschneidet sich also nicht und es wird der gleiche Wertebereich für Typen in einem unterschiedlichen Kontext verwendet. Gleiches gilt für die beiden Nachrichtentypen Interest und Content Object, die jeweils einen Gültigkeitsbereich aufspannen. Felder variabler Länge in einer Interest Nachricht benutzen denselben Wertebereich für Typen wie Felder in einer Content Object Nachricht. Die Unterscheidung erfolgt hier anhand des Kontext – Interest oder Content Object. Daraus folgt, dass praktisch 60 Typen für Flags und 62 Typen für Ganzzahltypen zur Verfügung stehen.

Für Arrays ergibt sich eine minimale Länge von Null Byte und eine maximale Länge von 31 Byte, da die niederwertigen fünf Bits die Länge der folgenden Daten kodieren. Im Gegensatz zu Arrays haben Namenskomponenten immer ein Element und daher kodieren die niederwertigen fünf Bits die Länge plus Eins. Namenskomponenten haben damit eine maximale Länge von 32 Byte. Die maximalen Längen von 31/32 Byte erscheinen auf den ersten Blick gering, reichen aber in der Praxis für ressourcenbeschränkte Geräte aus, da Namen im drahtlosen Netz ohnehin kurz gehalten beziehungsweise verkürzt werden.

Der Zugriff auf Felder variabler Länge ist nur sequentiell und nicht wahlfrei, wie bei den Feldern fester Länge, möglich. Felder variabler Länge nutzen dafür den Platz im Buffer beziehungsweise der Nachricht besser aus. Mit Feldern variabler Länge können darüber hinaus optionale Felder implementiert werden. Ebenso ist es nicht zwingend notwendig, dass die Typen der Felder variabler Länge einem Empfänger einer Nachricht bekannt sind, um die Nachricht zu verarbeiten. Unbekannte Felder variabler Länge werden bei der Verarbeitung einfach übersprungen. Das Überspringen von unbekannten Feldern variabler Länge ermöglicht damit Aufwärtskompatiblität zu zukünftigen Protokollerweiterungen. Für das Internet der Dinge mit seinen heterogenen Geräten hat das Überspringen unbekannter Typen noch einen weiteren Vorteil: In Abhängigkeit ihrer Ressourcen implementieren Geräte nur die jeweils benötigten Merkmale des Protokolls und die damit assoziierten Typen. Obwohl es Geräte im Netz gibt, die nicht alle Merkmale und Typen implementieren, sind trotzdem alle Geräte in der Lage, die Nachrichten zu verarbeiten.

Die Kodierung der Typ-Längen-Felder, ein Beispiel für ein Feld variabler Länge und der Zugriff auf Felder variabler Länge werden im Anhang in Abschnitt A.1.1 detailliert dargestellt. Da nun das Nachrichtenformat bekannt ist, wird im nächsten Abschnitt die Architektur der Datenstruktur Buffer sowie der davon abgeleiteten Datenstrukturen für Namen und Nachrichten vorgestellt.

ARCHITEKTUR VON NAMEN UND NACHRICHTEN

Im vorangegangenen Abschnitt wurde erläutert, dass die Repräsentation von inhaltszentrischen Nachrichten mit einem Bytearray effizient ist. Die Datenstruktur Buffer nutzt aus diesem Grunde nur einen Bytearray, in welchem der die gesamte Nachricht in serialisierter Form gespeichert ist. Die Datenstruktur

Buffer ist als Klasse implementiert und die Klassen für Namen und Nachrichten werden von der Klasse `Buffer` abgeleitet. Für die Darstellung der Architektur wird im Folgenden ein UML-Klassendiagramm verwendet. Das UML-Klassendiagramm in Abbildung 4.18 zeigt die Architektur der Buffer-Implementierungen mit der abstrakten Basisklasse `Buffer` und die davon abgeleiteten Klassen sowie die Vererbungsbeziehungen zwischen den Klassen. Die Abgeleiteten Klassen *(i)* `Name`, *(ii)* `Method`, *(iii)* `ServiceName`, *(iv)* `Type`, *(v)* `Interest` und *(vi)* `ContentObject` sind Implementierungen der Klasse `Buffer` und erweitern diese um Funktionalität.

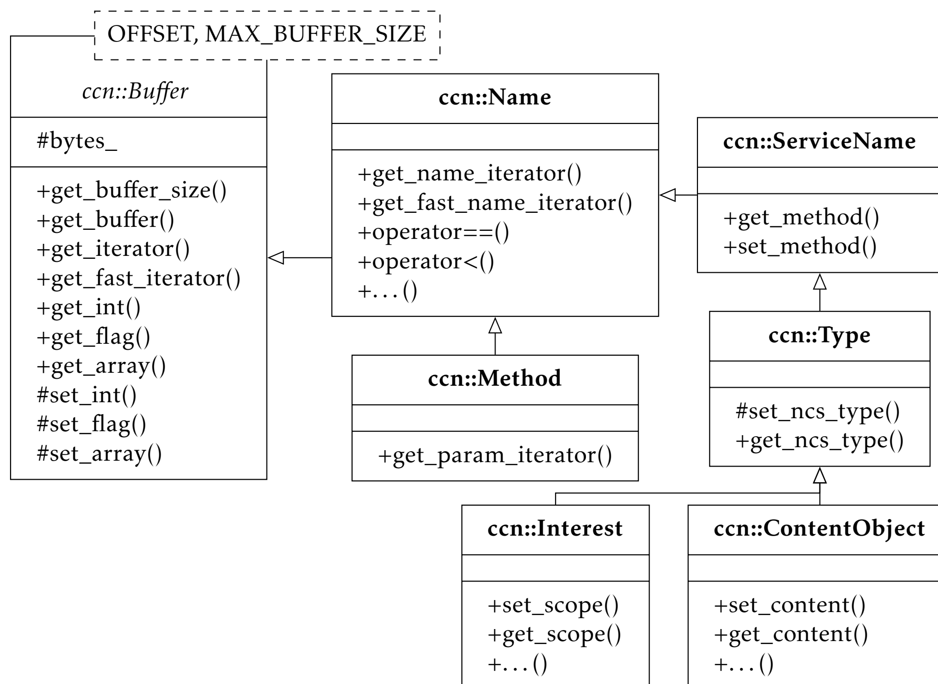


ABBILDUNG 4.18 – *Klassenhierarchie der Buffer-Implementierungen*

Die Klasse Buffer hat ein Bytearray (`bytes_`), welches den serialisierten Namen oder die serialisierte Nachricht aufnimmt. Das Bytearray ist die einzige Klassenvariable von Buffer. Darüber hinaus hat sie zwei Template-Parameter `MAX_BUFFER_SIZE` und `OFFSET`. Mit dem ersten Template-Parameter wird die Größe des Bytearrays festgelegt. Der zweite Template-Parameter gibt den Offset an, ab dem Felder variabler Länge im Buffer stehen. Hier zeigt sich der Vorteil von C++-Templates, denn die Positionen sind keine Pointer und verbrauchen somit keinen Speicherplatz.

Alle von der Klasse `Buffer` abgeleiteten Klassen haben keine Variablen. Die gesamte Information speichern diese in dem Bytearray der gemeinsamen Basisklasse Buffer. Für abgeleitete Klassen wird mit dem Template-Parameter `MAX_BUFFER_SIZE` die Größe des Bytearrays der Basisklasse festgelegt. Damit wird gleichzeitig die jeweilige Größe der von Buffer abgeleiteten Klasse festgelegt. Ein Interest nimmt für den `MAX_BUFFER_SIZE` einen kleineren Wert an als ein Content Object, somit ist ein Interest kleiner als ein Content Object. Die von

Buffer abgeleiteten Klassen werden im Folgenden kurz erklärt; Implementierungsdetails finden sich im Anhang in Abschnitt A.1.2.

Die Klasse Name repräsentiert einen CCN-Namen (vgl. Definition 2.8). Von der Klasse Name ist ServiceName abgeleitet. ServiceName implementiert sogenannte Service Namen (dt. Dienstnamen) und erweitert Name um Funktionalität, die für die namenszentrischen Dienste benötigt wird. Zu dieser Funktionalität zählt das Setzen und Auffinden der Namenskomponenten die Methoden und Segmentmarker kodieren. Die Klasse Interest ist wird wie alle CCN-IoT-Nachrichten von der Klasse Type abgeleitet. Klasse Type implementiert die Funktionalität, um auf den Nachrichtentyp zuzugreifen. Der Nachrichtentyp befindet sich bei CCN-IoT-Nachrichten als Feld fester Länge vor dem Namen. Die Klasse Content Object hat einen ähnlichen Aufbau wie Interest und implementiert zusätzlich Funktionalität, um Content hinzuzufügen. Die Klasse Method repräsentiert die Methode in CCN-IoT. Methoden werden für die Übertragung serialisiert und die Architektur von CCN-IoT sieht vor, dass serialisierbare Objekte von der Klasse Buffer abgeleitet werden. Da Methoden einen Namen haben, werden sie konsequenterweise von der Klasse Name abgeleitet. Durch die Ableitung von Name werden die Methoden zum Vergleich und Zugriff von Komponenten wiederverwendet. Die Klasse Method implementiert Funktionalität, um an den Namen der serialisierten Methode die Parameter anzuhängen.

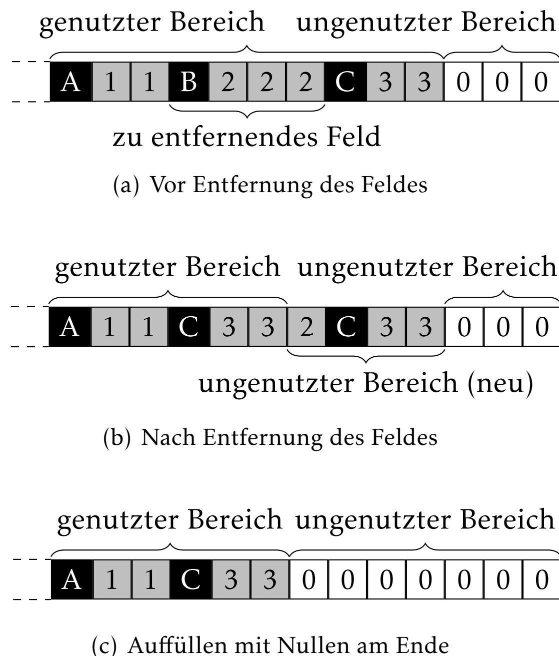
Die Klasse Buffer und die davon abgeleiteten Klassen haben Methoden, die Iteratoren zurückliefern. Iteratoren ermöglichen den Zugriff auf die im Bytearray serialisierten Elemente und helfen bei der effizienten Verarbeitung von Namen und Nachrichten, wie der folgende Abschnitt zeigt.

4.3.2 NACHRICHTENVERARBEITUNG

Die Verarbeitung von Nachrichten und Namen ist wichtig bei der Anwendung von namenszentrischen Diensten. So werden bei den Aktivitäten bei der Entwicklung von Diensten Anfragen Komponentenweise zusammengesetzt (vgl. Aktivitäten bei der Entwicklung von Diensten in Abbildung 3.2 auf Seite 40). Namenszentrische Dienste für Kommunikation und Routing modifizieren den Namen in Nachrichten, um die Weiterleitung zu beeinflussen.

Die Verarbeitung von Namen und Nachrichten umfasst die drei Operationen *(i) Einfügen*, *(ii) Löschen* und *(iii) Modifizieren* von Feldern variabler Länge sowie Komponenten in Namen und Nachrichten. Ein Problem ist, dass bei Einfüge- oder Löschoperationen die Länge des genutzten Bereichs des Buffers ermittelt werden muss (vgl. Aufbau des Buffers in Abbildung 4.15). Die Größe, oder auch Länge in Bytes, des tatsächlich genutzten Speicherbereichs ergibt sich aus der Startposition der Felder variabler Länge, gegeben durch den Template-Parameter OFFSET, und der Gesamtlänge der Felder variabler Länge.

Beim Verschieben der Daten, insbesondere wenn der genutzte Datenbereich gekürzt wird, wird die Differenz am Ende des Buffers wieder mit Nullen aufgefüllt, um das Ende des tatsächlich genutzten Speicherbereichs zu markieren. Abbildung 4.19 zeigt den Prozess des Verkürzens des Buffers an einem Beispiel.



(a) Vor Entfernung des Feldes

(b) Nach Entfernung des Feldes

(c) Auffüllen mit Nullen am Ende

Abbildung 4.19 – *Verkürzen eines Buffers*

In der Abbildung sind die Typ-Längen-Felder farblich abgesetzt und mit Großbuchstaben gekennzeichnet, welche in dem Beispiel exemplarisch für den Typ stehen. Die Daten der Felder variabler Länge sind mit Ziffern größer Null gekennzeichnet und ebenso farblich hervorgehoben. Zusammenhängende Daten haben die gleiche Ziffer. Der ungenutzte Speicherbereich am Ende des Buffers ist mit Nullen gekennzeichnet. Jedes Kästchen in dem Beispiel in Abbildung 4.19 entspricht einem Byte.

Die Gesamtlänge des genutzten Speicherbereichs in Abbildung 4.19(a) beträgt zehn Byte. Bei der Entfernung von Feld B in Abbildung 4.19(b) wird das benachbarte Feld C an den Anfang von Feld B kopiert. Die Länge des genutzten Speicherbereichs verkürzt sich um vier Byte, also ist der neue genutzte Speicherbereich nur noch sechs Byte lang. Der neue ungenutzte Bereich, von neuer zu alter Länge des genutzten Speicherbereichs, wird in Abbildung 4.19(c) wieder mit Nullen überschrieben.

Ein weiteres Problem bei der Verarbeitung ist, sicherzustellen, dass das Nachrichtenformat nicht durch unsachgemäßen Zugriff auf das Array des Buffers kompromittiert wird. Namenskomponenten stehen bei dem Nachrichtenformat von CCN-IoT stets zusammenhängend am Anfang der Felder variabler Länge. Komponenten mit Methoden stehen am Ende des Namens und Parameter von Methoden stehen hinter dem Methodennamen und weitere Felder variabler Länge, wie Steuerungsfelder und Content, stehen hinter dem Namen. Abbildung 4.20 fasst den Aufbau des Nachrichtenformats von CCN-IoT zusammen.

In dieser Arbeit wird den zwei Problemen mit *Iteratoren* begegnet, durch die der Zugriff auf Felder variabler Länge und damit auch auf Komponenten von

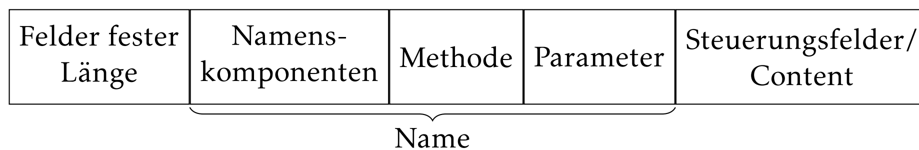


Abbildung 4.20 – *Nachrichtenformat von CCN-IoT*

Namen erfolgt. Iteratoren ermitteln einmal die Länge des Buffers und halten diese bei Editieroperationen konsistent. Damit die Länge eines Buffers nicht immer von neuem ermittelt werden muss, werden aufeinanderfolgende Editieroperationen mit dem anfangs akquirierten Iterator durchgeführt, weil der Iterator die Länge des genutzten Speicherbereichs konsistent hält. Somit sind Änderungen am Buffer effizient möglich. Der von der Werkzeugunterstützung generierte Code akquiriert am Anfang einer Sequenz von Editieroperationen einen Iterator, mit dem dann beispielsweise eine Anfrage zusammengestellt wird. Mit den Iteratoren ist es daher nicht nötig, die Länge des Buffers oder die Anzahl der Namenskomponenten mit im Buffer zu speichern, was wiederum Speicher einspart.

Iteratoren kapseln außerdem den Zugriff auf das Array des Buffers, um dieses vor Änderungen zu schützen, die das Nachrichtenformat kompromittieren. Iteratoren sorgen zum Beispiel dafür, dass die Namenskomponenten stets zusammenhängend bleiben, indem sie das Einfügen, Löschen und Modifizieren nur am Beginn der Felder variabler Länge und zwischen Namenskomponenten erlauben.

Das Ermitteln der Länge des Buffers kostet Zeit. Erfolgt nur ein lesender Zugriff auf den Buffer, ist die Länge des genutzten Speicherbereichs nicht wichtig. Aus diesem Grunde sieht der CCN-IoT-Buffer zwei Typen von Iteratoren vor: einen *(i)* schnellen, nur lesenden und einen *(ii)* langsamen, modifizierenden (und lesenden) Iterator. Dass sich die Aufteilung der Iteratoren in langsam und schnell durchaus lohnt, wird im nächsten Abschnitt bei der Evaluation der Nachrichtenverarbeitung gezeigt. Zu Implementierungsdetails der Iteratoren sei auf den Anhang, Abschnitt A.1.3, verwiesen.

4.3.3 EVALUATION NACHRICHTENVERARBEITUNG

In diesem Abschnitt werden die Buffer-Implementierungen evaluiert. Alle folgenden Experimente wurden auf der an der Fachhochschule Lübeck entwickelten TriSOS-Sensorknoten-Plattform durchgeführt. Ein TriSOS-Knoten ist mit einem Atmel AVR 8-Bit ATxmega128 Mikrocontroller ausgestattet, der mit einer Taktfrequenz von 32 MHz betrieben wird und über 64 kB Arbeitsspeicher verfügt. Alle Programme wurden mit der Atmel AVR 8-bit Tool Chain in der Version 3.4.4, die die GNU Compiler Collection in der Version 4.8.1 enthält, übersetzt. Die Übersetzung wurde mit einer Optimierung auf Codegröße durchgeführt.

Alle im Folgenden gezeigten Experimente und deren Ergebnisse wurden in der Veröffentlichung „Architecture and Message Processing for Name-Centric

Services in Wireless Sensor Networks“ (Teubler, Hellbrück) [3] vorgestellt. In einem ersten Experiment wird gezeigt, dass Buffer-Implementierungen unterschiedlicher Länge auch unterschiedlich schnell kopiert werden, was sich auf die Verarbeitungsgeschwindigkeit auswirkt. Es gibt insgesamt fünf Buffer-Implementierungen, die unterschiedlich groß sind.

Die Länge des Bytearrays wird über einen Template-Parameter gesteuert (vgl. Klassenhierarchie der Buffer-Implementierungen in Abbildung 4.18 auf Seite 81). Die Evaluation wird mit unterschiedlichen Längen für das Bytearray durchgeführt. Tabelle 4.1 zeigt die Größe der Buffer und die Laufzeit, die zum Kopieren der Buffer benötigt wird. Um die Messungen statistisch Abzusichern wurden die einzelnen Messungen 50-mal durchgeführt und der Mittelwert über alle Messungen gebildet. Die Kopien wurden byteweise mit dem automatisch generiertem C++-Zuweisungsoperator ausgeführt.

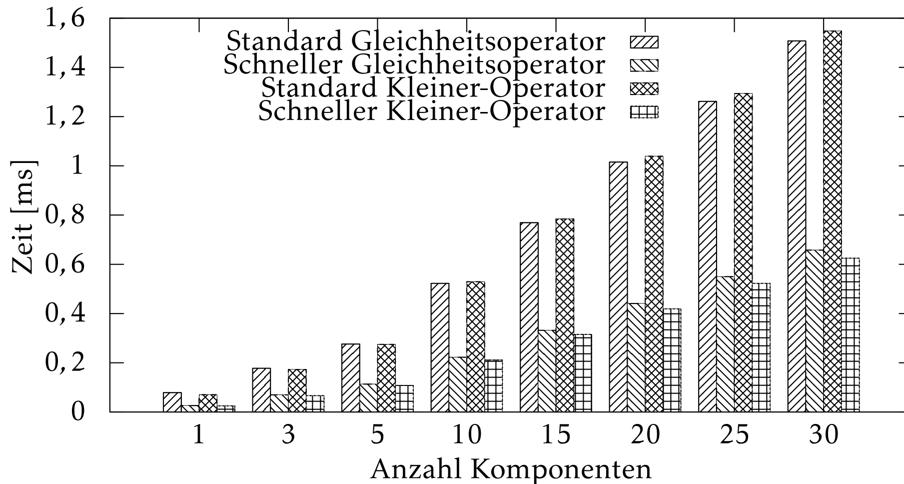
Buffer	Größe [Byte]	Laufzeit [µs]
`Method`	32	6,2
`Name/ServiceName`	60	11,8
`Interest`	70	13,3
`ContentObject`	121	22,9

Tabelle 4.1 – *Laufzeit für das Kopieren von Buffern unterschiedlicher Größe*

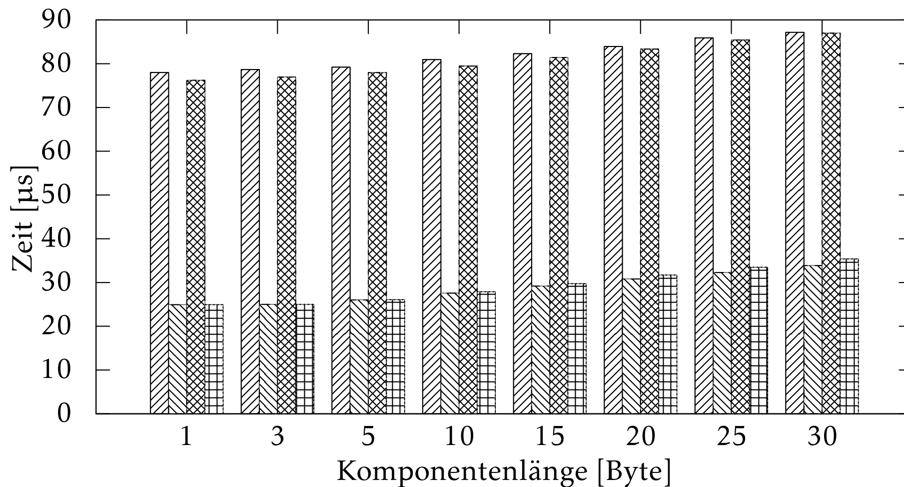
Die Größe der Methode ist auf 32 Byte limitiert, damit sie in eine Komponente passt, die Größen von Content Object und Interest sind heuristisch gewählt. Ein Interest, wird beispielsweise dreimal während der Verarbeitung im Dämon kopiert. Hätte der Interest dieselbe Größe wie das Content Object, würde jede Kopieraktion etwa doppelt so viel Zeit in Anspruch nehmen.

In einem zweiten Experiment wird die Leistung der Vergleichsoperatoren für Namen bewertet. Die Vergleichsoperatoren werden für die Evaluation der Verarbeitung deswegen genommen, weil Namen in CCN-IoT relativ häufig während des normalen Betriebs miteinander verglichen werden. Beispielsweise, wenn ein Interest an einem Knoten empfangen wird, wird der Name mit den Einträgen in der Forwarding Information Base (FIB) verglichen, um das Face zum Weiterleiten des Interests zu bestimmen. Weiterhin wird beim Einfügen neuer FIB-Einträge eine Einfügesortierung verwendet, wo der einzufügende FIB-Eintrag mit den existierenden FIB-Einträgen verglichen wird. Neben diesen zwei Beispielen gibt es noch viele weitere Stellen, wo Namen verglichen werden. Beide Vergleichsoperatoren, Gleichheitsoperator (`operator==()`) und Kleiner-Operator (`operator<()`), sind mit Iteratoren implementiert. Die Vergleichsoperatoren wurden für die Evaluation einmal unter Verwendung des Iterators für den schnellen, nur lesenden Zugriff und des Iterators für den (langsameren) modifizierenden Zugriff implementiert. Im Folgenden werden Operatoren, die mit dem modifizierenden (langsamen) Iterator implementiert wurden, als *standard* Operator und mit dem nur lesenden Iterator als *schneller* Operator bezeichnet. Ziel dieses Experiments ist es zu zeigen, dass die zwei Typen von Iteratoren (modifizierend und langsam, nur lesend und schnell) unterschiedlich leistungsfähig bei den Vergleichsoperationen sind und daher *(i)* zwei Typen von Iteratoren sinnvoll

sind und man bei *(ii)* Vergleichsoperatoren immer den schnellen, lesenden Iterator nehmen sollte. Bei den Vergleichen werden nur identische Namen verglichen, da hier die Namen Byte für Byte verglichen werden. Der Vergleich von identischen Namen stellt damit das Worst-Case-Szenario für die Vergleichsoperatoren dar.



(a) Unterschiedliche Komponentenanzahl, Komponenten jeweils ein Byte lang



(b) Eine Komponente, unterschiedliche Komponentenlänge

Abbildung 4.21 – *Leistungsbewertung der Vergleichsoperatoren*

Die Ergebnisse der Evaluation der Ausführungszeit der unterschiedlich implementierten Iteratoren sind in Abbildung 4.21 dargestellt. Mit steigender Ausführungszeit sinkt die Verarbeitungsgeschwindigkeit der Iteratoren. In dem ersten Teilexperiment der Evaluation wird die Anzahl der Komponenten erhöht, wobei jede Komponente nur ein Byte groß ist (z. B. /a, /a/a, /a/a/a, ...). Die Ergebnisse des ersten Teilexperiments zeigt Abbildung 4.21(a). Auf der y-Achse ist die Ausführungszeit eines Vergleichs in Millisekunden angegeben, auf der

x-Achse die Anzahl der Komponenten. Mit steigender Anzahl der Komponenten steigt auch die Ausführungszeit an. In dem zweiten Teilexperiment der Evaluation hat der Name nur eine Komponente, deren Länge schrittweise vergrößert wird (z. B. /a, /aa, /aaa, ...). Die Ergebnisse des zweiten Teilexperiments zeigt Abbildung 4.21(b). Auf der y-Achse ist die Ausführungszeit in Mikrosekunden und auf der x-Achse die Komponentenlänge aufgetragen.

Jede Operator-Implementierung, mit dem schnellen, nur lesenden oder dem modifizierenden Zugriff, ist funktional. Es gibt allerdings einen signifikanten Unterschied in der Ausführungszeit der Implementierungen. Der Kleiner-Operator mit der nur lesenden Iterator-Implementierung (*schneller Kleiner-Operator*) ist nahezu dreimal so schnell wie der modifizierende Iterator (*standard Kleiner-Operator*). Der Grund für dieses Verhalten ist, dass der modifizierende Iterator die Buffer-Länge ermittelt. Bei der Operator-Implementierung mit dem modifizierenden Iterator wird die Länge von dem linken und dem rechten Operanden jeweils in einer Schleife ermittelt, was für einen Vergleich der Operanden unnötig ist. Anschließend werden beide Buffer in einer weiteren Schleife durchlaufen und Byte für Byte verglichen. Die Implementierung mit den schnellen Iteratoren vermeiden die ersten beiden Schleifendurchläufe und führen gleich den byteweisen Vergleich aus.

Ein Vergleich von Abbildung 4.21(a) mit Abbildung 4.21(b) zeigt, dass die Anzahl der Komponenten (allgemeiner, die Felder variabler Länge) einen stärkeren Einfluss auf die Ausführungszeit und damit auf Verarbeitungsgeschwindigkeit hat, als die Länge des Feldes variabler Länge. Der leichte Anstieg in der Ausführungszeit in Abbildung 4.21(b) bei wachsender Komponentenlänge wird durch die C-Standardbibliotheksfunktion `memcmp()` verursacht, mit der die Komponenten verglichen werden.

Die Vergleichsoperation mit dem schnellen Iterator in Abbildung 4.21(b) braucht 0,0025 ms. Multipliziert man 0,0025 ms mit 30, so erhält man 0,7 ms. Dieser Wert von 0,7 ms korrespondiert mit dem des Namens mit 30 Komponenten in Abbildung 4.21(a) mit dem schnellen Iterator. Der Anstieg der Ausführungszeit in Abbildung 4.21(a) ist ein Ergebnis des wiederholten Aufrufs von `memcmp()`. Ist die Länge des Namens bekannt, ist der Vergleich nur mit einem Aufruf von `memcmp()` möglich. Die Bestimmung der Namenslänge ist genau das Verhalten des modifizierenden (langsamen) Iterators. Im Endeffekt braucht der modifizierende Iterator zwei Durchläufe, während der schnelle Iterator nur einen braucht. Eine Möglichkeit, den Vergleich weiter zu beschleunigen wäre, die Gesamtlänge des Namens mit dem Namen zu speichern. Die Speicherung der Gesamtlänge ist eine redundante Information und benötigt zusätzlichen Speicher für jeden Buffer, der auf einem (ressourcenbeschränkten) Gerät gespeichert wird.

In der Praxis werden die Vergleichsoperatoren selbstverständlich mit dem schnellen Iterator implementiert, da beim Vergleich keine Modifikationen vorgenommen werden. Der Vergleich der Implementierungen in Abbildung 4.21 zeigt, das die Trennung der Funktionalitäten einen effizienten und bedarfsgerechten Einsatz der zwei Iteratortypen erlaubt.

Die Buffer-Implementierungen für Namen und Nachrichten sind CCN-IoT-spezifisch. CCN-IoT verfügt als Implementierung von CCN auch über die Daten-

strukturen Content Store, Pending Interest Table und Forwarding Information Base. Die CCN-IoT-Implementierungen der CCN-Datenstrukturen werden im folgenden Abschnitt eingeführt.

4.3.4 CCN-DATENSTRUKTUREN

Ein weiterer Teil der Implementierung von CCN-IoT sind die CCN-Datenstrukturen Content Store (CS), Pending Interest Table (PIT) und Forwarding Information Base (FIB), die in Abschnitt 2.4 eingeführt wurden. Grundlage für die Implementierung der CCN-Datenstrukturen in CCN-IoT bilden die generischen Datenstrukturen *Set* und *Vec*. Set speichert die Elemente sortiert, wobei die Elemente sortiert eingefügt werden (Einfügesortierung). In Set ist die Speicherung zweier identischer Elemente nicht möglich. Vec ist wie ein Array aufgebaut und unterstützt einen wahlfreien lesenden Zugriff auf die Elemente und das wahlfreie Entfernen von Elementen sowie das Einfügen von Elementen am Anfang und am Ende. Die Implementierung des CS und der FIB nutzen als Grundlage die Datenstruktur Set, da die Elemente stets über ihren eindeutigen Namen identifiziert werden. Ebenso ist eine Sortierung der Elemente (nach der Shortlex-Ordnung) notwendig, wie in den Grundlagen in Abschnitt 2.4 beschrieben.

Das sortierte Einfügen in Set kostet Ressourcen und da bei der PIT eine sortierte Speicherung der Interests nicht notwendig ist, wird hier die Datenstruktur Vec verwendet. Andere, in der Literatur diskutierte Implementierungen der PIT, speichern keine Interests, sondern nur die Namen der Interests in Form von Hashes oder Bloom-Filtern, wie beispielsweise in „DiPIT: A Distributed Bloom-Filter based PIT Table for CCN Nodes“ von Wei You et al. [176].

In CCN-IoT hingegen dient die PIT auch als Zwischenspeicher für Interests, die an andere Faces weitergeleitet werden. Bei der Weiterleitung wird der Interest in der PIT gespeichert und nach einem kurzen, zufälligen Zeitraum an die entsprechenden Faces weitergeleitet. Das Zwischenspeichern und verzögerte Versenden der Interests hat den Vorteil, dass das Senden von Interests und der darauffolgende Empfang von Content Objects stets asynchron erfolgt. Ein weiterer Vorteil des asynchronen Sendens ist, dass bei der Weiterleitung der Interests über die Funkschnittstelle ein Jitter eingefügt wird, was die Wahrscheinlichkeit von Kollisionen beim Fluten von Interests verringert.

Implementierungsdetails der CCN-Datenstrukturen in CCN-IoT sind für die Arbeit nebensächlich, daher werden diese im Anhang in Abschnitt A.1.4 behandelt. Wichtig zu erwähnen ist, dass es Bestrebungen gibt, die CCN-Datenstrukturen – mit dem Hinblick auf das Internet der Dinge – zu optimieren. Unter diese Bestrebungen fallen beispielsweise die oben genannte Arbeit von Wei You et al. [176] als auch die Arbeit „I (FIB) F: Iterated Bloom Filters for Routing in Named Data Networks“ von Muñoz et al. [115]. Im Rahmen der Arbeit sind ebenfalls Strategien zur Speicheroptimierung der FIB entstanden, die veröffentlicht wurden und im folgenden Abschnitt vorgestellt werden.

4.3.5 SPEICHEROPTIMIERUNG DER FIB

Dieser Abschnitt stellt Strategien zur Speicheroptimierung der Forwarding Information Base vor, die in „Memory Efficient Forwarding Information Base for Content-Centric Networking“ Teubler et al. [1] veröffentlicht wurden. Die Optimierungsstrategien nutzen *Hashing* und *Bloom-Filter*. Der Ansatz mit Hashes wird im Folgenden als *FIB-Hash*, der Ansatz mit Bloom-Filtern als *FIB-BF* bezeichnet.

FIB-HASH

Hashing ist eine Möglichkeit, eine Menge $S = \{s_1, \ldots, s_n\}$ zu repräsentieren. Eine Hash-Funktion h bildet jedes Element von S auf eine Zahl fester Größe mit m Bit ab: $h : s_i \rightarrow \{0, \ldots, 2^m\}$. Eine wichtige Eigenschaft von h ist, dass unterschiedliche Elemente auf unterschiedliche Zahlen abgebildet werden. Bei FIB-Hash werden die Präfixe s_i in der FIB (vgl. Aufbau der FIB in Abbildung 2.8(c) auf Seite 29) durch die entsprechende Hash-Werte $h(s_i)$ ersetzt. Beim Empfang eines Interests werden nacheinander die Hash-Werte aller Präfixe des Interest-Namens berechnet und die Präfixe mit absteigender Länge mit den Hashes in allen FIB-Einträgen auf Gleichheit geprüft. Ist eine Prüfung auf Gleichheit erfolgreich, dann handelt es sich um den längsten gemeinsamen Präfix von Interest-Namen und FIB-Eintrag. Abbildung 4.22 zeigt die Verarbeitung des Interest-Namens bei FIB-Hash in einem Beispiel: Im *1.* Schritt werden die Hashes der Präfixe berechnet. In den ersten zwei Runden gibt es keinen Treffer bei den FIB-Einträgen, in der dritten Runde gibt es mit dem Hash-Wert 0x9c4e einen Treffer, da er dem ersten Hash in den FIB-Einträgen entspricht. Im *2.* Schritt wird der Interest an die entsprechenden Faces des ermittelten FIB-Eintrages weitergereicht.

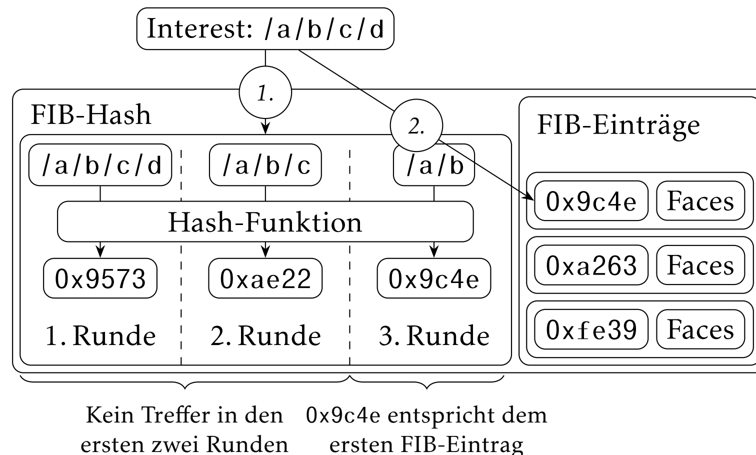


ABBILDUNG 4.22 – *Interest-Verarbeitung in FIB-Hash*

FIB-BF

Ein Bloom-Filter, beschrieben von Burton H. Bloom in „Space/Time Trade-offs in Hash Coding with Allowable Errors“ [48], ist eine Datenstruktur fester Größe, welche Elemente einer Menge repräsentiert. Um festzustellen, ob ein Element in der Menge ist, wird das Element gegen den Bloom-Filter geprüft.

Die Prüfung ergibt entweder, dass das Element sicher nicht in der Menge oder, mit einer bestimmten Falsch-Positiv-Wahrscheinlichkeit P_{FP}, vielleicht in der Menge ist. Aufgrund der Konstruktion von Bloom-Filtern ist ist es nicht möglich, hinzugefügte Elemente aus Bloom-Filtern zu entfernen. Mit den zählenden Bloom-Filtern gibt es eine Erweiterung von Bloom-Filtern, die das Entfernen von Elementen unterstützt. Allerdings beanspruchen zählende Bloom-Filter mehr Platz/Speicher als normale Bloom-Filter. Die FIB-Implementierung mit zählenden Bloom-Filtern wird im Folgenden *FIB-CBF* (engl. Counting Bloom Filter) genannt.

Bei FIB-BF ist jedem Face ein Bloom-Filter zugeordnet. Ein Bloom-Filter enthält die Präfixe, die dem jeweiligen Face zugeordnet sind. Abbildung 4.23 zeigt den Aufbau von FIB-BF.

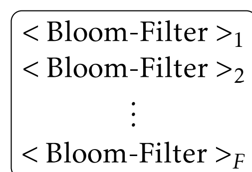


ABBILDUNG 4.23 – *Aufbau von FIB-BF*

Die Interest-Verarbeitung ist ähnlich zu der bei FIB-Hash. Für jeden Präfix – von lang nach kurz – wird geprüft, ob er in einem der F Bloom-Filter ist. Im Gegensatz zu FIB-Hash, wo bei einem erfolgreichen Vergleich gestoppt wird, wird der Präfix gegen alle Bloom-Filter geprüft, da es möglicherweise mehrere Faces gibt, denen der selbe Präfix zugeordnet ist.

Die Struktur von FIB-BF ist einfacher als die von Muñoz et al. [115] vorgeschlagene Implementierung der FIB. Muñoz et al. modifizieren zusätzlich die Nachrichten, so werden die Namen in den Nachrichten ebenfalls durch Bloom-Filter ersetzt. FIB-BF betrachtet die Funktion der FIB isoliert und ist unabhängig von tiefgreifenden Änderungen am System. Im folgenden Abschnitt werden unter anderem FIB-Hash, FIB-BF und FIB-CBF miteinander verglichen und Rahmenbedingungen identifiziert, für die die jeweiligen Lösungen am besten sind.

4.3.6 EVALUATION SPEICHEROPTIMIERUNG DER FIB

In diesem Abschnitt werden die Strategien zur Speicheroptimierung der FIB mittels theoretischen Betrachtungen evaluiert. Bei der Evaluation werden FIB-Hash, FIB-BF, FIB-CBF und die Implementierung der FIB von CCN-IoT miteinander verglichen. Für das bessere Verständnis des Abschnitts werden zunächst benötigte Grundlagen und Definitionen eingeführt.

GRUNDLAGEN

Die FIB ist, theoretisch betrachtet, eine Relation zwischen Präfix und Face, wie in dem Entity-Relationship-Diagramm in Abbildung 4.24 dargestellt. Mit N werden im Folgenden die Anzahl der Präfixe und mit F die Anzahl der Faces

bezeichnet. Aus Abbildung 4.24 geht hervor, dass jedem Präfix mindestens ein Face zugeordnet ist. Im Allgemeinen ist es jedoch nicht erforderlich, dass einem Face ein Präfix zugeordnet ist.

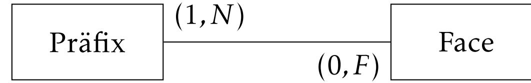


Abbildung 4.24 – *Entity-Relationship-Diagramm der Relation zwischen Präfix und Face*

Die FIB von CCN-IoT, dargestellt in Abbildung 4.25, setzt diese Relation um. Sie ist eine sortierte Liste von N Tupeln, bestehend aus Präfix und einem Bitvektor $([b_1, b_2, \ldots, b_F])$. Die Sortierung der Einträge erfolgt anhand der Präfixe nach Shortlex-Ordnung. Ist ein Präfix dem f-ten Face zugeordnet, so ist das f-te Bit (b_f) im Bitvektor gesetzt $(f \in \{1, \ldots, F\})$. Die Präfixe sind Instanzen der Klasse `Name` (vgl. Klassen der Buffer-Implementierungen in Abbildung 4.18 auf Seite 81). Präfix n besteht dabei aus M_n Komponenten (Comp_{n,M_n} mit $n \in \{1, \ldots, N\}$). Jede Komponente ist ein String bestehend aus Bytes, die Komponenten werden mit Typ-Längen-Feldern getrennt.

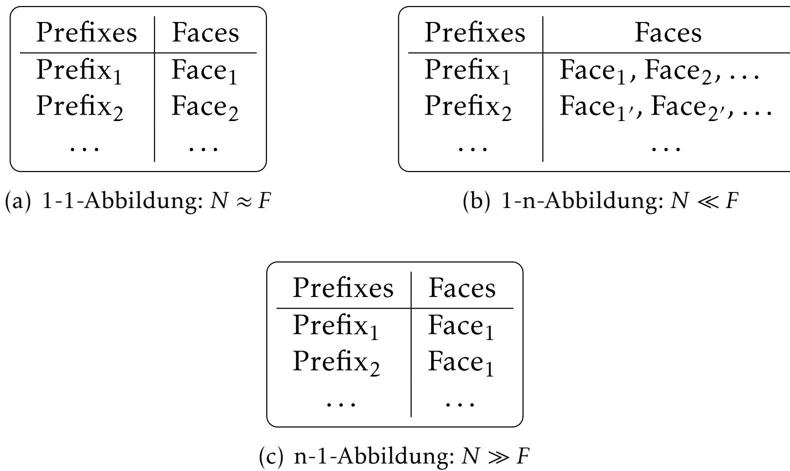
$$
\begin{array}{lllll}
(/\mathrm{Comp}_{1,1} & /\mathrm{Comp}_{1,2} & \cdots & /\mathrm{Comp}_{1,M_1}, & [\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_F]), \\
(/\mathrm{Comp}_{2,1} & /\mathrm{Comp}_{2,2} & \cdots & /\mathrm{Comp}_{2,M_2}, & [\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_F]), \\
\vdots & \cdots & \ddots & \vdots & \vdots \\
(/\mathrm{Comp}_{N,1} & \cdots & /\mathrm{Comp}_{N,M_N-1} & /\mathrm{Comp}_{N,M_N}, & [\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_F])
\end{array}
$$

Abbildung 4.25 – *Struktur der FIB von CCN-IoT*

Da im Folgenden ein Vergleich des Speicherbedarfs der CCN-IoT FIB-Implementierung mit FIB-Hash und FIB-(C)BF durchgeführt wird, wird zunächst eine Größenabschätzung für die FIB-Implementierung von CCN-IoT gegeben. Die Größe von Bloom-Filtern wird üblicherweise in Bit angegeben. Für die FIB-Implementierung von CCN-IoT so wie sie in Abbildung 4.25 dargestellt ist, berechnet sich die Größe in Bit aus $\sum_{n=1}^{N} \sum_{m=1}^{M_n} (8 \cdot (|\mathrm{Comp}_{n,m}|_{Byte} + 1) + F)$. Dabei ist $|\mathrm{Comp}_{n,m}|_{Byte}$ die Größe in Byte der m-ten Komponente des n-ten Präfix. Für die Größenabschätzung ist die Betrachtung der einzelnen Komponenten eines Präfix für den Vergleich nicht erforderlich, da der Präfix als Ganzes betrachtet wird. Ein Präfix als Ganzes für einen FIB-Eintrag wird im Folgenden als Prefix_n (inklusive Typ-Längen-Felder) bezeichnet.

Die Länge eines Präfix, bezeichnet mit $|\mathrm{Prefix}_n|$, hat ebenfalls einen signifikanten Einfluss auf die Größe der FIB-Implementierung von CCN-IoT. Bei FIB-Hash, FIB-BF und FIB-CBF hat die Länge des Präfix keinen Einfluss. Aus diesem Grunde gilt bei der Größenabschätzung die Annahme, dass alle Präfixe in der FIB die gleiche Länge haben, was mit $|\mathrm{Prefix}_N|_{Bit}$ beziehungsweise $|\mathrm{Prefix}_N|_{Byte}$ ausgedrückt wird. Die Größe der FIB berechnet sich nun vereinfacht aus $N \cdot (|\mathrm{Prefix}_N|_{Bit} + F)$. Aus dieser Berechnung wird auch deutlich, dass insbesondere die Anzahl der Präfixe N die Größe der FIB wesentlich bestimmt.

Für FIB-BF in Abbildung 4.23 wird die Größe beispielsweise durch F bestimmt. Um die Auswirkungen von N und F auf die FIB-Implementierungen zu zeigen, werden in dieser Evaluation drei Fälle betrachtet: *(a)* $N \approx F$, *(b)* $N \ll F$ und *(c)* $N \gg F$. Nach diesen drei Fällen und den möglichen Beziehungen zwischen Präfix und Face in Abbildung 4.24, ergeben sich auch drei mögliche Abbildungen von Präfixen zu Faces in der FIB, die in Abbildung 4.26 dargestellt sind.



Prefixes	Faces
Prefix_1	Face_1
Prefix_2	Face_2
...	...

(a) 1-1-Abbildung: $N \approx F$

Prefixes	Faces
Prefix_1	$\text{Face}_1, \text{Face}_2, \ldots$
Prefix_2	$\text{Face}_{1'}, \text{Face}_{2'}, \ldots$
...	...

(b) 1-n-Abbildung: $N \ll F$

Prefixes	Faces
Prefix_1	Face_1
Prefix_2	Face_1
...	...

(c) n-1-Abbildung: $N \gg F$

ABBILDUNG 4.26 – *Abbildungen von Präfixen zu Faces in der FIB*

Bei der 1-1-Abbildung in Abbildung 4.26(a) wird ein Präfix einem Face zugeordnet, bei der 1-n-Abbildung in Abbildung 4.26(b) werden einem Präfix mehrere Faces zugeordnet und bei der n-1-Abbildung in Abbildung 4.26(c) werden mehrere Präfixe auf ein Face abgebildet. Wie später gezeigt wird, ist die Abbildung der Präfixe auf Faces entscheidend für die Effizienz der jeweiligen Optimierungsstrategie der FIB.

In der Praxis gibt es bis bisher keine belastbaren Aussagen, wie Präfixe auf Faces in der FIB grundsätzlich abgebildet werden. Für das öffentlich zugängliche NDN-Testbed wurde exemplarisch eine Untersuchung vorgenommen, wie dort Präfixe auf Faces abgebildet werden. Die Ergebnisse dieser Untersuchung sind im Anhang in Abschnitt A.2.1 aufgeführt. Daher werden in der Evaluation die drei möglichen Abbildungen betrachtet.

Im Folgenden werden die wichtigsten Ergebnisse aus der diesem Abschnitt zugrunde liegenden Veröffentlichung [1] vorgestellt. Für die folgenden Simulationen gelten die Rahmenbedingungen, wenn nicht explizit anders angegeben: Als Falsch-Positiv-Wahrscheinlichkeit für die Bloom-Filter wird $P_{FP} = 10^{-6}$ festgelegt und die Präfixe werden gleich (oder annähernd gleich) auf die Faces verteilt. Eine Falsch-Positiv-Wahrscheinlichkeit von 10^{-6} bietet einen vernünftigen Kompromiss zwischen der Größe und Präzision des Bloom-Filters. Da die Präfixe gleich auf die Faces verteilt werden, ist die Kapazität der Bloom-Filter $N_{BF} \approx N/F$. Sind die Präfixe nicht gleich auf die Faces verteilt, dann gibt es in FIB-BF Bloom-Filter, die weniger Elemente als N_{BF} aufnehmen, was eine ineffiziente Speicherausnutzung darstellt.

Für die Anzahl der Präfixe N werden in den Simulationen Werte von 50 und 60 und für die Anzahl der Faces F Werte von 10 bis 15 verwendet. Die Präfixlänge wird mit $|Prefix_N|_{Byte} = 15$ festgelegt. Werte für die Anzahl der Präfixe N und der Anzahl der Faces F sowie die Präfixlänge sind ebenfalls das Ergebnis einer Untersuchung der FIB Einträge NDN-Testbeds. Die Werte für die Anzahl der Präfixe und der Anzahl der Faces liegen ungefähr in der Mitte der beobachteten Werte im NDN-Testbed, wie aus Abbildung A.14 auf Seite 174 im Anhang hervorgeht. Bei der Präfixlänge gibt es Häufungen bei 11 bis 16 Byte und 29 bis 34 Byte, wie im Anhang in Abbildung A.15 auf Seite 176 dargestellt. Die Wahl von 15 Byte für die Präfixlänge orientiert sich an der ersten Häufung und erscheint realistisch für das Internet der Dinge. Verfahren wie FIB-Hash und FIB-BF werden außerdem mit zunehmender Präfixlänge immer besser, daher ist für eine kritische Evaluation der Wert für die Präfixlänge eher kleiner zu wählen.

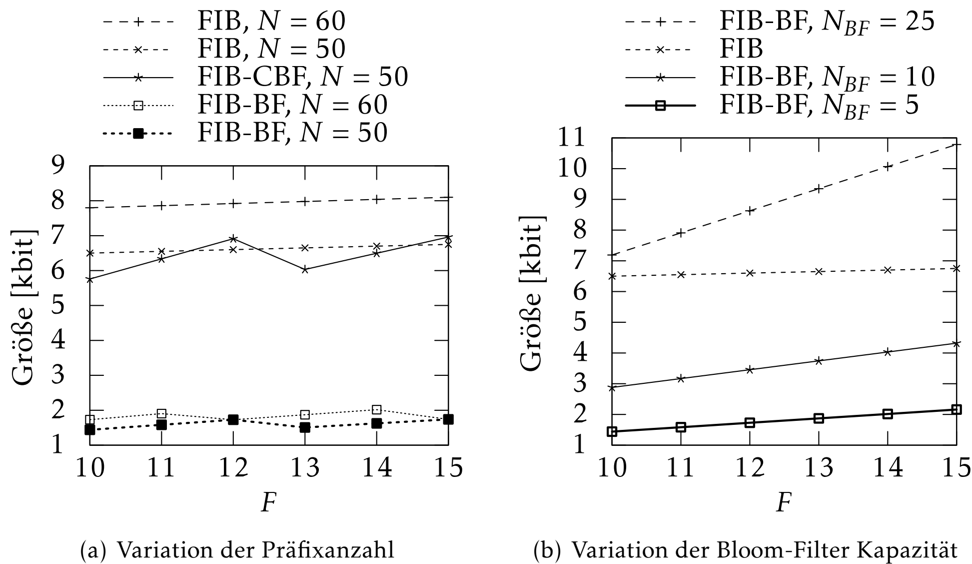
Für FIB-BF wird eine n-1-Abbildung von Präfixen auf Faces vorgeschlagen, da nach der Konstruktion von FIB-BF (vgl. Struktur von FIB-BF in Abbildung 4.23 auf Seite 90) die Bloom-Filter mehrere Präfixe für jeweils ein Face aufnehmen. Somit steigt der Speicherverbrauch von FIB-BF mit wachsendem F schneller als mit wachsendem N_{BF}. Die Größe von FIB-BF ist $size_{BF}(N_{BF}, P_{FP}) \cdot F$, wobei $size_{BF}(N_{BF}, P_{FP})$ die Größe der einzelnen Bloom-Filter von FIB-BF ist. Die Größe der zählenden Bloom-Filter für FIB-CBF erhöht sich um den Faktor vier, da jedes Bit durch einen vier Bit breiten Zähler im Bloom-Filter ersetzt wird. Ein vier Bit breiter Zähler ist laut Fan et al. [71] für die meisten Anwendungen von zählenden Bloom-Filtern ausreichend.

EINFLUSS VON N, F UND N_{BF}

In diesem Abschnitt werden zunächst die Ergebnisse des Einflusses von N, F und N_{BF} auf FIB-BF und FIB-CBF gezeigt. Die Speicherersparnis von FIB-Hash ist offensichtlich, weil die Präfixe durch vier Byte große Hashes ersetzt werden. Die FIB-Implementierung von CCN-IoT (vgl. Abbildung 4.25 auf Seite 91) wird im Folgenden mit FIB bezeichnet. Abbildung 4.27 zeigt, wie sich die Größen von FIB, FIB-BF und FIB-CBF mit steigender Anzahl von Faces F verändern. Die x-Achse zeigt die Anzahl der Faces F und die y-Achse die die Größe der jeweiligen Datenstrukturen in kbit (Kilobit). Um ganzzahlige Werte für die Kapazität der Bloom-Filter zu erhalten, ist $N_{BF} := \lceil N/F \rceil$.

Unter der Annahme, dass die Präfixe gleich auf die Faces verteilt werden, wird in Abbildung 4.27(a) deutlich, dass FIB-BF viereinhalb mal weniger Speicher verbraucht als FIB. Im Gegensatz zu FIB wächst FIB-BF nicht mit steigendem F. Wird allerdings F weiter vergrößert, so dass $N = F$ gilt, handelt es sich um eine 1-1-Abbildung, für welche FIB-Hash besser geeignet ist.

Die Lösung mit zählenden Bloom-Filtern (FIB-CBF, $N = 60$) hat einen viermal höheren Speicherbedarf und liegt damit auf dem Niveau von FIB. FIB-CBF für $N = 60$ ist aufgrund des hohen Speicherverbrauchs in Abbildung 4.27(a) nicht dargestellt.



(a) Variation der Präfixanzahl

(b) Variation der Bloom-Filter Kapazität

Abbildung 4.27 – *Größen von FIB mit FIB-BF und FIB-CBF*

Weiterhin ist der Abstand der Kurven in Abbildung 4.27(a) zwischen FIB-BF $N = 50$ und $N = 60$ geringer als zwischen FIB $N = 50$ und $N = 60$. Für FIB ist liegt der Abstand bei 1,3 kbit, für FIB-BF hingegen liegt er zwischen 0 und 0,39 kbit. Das heißt, dass FIB-BF auch mit steigender Anzahl an Präfixen N besser als FIB skaliert.

Die Annahme, dass die Präfixe gleich auf die Faces verteilt sind und die Kapazität der Bloom-Filter $N_{BF} \approx N/F$, wichtig ist, zeigt Abbildung 4.27(b). In dem Setup in Abbildung 4.27(b) ist $N = 50$ und für N_{BF} werden die Werte 5, 10 und 25 angenommen. $N_{BF} = 5$ entspricht $N_{BF} \approx N/F$ bei einer Anzahl von 10 bis 15 Faces. Wird nun N_{BF} erhöht, erhöht sich auch der Speicherbedarf von FIB-BF. Ist $N_{BF} = 25$, so spart FIB-BF gegenüber FIB keinen Speicher ein.

EINFLUSS DER PRÄFIXLÄNGE

Diese Evaluation schließt mit der Untersuchung des Einflusses der Präfixlänge auf FIB, FIB-Hash, FIB-BF und FIB-CBF. Da die Größen der alternativen FIB-Implementierungen FIB-Hash, FIB-BF und FIB-CBF nicht von der Präfixlänge abhängen, übertreffen die alternativen Implementierungen FIB stets bei entsprechender Präfixlänge bezüglich des Speicherverbrauchs. Die Frage, die sich stellt, ist, ab welcher Präfixlänge die alternativen Implementierungen weniger Speicher als FIB verbrauchen. Um diese Frage zu beantworten, muss die kürzeste Präfixlänge ermittelt werden, ab der FIB-BF und FIB-CBF den gleichen Speicherverbrauch wie FIB haben. In dieser Arbeit wurde die kürzeste Präfixlänge für $N = 50$ ermittelt. Die Ergebnisse sind in Abbildung 4.28 dargestellt. Auf der x-Achse ist die Anzahl der Faces und auf der y-Achse ist die Präfixlänge angegeben. Der Bereich oberhalb der jeweiligen Kurve in Abbildung 4.28 steht damit für alle Präfixlängen, wo die jeweilige alternative Implementierung weniger Speicher verbraucht als FIB.

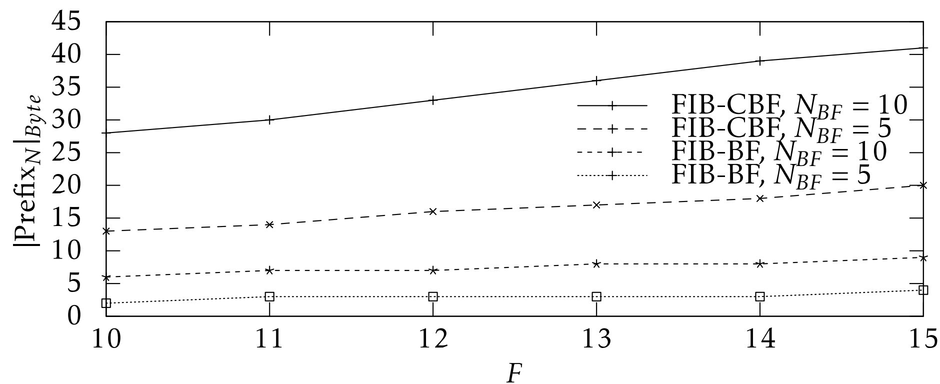


ABBILDUNG 4.28 – *Kürzeste Präfixlänge ab der FIB-BF und FIB-CBF einen identischen Speicherverbrauch zu FIB haben*

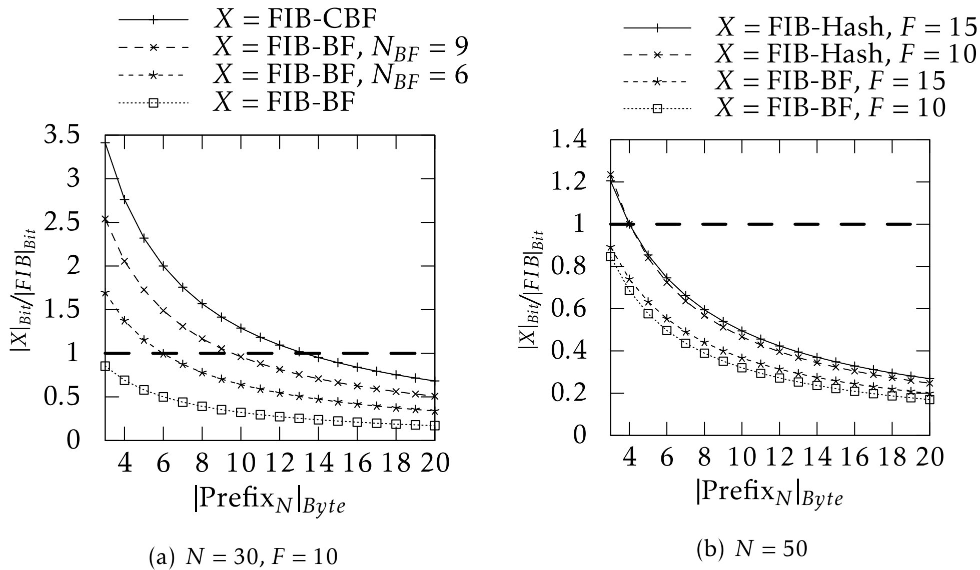
Es zeigt sich, dass FIB-BF unter den gegebenen Voraussetzungen bereits ab einer Präfixlänge von zwei bis vier Byte speicherplatzeffizienter als FIB ist. Wird die Kapazität der Bloom-Filter N_{BF} von fünf auf zehn erhöht, erhöht sich die Präfixlänge um den Faktor zwei, ab der FIB-BF speicherplatzeffizienter ist. Für FIB-CBF mit den zählenden Bloom-Filtern erhöht sich die Präfixlänge um den Faktor vier bis sechs.

Möchte man das Verhältnis von FIB, FIB-Hash, FIB-BF und FIB-CBF zur Präfixlänge zusammenfassen, bietet es sich an, den Quotient aus den Größen von FIB-Hash, FIB-BF und FIB-CBF und der Größe von FIB zu betrachten. Im Folgenden wird der Quotient als *Größenverhältnis* bezeichnet. Ist das Größenverhältnis kleiner eins, dann sind die alternativen Implementierungen speicherplatzeffizienter. Das Größenverhältnis aus FIB-Hash, FIB-BF und FIB-CBF und FIB ist auf der y-Achse von Abbildung 4.29 und die Präfixlänge ist auf der x-Achse aufgetragen. Die gestrichelte Linie in Abbildung 4.29 zeigt an, wo das Größenverhältnis kleiner eins ist.

In dem Beispiel in Abbildung 4.29(a) sind $N = 30$ und $F = 10$. Sind $N_{BF} = 6 = 2N/F$ beziehungsweise $N_{BF} = 9 = 3N/F$, so sind die Präfixe nicht mehr gleich auf die Faces verteilt und die Speicherplatzeffizienz verglichen mit FIB nimmt ab. Auch die Tatsache, dass FIB-BF stärker mit steigender Anzahl an Faces F wächst als mit steigender Anzahl an Präfixen N wird hier deutlich. Dieser Umstand ist in Tabelle 4.2 an ausgewählten Werten aus Abbildung 4.29 dargestellt. Es zeigt sich, dass Präfixlängen bei unterschiedlichen N ähnlicher sind, als mit unterschiedlichen F.

Ab einer Präfixlänge von vier Byte spart FIB-Hash immer Speicher, wie Abbildung 4.29(b) zeigt, da für FIB-Hash vier Byte große Hashes angenommen werden. FIB-CBF bei $N = 30$ und $F = 10$ ist erst ab einer Präfixlänge von mehr als 14 Byte speicherplatzeffizienter als FIB. Weiterhin verbraucht FIB-Hash in dem gegeben Setup immer weniger Speicher als FIB-CBF.

Zusammengefasst ist FIB-BF speicherplatzeffizienter als FIB und FIB-Hash, wenn die Präfixe gleich auf die Faces verteilt sind. FIB-Hash ist flexibler, da



(a) $N = 30, F = 10$

(b) $N = 50$

Abbildung 4.29 – *Präfixlänge gegen Größenverhältnis (Größe von HIB-Hash, FIB-BF und FIB-CBF geteilt durch die Größe von FIB)*

Label	Abbildung	Präfixlänge $\|Prefix\|_{Byte}$			
		4	6	8	10
X = FIB-BF, $N = 30$, $F = 10$	4.29(a)	0,69	0,5	0,39	0,32
X = FIB-BF, $N = 50$, $F = 15$	4.29(b)	0,74	0,55	0,44	0,37
X = FIB-BF, $N = 50$, $F = 15$		0,69	0,5	0,39	0,32

Tabelle 4.2 – *Ausgewählte Werte aus Abbildung 4.29*

FIB-Hash unabhängig von der Verteilung der Präfixe auf die Faces ist. Weiterhin erlaubt FIB-Hash im Gegensatz zu FIB-BF das Entfernen von Präfixen. FIB-Hash, FIB-BF und FIB-CBF sind, wenn die Präfixe lang genug sind, immer speicherplatzeffizienter als FIB, da nur bei FIB die Länge der Präfixe eine Rolle spielt. Tabelle 4.3 fasst die Evaluationsergebnisse der vorgeschlagenen Lösungen zur Optimierung der FIB kurz zusammen.

FIB-Typ	Abbildungs-charakteristik (Präfix zu Face)	Entfernen von Elementen	Deterministisch (Keine Falsch-Positiven Ergebnisse)	Kompressionsrate (typisch)
FIB-Hash	Keine Einschränkungen	Ja	Ja	~1/3
FIB-BF	n-1	Nein	Nein	~1/4
FIB-CBF	n-1	Ja	Nein	~0

Tabelle 4.3 – *Zusammenfassung der Evaluationsergebnisse*

4.4 ZUSAMMENFASSUNG

Dieses Kapitel stellte die Architektur und die Implementierung vor. Die Architektur gliedert sich in Systemarchitektur und Softwarearchitektur. Im Abschnitt Systemarchitektur wurden die Hard- und Softwarekomponenten des inhalts-/ namenszentrischen Internet der Dinge sowie die Schnittstellen zwischen den Komponenten identifiziert. Hardwarekomponenten sind Rechner wie PCs und Server im drahtgebundenen Netz und Sensorknoten im drahtlosen Netz. Als Vermittler zwischen dem drahtgebundenen und dem drahtlosen Netz kommen Gateways zum Einsatz.

Softwarekomponenten sind die Komponenten von CCN, wie der Dämon, und die Komponenten der namenszentrischen Dienste. Namenszentrische Dienste setzen sich aus den Softwarekomponenten Dienstanbieter und Dienstnutzer sowie den Proxies zusammen. Die Schnittstellen zwischen dem Diensten und dem Dämon sind die Faces. Zwischen den Diensten realisieren die Dienstmethoden die sogenannte Service-Schnittstelle.

Die Softwarearchitektur betrachtet den inneren Aufbau der Komponenten am Beispiel von CCN-IoT. CCN-IoT ist die CCN-Implementierung für ressourcenbeschränkte drahtlose Sensorknoten, die im Rahmen der Arbeit entstanden ist. Der Fokus der Softwarearchitektur liegt auf den Klassen und deren Beziehungen untereinander.

Der zweite Teil des Kapitels greift zum einen wichtige Aspekte der Implementierung von CCN-IoT auf. Weiterhin wurde im zweiten Teil des Kapitels eine theoretische Untersuchungen zur Speicheroptimierung der Forwarding Information Base präsentiert und untersucht.

Ein wichtiger Aspekt der Implementierung von CCN-IoT ist die Datenstruktur Buffer, mit der Namen, Nachrichten und die Dienstmethoden der namenszentrischen Dienste umgesetzt werden. Bei der Verarbeitung von Nachrichten und Namen erfolgt mit sogenannten Iteratoren. Die Nachrichtenverarbeitung wurde ebenfalls in diesem Kapitel untersucht. Dabei wurde gezeigt, dass die Trennung in einen nur lesenden und einen lesenden und modifizierenden Iterator sinnvoll ist. Der nur lesende Iterator ist schneller als der lesende und modifizierende Iterator. Beide Iteratoren bauen aufeinander auf, somit wird Funktionalität wiederverwendet und letztlich nur soviel Programmspeicher verbraucht, wie für den lesenden und modifizierenden Iterator notwendig wäre. Weiterhin wurde auf die Implementierung der Datenstrukturen Content Store, Pending Interest Table und Forwarding Information Base in CCN-IoT eingegangen.

Um den Speicher der Forwarding Information Base zu optimieren, wurden Lösungen mit Hashing und Bloom-Filtern vorgeschlagen. Das Ergebnis der Untersuchungen zeigt, dass die Lösung mit Bloom-Filtern die höchste Kompressionsrate unter den vorgeschlagenen Optimierungslösungen hat, wenn die Rahmenbedingung gilt, dass Präfixe gleich auf die Faces verteilt sind. Die Lösung mit den Bloom-Filtern hat den Nachteil, dass eine Entfernung von Elementen aus der Forwarding Information Base nicht möglich ist.

KAPITEL 5

DIENSTBESCHREIBUNG UND WERKZEUGUNTERSTÜTZUNG

Der Beitrag dieses Kapitels ist die Umsetzung der Dienstbeschreibung und der Werkzeugunterstützung, die in Abschnitt 3.3.3 als wichtige Bestandteile der namenszentrischen Dienste eingeführt wurden. Ziel der Dienstbeschreibung und Werkzeugunterstützung ist es, die Entwicklung von namenszentrischen Diensten zu vereinfachen. So ist die Dienstbeschreibung ein Hilfsmittel für Entwickler, die Dienstbeschreibungen aus der Entwicklung von Diensten mit dem knotenzentrischen Ansatz kennen. Die Werkzeugunterstützung ersetzt die manuelle Entwicklung von namenszentrischen Diensten, da die manuelle Entwicklung fehleranfällig ist, wie im Abschnitt Dienstentwicklung auf Seite 39 dargestellt. Dieses Kapitel vervollständigt Architektur und Implementierung aus Kapitel 4 um die Realisierung von namenszentrischen Diensten.

Das Konzept von Dienstbeschreibung und Werkzeugunterstützung ist nicht neu. Allerdings gibt es für namenszentrische Dienste speziell mit dem Fokus auf das Internet der Dinge bisher keinen Vorschlag der Umsetzung von Dienstbeschreibung und Werkzeugunterstützung. Abschnitt 5.1 stellt daher verwandte Arbeiten zu Dienstbeschreibung und Werkzeugunterstützung aus dem knotenzentrischen Umfeld vor, die die namenszentrischen Dienste für das Internet der Dinge beeinflusst haben und legt auch dar, dass existierende Ansätze für Dienstbeschreibungen ungeeignet sind. Unter den verwandten Arbeiten werden darüber hinaus Grundlagen von JSON vorgestellt, weil die Dienstbeschreibung das JSON-Format benutzt.

Eine Dienstbeschreibung definiert die Datentypen und Methoden eines Dienstes. Daher ist ein Beitrag dieses Kapitels die Formalisierung der Beschreibung dieses Musters durch ein *Schema* in Abschnitt 5.2.

Bei der Werkzeugunterstützung nutzt man die Tatsache, dass Anweisungen im Quellcode bei der Interaktion von Diensten immer ähnlich sind. Das Werkzeug

ist somit in der Lage, Teile des Quellcodes zur Interaktion von Diensten automatisch zu erstellen, wobei die Dienstbeschreibung als Konfiguration für das Werkzeug dient. Die Werkzeugunterstützung wird in Abschnitt 5.3 vorgestellt und evaluiert.

5.1 VERWANDTE ARBEITEN

Serviceorientierte Architekturen, die auf dem Simple Object Access Protocol (SOAP) [49] basieren, nutzen zur Beschreibung der Dienste die Web Services Description Language (WSDL, vgl. Abschnitt 3.1). Um aus einer WSDL Dienstbeschreibung die Proxies und Stub Code zu generieren, gibt es verschiedenste Werkzeuge. gSOAP [164] generiert C/C++-Code, die Werkzeuge aus dem Apache CXF Framework [36] generieren Java-Code und das ZSI-Framework [141] stellt Werkzeuge zur Generierung von Code für die Programmiersprache Python bereit. Da SOAP Dienste interoperabel sind, gibt es Werkzeuge für verschiedenste Programmiersprachen. Die Werkzeugunterstützung, die im Rahmen der Arbeit für die namenszentrischen Dienste entwickelt wurde, unterstützt derzeit die Programmiersprachen C++ und Java und zeigt somit prinzipiell die Interoperabilität auf der Ebene der Programmiersprachen.

Neben WSDL, dem weit verbreiteten Standard für Dienstbeschreibungen in serviceorientierten Architekturen, gibt es noch eine Vielzahl weiterer Standards oder Vorschläge für die Beschreibung von Diensten. Einen Vorschlag für die Beschreibung von Diensten ist das JSON Web Service Protocol, kurz JSON-WSP. Leider gibt es zu JSON-WSP keine eigene Veröffentlichung, die Veröffentlichungen von Cirani et al. [56] und Huo et al. [81] jedoch beziehen sich auf JSON-WSP. JSON-WSP nutzt nicht wie WSDL XML als Grundlage, sondern, wie der Name sagt, die Java Script Object Notation (JSON) [183]. Im Gegensatz zu XML ist JSON deutlich leichtgewichtiger bei gleicher Beschreibungsmächtigkeit bei der Anwendung als Dienstbeschreibung, wie Nurseitov et al. in „Comparison of JSON and XML Data Interchange Formats: A Case Study“ [119] festgestellt haben. JSON-WSP hat sich in der Vergangenheit nicht durchgesetzt, weil ein Schema-Mechanismus fehlte, wie es ihn für XML gibt. Mit einem Schema lässt sich die Struktur eines Dokuments beschreiben. Somit lässt sich beispielsweise prüfen, ob ein Dokument eine gültige Dienstbeschreibung ist. Cirani et al. und Huo et al. nutzen in ihren Arbeiten ebenfalls eine an JSON angelegte Dienstbeschreibung. JSON-Schema [75, 99] ist seit Ende 2016 standardisiert. Da JSON verhältnismäßig leichtgewichtig und nun standardisiert ist, wurde es für die Dienstbeschreibung der namenszentrischen Dienste gewählt.

Die Dienstbeschreibung für die namenszentrischen Dienste basiert ebenfalls auf JSON und ist von JSON-WSP inspiriert. Zum Zeitpunkt der Entwicklung der Dienstbeschreibung für die namenszentrischen Dienste gab es noch kein JSON-Schema. Somit gibt es in der Veröffentlichung „Tool Chain for Application Development with Name-Centric Services“ [4] noch kein Schema für die Dienstbeschreibung der namenszentrische Dienste. In Abschnitt 5.2 ist nun ein Schema für die Beschreibung von namenszentrischen Diensten ergänzt. Im folgenden Abschnitt 5.1.1 erfolgt eine kurze Einführung in JSON.

5.1.1 JSON-GRUNDLAGEN

Nach RFC 8259 [51] ist JSON ein leichtgewichtiges, textbasiertes und sprachunabhängiges Datenaustauschformat. In JSON werden die folgenden sechs Datentypen definiert: *(i)* Zeichenketten, *(ii)* Zahlen, *(iii)* boolesche Werte, *(iv)* Nullwert, *(v)* Objekte und *(vi)* Arrays.

Eine *Zeichenkette* ist eine Folge von Null oder mehr Zeichen, die in Anführungszeichen (`"`) eingeschlossen sind. *Zahlen* sind Folgen aus den Ziffern 0 bis 9 und können ein negatives Vorzeichen (`-`) haben. In der Folge aus Ziffern kann ein Dezimalpunkt (`.`) vorkommen und am Ende der Zahl kann die Angabe eines Exponenten folgen. Ein Exponent beginnt mit einem „`e`“ oder „`E`“; darauf folgt ein Vorzeichen (`+` oder `-`) und eine Folge aus den Ziffern 0 bis 9. *Boolesche Werte* nehmen einen der zwei Wahrheitswerte `true` und `false` an. Ein *Nullwert* wird mit dem Schlüsselwort „`null`“ dargestellt. Ein Objekt beginnt mit einer öffnenden geschweiften Klammer (`{`) und endet mit einer schließenden geschweiften Klammer (`}`). Zwischen den geschweiften Klammern eines Objektes stehen die sogenannten *Eigenschaften* des Objektes in einer ungeordneten, mit Kommata getrennten Liste. Eine Eigenschaft ist ein *Schlüssel-Wert-Paar*, wobei der Schlüssel eine Zeichenkette ist, die mit einem Doppelpunkt (`:`) vom Wert getrennt wird: `"Schlüssel":Wert`. Der Wert ist einer der sechs JSON-Datentypen. Weiterhin sind in JSON auch leere Objekte, also Objekte ohne Eigenschaften zulässig. `Arrays` beginnen mit einer geöffneten eckigen Klammer (`[`) und enden mit einer schließenden eckigen Klammer (`]`). Arrays enthalten durch Kommata getrennte, geordnete Listen von Werten, wobei die Werte auch von unterschiedlichen Typen sein können. Ein Array mit einer Zeichenkette, einer Zahl und einem Objekt ist beispielsweise `["abc",123,{}]`. In JSON sind auch leere Arrays möglich.

Nach dieser Einführung in JSON wird im folgenden Abschnitt das Schema der Dienstbeschreibung erklärt. Dabei folgt zuerst eine Motivation, warum ein Schema benötigt wird. Im Anschluss daran wird das Schema selbst erklärt, wo die Begriffe verwendet werden, die in diesem Abschnitt eingeführt wurden.

5.2 SCHEMA DER DIENSTBESCHREIBUNG

Das Schema der Dienstbeschreibung für die namenszentrischen Dienste, im Folgenden kurz Schema genannt, dient als Hilfsmittel im Entwicklungsprozess für namenszentrische Dienste. Das Schema dient dem Entwickler als *Spezifikation*, um eine korrekte Dienstbeschreibung zu erstellen. Weiterhin dient die Dienstbeschreibung zur *Validierung*. Bei der Validierung werden Dienstbeschreibung und Schema als Eingabe in ein Software-Werkzeug, den *Validator*, gegeben. Ein Validator für JSON prüft, ob ein JSON-Dokument zu einem JSON-Schema passt. Im Kontext der namenszentrischen Dienste ist das JSON-Dokument die Dienstbeschreibung und das JSON-Schema das Schema der Dienstbeschreibung für die namenszentrischen Dienste. Ein Beispiel einer Dienstbeschreibung, die dem Schema folgt, findet sich im Anhang in Abschnitt A.3.2. In dem Beispiel wird eine Dienstbeschreibung für eine automatische Klimaregelung in Seecontainern präsentiert.

Wie das Schema als Hilfsmittel für *Dokumentation* und *Validierung* eingesetzt wird, ist in Abbildung 5.1 zusammengefasst. In Abbildung 5.1(a) hilft das Schema dem Entwickler, eine korrekte Dienstbeschreibung zu erstellen. Bei der Validierung in Abbildung 5.1(b) wird vom Entwickler mit dem Validator geprüft, ob die Dienstbeschreibung dem Schema folgt. Folgt die Dienstbeschreibung dem Schema, ist die Validierung erfolgreich. Die Validierung schlägt fehlt, wenn die Dienstbeschreibung nicht dem Schema folgt.

(a) Dokumentation bei Erstellung (b) Validierung

Abbildung 5.1 – *Anwendung des Schemas im Entwicklungsprozess*

Da nun klar ist, wie das Schema als Hilfsmittel im Entwicklungsprozess für namenszentrische Dienste eingesetzt wird, wird im Folgenden das Schema für die Dienstbeschreibung der namenszentrischen Dienste eingeführt. In diesem Kapitel wird das Schema auszugsweise wiedergegeben. Das vollständige Schema der Dienstbeschreibung wird aus Platzgründen im Anhang in Abschnitt A.3.1 dargestellt.

Die im Konzept in Abschnitt 3.3.3 kurz vorgestellte Dienstbeschreibung besteht aus drei Teilen: *(i)* Metadaten, *(ii)* Datentypen und *(iii)* Methoden. Metadaten sind Begleitinformationen wie beispielsweise die *Version* eines Dienstes. Datentypen sind vergleichbar mit den Strukturdatentypen der Programmiersprache C [25] und bestehen wie diese aus primitiven Datentypen oder wiederum aus strukturierten Datentypen. Methoden werden durch ihre Signatur beschrieben.

Metadaten, Datentypen und Methoden sind *Eigenschaften* (vgl. Eigenschaften von JSON in Abschnitt 5.1.1) eines JSON-Objektes, welches das Schema der Dienstbeschreibung repräsentiert. Der Quelltext 5.1 zeigt einen Teil des Schemas. An den geschweiften Klammern in Zeile 1 und Zeile 25 sieht man, dass das Schema selbst ein JSON-Objekt ist.

In JSON-Schema gibt es definierte Eigenschaften. Das heißt, dass die Schlüssel der Eigenschaften Schlüsselwörter von JSON-Schema sind und die Werte nur einen bestimmten Typ haben. Die in dem Schema für die Dienstbeschreibung verwendeten Schlüsselwörter von JSON-Schema *(i)* type, *(ii)* properties, *(iii)* pattern, *(iv)* additionalProperties und *(v)* required werden im Folgenden eingeführt. Eine komplette Darstellung der Eigenschaften/Schlüsselwörter von JSON-Schema findet sich in dem IETF Draft zur JSON-Schema Validierung [99]. Im Folgenden wird der Begriff *JSON-Dokumente* für die Dokumente verwendet, die valide im Sinne des Schemas sind. Im Kontext der namenszentrischen Dienste sind die JSON-Dokumente alle gültigen *Dienstbeschreibungen*.

```
{
  "type": "object",
  "properties": {
    "type": {
      "type": "string",
      "pattern":"ncs\/desc"
    },
    "name": { "type": "string" },
    "version": { "type": "string" },
    "uri": {
      "type": "string",
      "pattern": "^(ccnx:)(/[A-Za-z0-9-._~\\?#]+)+$"
    },
    "doc": { "type": "string" },
    "types": {
      ⋮
    },
    "methods": {
      ⋮
    },
  },
  "additionalProperties": false,
  "required": [ "type", "name", "version", "uri" ],
    ⋮
}
```

Quelltext 5.1 – *JSON-Schema der Dienstbeschreibung (gekürzt)*

Die Eigenschaft mit dem Schlüssel `type` taucht in Quelltext 5.1 mehrmals auf. Der Wert `object` in Zeile 2 sagt aus, dass die JSON-Dokumente selbst JSON-Objekte sind. Weiterhin wird die Eigenschaft mit dem Schlüssel `type` dazu verwendet, JSON-Typen für Werte von Eigenschaften in den JSON-Dokumenten festzulegen.

Die Eigenschaften der JSON-Dokumente werden in der Eigenschaft des Schemas mit dem Schlüssel `properties` eingeführt. Der Wert von `properties` ist ein Objekt (Zeile 3 bis 21). Die Eigenschaften des `properties`-Objekts sind die Eigenschaften der JSON-Dokumente. So gibt es in jeder Dienstbeschreibung eine Eigenschaft mit dem Schlüssel `type` (Zeile 4), diese hat einen Wert vom mit dem JSON-Typ Zeichenkette (`string` in Zeile 5).

Der Wert der Eigenschaft mit dem Schlüssel `pattern` ist ein regulärer Ausdruck vom JSON-Typ `string`. Mit einem regulären Ausdruck wird der Wertebereich eines Strings eingeschränkt. Die Eigenschaft mit dem Schlüssel `type` der Dienstbeschreibungen kann nur den Wert `"ncs/desc"` annehmen (Zeile 6).

Die Eigenschaft mit dem Schlüssel `additionalProperties` hat in Quelltext 5.1 den booleschen Wert `false`. Das heißt, dass es außer den in `properties` definierten Eigenschaften keine weiteren Eigenschaften gibt.

Die Eigenschaft mit dem Schlüssel `required` legt die obligatorischen Eigenschaften der JSON-Dokumente fest. Wenn eine in dem JSON-Array von `required` aufgeführte Eigenschaft in den JSON-Dokumenten fehlt, ist die Dienstbeschreibung ungültig. Nach dem Schema in Quelltext 5.1 hat eine Beschreibung für die namenszentrischen Dienste sieben Eigenschaften: `type`, `name`, `version`, `uri`, `doc`, `types` und `methods`. Die Eigenschaften werden im Folgenden erklärt.

EIGENSCHAFTEN DER DIENSTBESCHREIBUNG

Die ersten fünf Eigenschaften `type`, `name`, `version`, `uri`, `doc` sind die Metadaten der Beschreibung. Eigenschaft `types` definiert die Datentypen und die Eigenschaft `methods` die Methoden der Dienstbeschreibung.

Eigenschaft `type` beschreibt den Typ des Dokuments. Der reguläre Ausdruck in Quelltext 5.1, Zeile 6 stellt sicher, dass er immer `ncs/desc` (engl. kurz für Name-Centric Service Description) ist. Die `type`-Eigenschaft gibt es auch bei JSON-WSP, wo der Wert `"jsonwsp/description"` lautet. Mit der `type`-Eigenschaft ist eine schnelle Unterscheidung einer Dienstbeschreibung eines namenszentrischen Dienstes von einer Dienstbeschreibung eines JSON-WSP-Dienstes ohne Validator möglich.

Eigenschaft `name` ist der Name des Dienstes, der ihm während der Entwicklungsphase zugewiesen wird. Der Name wird in Nachrichten verwendet, beziehungsweise es werden Hashes von den Namen gebildet, die in den Nachrichten verwendet werden. Details zur Zuweisung von Namen und deren Nutzung werden in Kapitel 6 behandelt. Weiterhin leitet die Werkzeugunterstützung aus dem Wert der Eigenschaft `name` die Namensräume oder Pakete und Klassennamen ab, die im generierten Code verwendet werden.

Der Wert der Eigenschaft `version` ist die Version des Dienstes. Üblicherweise beinhaltet der String einen numerischen Wert, es sind jedoch beliebige Strings zur Identifikation der Version möglich. Dienstanbieter- und Dienstnutzer-Proxy sind inkompatibel, wenn ihre Implementierung auf unterschiedlichen Versionen beruhen. Beispielsweise wird die Version geändert, wenn sich das Verhalten eines Dienstes ändert, die Schnittstelle des Dienstes, wie die Methoden und die Signaturen, sich nicht aber geändert haben.

Der Wert der Eigenschaft `uri` ist der Name der Dienstbeschreibung. Bei JSON-WSP gibt es die Eigenschaft URL, die die Adresse beziehungsweise den Ort des Dienstanbieters angibt. Bei den namenszentrischen Diensten hingegen wird mit der URI die Dienstbeschreibung adressiert. Neue Versionen einer Dienstbeschreibung sind immer unter dem Namen auffindbar. Die Dienstbeschreibung ist im Netz auf Geräten gespeichert, die über die notwendigen Ressourcen verfügen. Die URI folgt der URI-Syntax [42] für CCN-Namen, das URI-Schema ist `ccnx`.

Eigenschaft `doc` dient der Dokumentation (engl. Documentation, kurz Doc). Eine `doc`-Eigenschaft gibt es einmal für die gesamte Dienstbeschreibung, weiterhin haben Datentypen und Methoden ebenfalls eine `doc`-Eigenschaft. Die `doc`-Eigenschaft der Dienstbeschreibung beschreibt den Dienst im Klartext, vergleichbar eines Kommentars in einer Programmiersprache. Alle `doc`-Eigenschaften der Beschreibung sind optional. Die Werkzeugunterstützung generiert aus den `doc`-

```
1      ⋮
2  "types": {
3    "type": "object",
4    "patternProperties": {
5      "^[A-Za-z_][A-Za-z_0-9]*$": {
6        "type": "object",
7          "patternProperties": {
8            "^[A-Za-z_][A-Za-z_0-9]*$": {
9              "type": "object",
10             "properties": {
11               "doc": { "type": "string" },
12               "type": {"$ref":"#/defs/validIdentifier"}
13             },
14             "additionalProperties": false,
15             "required": [ "type" ]
16           }
17         },
18         "minProperties": 1
19     }
20   },
21   "additionalProperties": false,
22   "minProperties": 1
23 },
24     ⋮
```

Quelltext 5.2 – *JSON-Schema der zusammengesetzten Datentypen*

Eigenschaften Kommentare für den Stub-Code, so sind die Informationen für die Entwickler schnell verfügbar.

Eigenschaft `types` beschreibt die zusammengesetzten Datentypen des Dienstes. Der Wert der `types`-Eigenschaft ist ein JSON-Objekt. Eigenschaft `types` ist optional. Ist sie definiert, enthält diese mindestens einen zusammengesetzten Datentyp. Die Definition der `types`-Eigenschaft findet sich in Zeile 3 in Quelltext 5.2 des Schemas.

Die Eigenschaft mit dem JSON-Schema Schlüsselwort `patternProperties` definiert eine Eigenschaft im JSON-Dokument und ist daher vergleichbar mit der Eigenschaft `properties`. Im Gegensatz zur `properties`-Eigenschaft sind die Bezeichnungen für die definierten Eigenschaften beliebig, allerdings mit Einschränkungen, wählbar. Die Beschränkung wird durch den regulären Ausdruck erreicht, der anstatt einer konkreten Bezeichnung für die Eigenschaft angegeben ist. Bezeichner für zusammengesetzte Typen sind die Schlüssel für die Eigenschaften in der Dienstbeschreibung, die einen zusammengesetzten Datentyp beschreiben. Diese Bezeichner werden unter Berücksichtigung von Namenskonventionen in der Dienstbeschreibung beliebig gewählt. Die regulären Ausdrücke in Zeile 4 und 8 beschränken Namen für Typen auf solche, die mit einem Buchstaben beginnen und dann Zahlen, Buchstaben und Unterstriche in beliebiger Kombination enthalten.

Die JSON-Schema Eigenschaft `minProperties` hat den Wert Eins, was bedeutet dass es mindestens einen zusammengesetzten Datentyp geben muss, wenn die Eigenschaft `types` in der Dienstbeschreibung definiert wurde. Wie oben bereits erwähnt, macht die Angabe der `types`-Eigenschaft ohne Datentypen keinen Sinn.

Ein zusammengesetzter Datentyp in der Dienstbeschreibung ist wiederum ein JSON-Objekt (Zeile 9 in Quelltext 5.2). Er hat mindestens ein Feld (Zeile 18). Ein Feld hat einen Namen (Zeile 8) und zwingend einen Typ (Zeile 12 und Zeile 15). Die URI-Referenz in Zeile 12 ist ein Verweis auf eine Eigenschaft in dem Schema, die weiter unten im Abschnitt „Abhängigkeiten und Datentypen“ eingeführt wird. Mit URI-Referenzen werden Duplikate in der Schema-Definition vermieden. Optional hat ein Feld einen Kommentar für Dokumentationszwecke (Zeile 11).

Der Wert der Eigenschaft `methods` wird ebenfalls als Objekt definiert. Wie für die Datentypen gilt, dass es mindestens einen Methode geben muss, wenn die `methods`-Eigenschaft definiert wurde. Quelltext 5.3 zeigt die `methods`-Eigenschaft. Dass dort mindestens eine Methode definiert werden muss, geht aus der `minProperties`-Eigenschaft in Zeile 29 hervor. Eine Methode hat einen Namen (Zeile 5) und die Eigenschaften `doc`, `params` und `ret_type`.

Eigenschaft `doc` dient der Dokumentation der Methode. In der Eigenschaft `params` wird die Parameterliste der Methode abgebildet. Die Definition der Parameterliste erfolgt im Wert (JSON-Objekt) der `params`-Eigenschaft in den Zeilen 9 bis 23 im Schema in Quelltext 5.3. Parameter werden mit der `patternProperties`-Eigenschaft definiert (Zeile 11). Ein Parameter hat einen Typ (Zeile 16), beschrieben durch die Eigenschaft `type` und einen Kommentar (Zeile 15). Der Typ ist für den Parameter obligatorisch, was durch die `required`-Eigenschaft in Zeile 19 ausgedrückt wird. Mit der optionalen Eigenschaft `ret_type` in Zeile 24 wird der Rückgabetyp definiert. Der Wert von `ret_type` ist ein Objekt mit einer URI-Referenz. An der URI-Referenz werden die Datentypen für den Rückgabewert definiert. URI-Referenzen werden weiter unten eingeführt.

ABHÄNGIGKEITEN UND DATENTYPEN

Zusammengesetzte Datentypen werden von Dienstmethoden zurückgegeben. Damit ist die Eigenschaft `types` abhängig von der Eigenschaft `methods`. In dem Schema der Dienstbeschreibung werden Abhängigkeiten mit der Eigenschaft `dependencies` (Zeile 2 in Quelltext 5.4) eingeführt.

Die Basis für die zusammengesetzten Datentypen der Dienstbeschreibung sind primitive Datentypen. Gültige Bezeichner werden mit der Eigenschaft `defs` definiert. Eigenschaft `defs` ist kein JSON-Schema Schlüsselwort; der Name kann beliebig gewählt werden. Die Bezeichnung `defs` ist eine Kurzform für *Definitions* (dt. Definitionen) und wurde gewählt, weil sich dort Definitionen des Schemas befinden. Der Wert von `defs` ist ein JSON-Objekt, definiert zwischen den geschweiften Klammern in Zeile 3 und 17. In `defs` werden die Eigenschaften `ncsIdentifier` und `validIdentifier` definiert. Unter `ncsIdentifier` werden die primitiven Datentypen der namenszentrischen Dienste definiert und unter `validIdentifier` die gültigen Typen für Felder von zusammengesetzten Da-

```
1       ⋮
2  "methods": {
3    "type": "object",
4    "patternProperties": {
5      "^[A-Za-z_][A-Za-z_0-9]*$": {
6        "type": "object",
7        "properties": {
8          "doc": { "type": "string" },
9          "params": {
10           "type": "object",
11           "patternProperties": {
12             "^[A-Za-z_][A-Za-z_0-9]*$": {
13               "type": "object",
14               "properties": {
15                 "doc": { "type": "string" },
16                 "type": {"$ref":"#/defs/ncsIdentifier"}
17               },
18               "additionalProperties": false,
19               "required": [ "type" ]
20             }
21           },
22           "minProperties": 1
23         },
24         "ret_type": {"$ref":"#/defs/validIdentifier"}
25       }
26     }
27   },
28   "additionalProperties": false,
29   "minProperties": 1
30 }
31      ⋮
```

Quelltext 5.3 – *JSON-Schema der Dienstmethoden*

tentypen, die entweder ein primitiver Datentyp oder ein zusammengesetzter Datentyp sind. Die URI-Referenzen in den Quelltexten 5.2 und 5.3 verweisen auf die Eigenschaften `ncsIdentifier` und `validIdentifier` in der `defs`-Eigenschaft.

`ncsIdentifier` ist eine Eigenschaft vom Typ String, das Pattern in Zeile 6 definiert die Bezeichnernamen für primitive Datentypen und Arrays aus primitiven Datentypen. Primitive Datentypen sind Ganzzahltypen oder Arrays von diesen Typen. Primitive Typen, die mit einem `i` (engl. für Integer) beginnen, sind vorzeichenbehaftet und Typen, die mit einem `u` (engl. für Unsigned) beginnen, sind vorzeichenlos. Die Zahl hinter dem Buchstaben gibt die Breite des Datentyps in Bit an. Ist hinter dem Datentyp eine öffnende und schließende eckige Klammer angegeben, handelt es sich um einen Array aus dem primitiven Datentyp. Alle primitiven Datentypen der namenszentrischen Dienste sind in Tabelle 5.1 mit Bezeichner und Wertebereichen zusammengefasst.

```
1     ⋮
2  "dependencies": { "types": [ "methods" ] },
3  "defs": {
4    "ncsIdentifier": {
5      "type": "string",
6      "pattern": "^[iu](8|16|32)(\\[\\])*$"
7    },
8    "validIdentifier": {
9      "anyOf": [
10       {
11         "type": "string",
12         "pattern": "^[A-Za-z_][A-Za-z_0-9]*$"
13       },
14       { "$ref": "#/defs/ncsIdentifier" }
15     ]
16   }
17 }
18    ⋮
```

QUELLTEXT 5.4 – *JSON-Schema der Abhängigkeiten und Definitionen der Typnamen*

Bezeichner		Wertebereich
Primitive Datentypen	i8	$-2^7 \ldots 2^7 - 1$
	i16	$-2^{15} \ldots 2^{15} - 1$
	i32	$-2^{31} \ldots 2^{31} - 1$
	u8	$0 \ldots 2^8$
	u16	$0 \ldots 2^{16}$
	u32	$0 \ldots 2^{32}$
Array	x[]	x ist ein primitiver Datentyp

TABELLE 5.1 – *Bezeichner und Wertebereiche der Datentypen der namenszentrischen Dienste*

In der aktuellen Implementierung von CCN-IoT sind für Rückgabetypen neben primitiven Datentypen und Arrays auch zusammengesetzte Datentypen zugelassen. Die gültigen Bezeichner für Rückgabetypen werden von der Eigenschaft `validIdentifier` definiert. Eigenschaft `validIdentifier` ist entweder ein Bezeichner für einen zusammengesetzten Datentyp, der dem Pattern in Zeile 12 folgt oder, nach Zeile 14, ein Bezeichner für einen primitiven Datentyp wie aus Tabelle 5.1.

5.3 ENTWICKLUNG VON DIENSTEN

Dieser Abschnitt führt in die Entwicklung der namenszentrischen Dienste für das Internet der Dinge ein. Wie im Konzept in Abschnitt 3.3.3 eingeführt, wird ein Werkzeug zur Generierung von Proxies und Stub-Codes verwendet. Daher wird im Folgenden auf den Aufbau und die Implementierung des Werkzeugs sowie auf die generierten Dateien eingegangen.

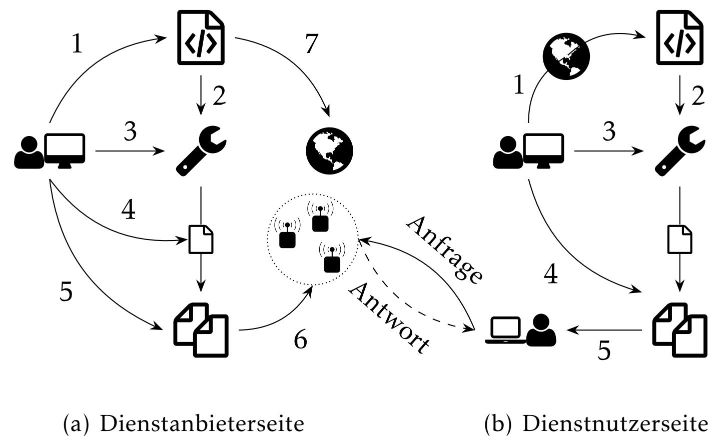
5.3.1 WERKZEUG ZU CODEGENERIERUNG

Bei dem Werkzeug handelt es sich um Hilfsmittel zur Generierung von Quellcode für Dienstanbieter- und Dienstnutzer-Proxies sowie den Stub-Code. Bevor die Details zum Aufbau des Werkzeugs präsentiert werden, wird zunächst erläutert, wie die Werkzeugunterstützung im Softwarelebenszyklus der namenszentrischen Dienste eingesetzt wird.

Die Implementierung eines neuen namenszentrischen Dienstes startet mit der Erstellung einer Dienstbeschreibung durch einen Entwickler. Ist die Dienstbeschreibung fertiggestellt, wendet der Entwickler ein Werkzeug zur Codegenerierung auf die Dienstbeschreibung an. Aus der Dienstbeschreibung wird der Dienstanbieter-Proxy sowie der Stub-Code für den Dienstanbieter erstellt. Der Entwickler implementiert die Logik des Dienstes im Stub-Code. Mit Logik ist hier der Anwendungscode gemeint, beispielsweise bei einem Dienst zur Temperaturermittlung wäre die Logik das Auslesen des Rohwertes des Temperatursensors sowie die Umrechnung in Grad Celsius. Ist der Dienst fertig implementiert und getestet, wird der Dienst auf die Knoten ausgebracht. Damit Dienstnutzer den Dienst auch nutzen können, wird die Dienstbeschreibung auf (mindestens) einem Knoten im inhaltszentrischen Netz bereitgestellt, indem sie im Content Store des Knotens abgelegt wird. In dieser Arbeit werden in Kapitel 6, Abschnitt 6.4.3 Standardnamen vorgeschlagen, die dabei helfen, eine bereitgestellte Dienstbeschreibung abzurufen. Ist der Dienst und seine Beschreibung über das Netz verfügbar, gilt der Dienst als *ausgerollt*.

Ein Entwickler, der einen Dienst nutzen möchte, ruft die bereitgestellte Dienstbeschreibung über das Netz ab. Die Dienstbeschreibung dient als Eingabe für einen Codegenerator, der einen Dienstnutzer-Proxy generiert. Für den Entwickler ist es vorteilhaft, dass die Dienstbeschreibung bereits existiert. Somit entfällt der Entwicklungsaufwand für den Dienstnutzer-Proxy. Der generierte Code des Dienstnutzer-Proxies wird in einen Anwendungscode eingebunden, wo der Entwickler die Methoden des Dienstnutzer-Proxies in dem Anwendungscode aufruft. Die fertige Software wird anschließend von den Benutzern installiert, wobei sie die angebotenen Dienste über das Netz nutzen.

Abbildung 5.2 fasst den Einsatz der Werkzeugunterstützung im Softwarelebenszyklus für die namenszentrischen Dienste im Internet der Dinge an einem Beispiel zusammen. Nummerierte Pfeile in Abbildung 5.2 teilen den Softwarelebenszyklus in Schritte ein, die im Folgenden erklärt werden. Die Entwicklung eines Dienstes beginnt mit der Entwicklung des Dienstanbieters, dargestellt in Abbildung 5.2(a). In *Schritt 1* erstellt ein Entwickler zunächst eine Dienstbeschreibung. Die Dienstbeschreibung dient als Eingabe für den Codegenerator



(a) Dienstanbieterseite (b) Dienstnutzerseite

Abbildung 5.2 – *Werkzeugunterstützung im Softwarelebenszyklus der namenszentrischen Dienste*

in *Schritt 2*, welcher, angewendet auf die Dienstbeschreibung in *Schritt 3* den Dienstanbieter-Proxy und den Stub-Code erstellt. Anschließend in *Schritt 4* implementiert der Entwickler die Logik des Dienstes im Stub-Code und integriert in *Schritt 5* den Dienstanbieter-Proxy sowie den im Stub implementierten Dienstanbieter-Code in eine Laufzeitumgebung für die Knoten, die CCN-IoT sowie die benötigten Treiber enthält. Nach Schritt 5 ist die Software für den Dienstanbieter fertig. Diese Software wird in *Schritt 6* auf die Knoten – ein drahtloses Sensornetz in Abbildung 5.2(a) – ausgebracht und die Dienstbeschreibung in *Schritt 7* über das Netz bereitgestellt.

Möchte ein Entwickler in Abbildung 5.2(b) den Dienst in seiner Software nutzen, so ruft er die bereitgestellte Dienstbeschreibung in *Schritt 1* über das Netz ab. Die Dienstbeschreibung dient als Eingabe für den Codegenerator in *Schritt 2*, welcher, angewendet auf die Dienstbeschreibung in *Schritt 3* den Dienstnutzer-Proxy erstellt. In *Schritt 4* wird der generierte Dienstnutzer-Proxy in die Software integriert, die den Dienst nutzen will. Diese Software ist somit ein Dienstnutzer und wird in *Schritt 5* an Anwender ausgeliefert. Bei der Anwendung der Software werden Anfragen und Antworten zwischen Dienstanbieter und Dienstnutzer über das Netz ausgetauscht.

Die Anwendung der Werkzeugunterstützung im Softwarelebenszyklus ist nun geklärt. Im nächsten Abschnitt wird erklärt, wie die Werkzeugunterstützung arbeitet und wie sie aufgebaut ist.

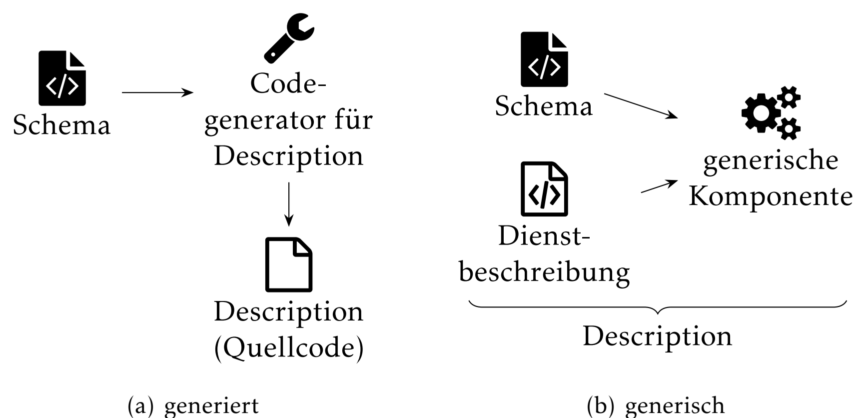
5.3.2 ARCHITEKTUR UND IMPLEMENTIERUNG

Dieser Abschnitt zeigt auf, wie die Werkzeugunterstützung aus der Dienstbeschreibung den Code generiert. Der Abschnitt gliedert sich in zwei Teile: im ersten Teil wird die Komponente zur Verarbeitung der Dienstbeschreibung beschrieben. Diese wird von Codegeneratoren verwendet, die im zweiten Teil beschrieben werden.

VERARBEITUNG DER DIENSTBESCHREIBUNG

Die Dienstbeschreibung liegt als JSON-Dokument vor. Um aus der Beschreibung Code zu generieren, muss die Dienstbeschreibung in eine Form gebracht werden, mit der ein Codegenerator arbeiten kann. So eine Form ist beispielsweise die Repräsentation der Eigenschaften des JSON-Dokuments als Objekte. Die Form, die die Eigenschaften als Objekte repräsentiert, ist eine Komponente (z. B. eine Klasse) der Werkzeugunterstützung. Diese Komponente wird – um sie von dem JSON-Dokument der Dienstbeschreibung abzugrenzen – im Folgenden mit dem englischen Begriff *Description* bezeichnet.

Die Komponente Description folgt ihrem Aufbau nach den formalen Beschreibungen, die durch das Schema sowie durch die Dienstbeschreibung gegeben sind. Damit wäre neben einer manuellen Implementierung auch eine automatische Erstellung der Komponente Description möglich. Zusammengefasst ergeben sich für die Implementierung der Komponente Description drei Ansätze: *(a)* Description wird automatisch erstellt, *(b)* Description ist eine *generische Komponente*, die aus Schema und Dienstbeschreibung die Objekte, die die Eigenschaften repräsentieren, dynamisch erzeugt und die *(c)* manuelle Implementierung von Description. Die ersten beiden Ansätze der Aufzählung sind in Abbildung 5.3 dargestellt. In Abbildung 5.3(a) dient das Schema als Eingabe für einen Codegenerator für Description (nicht zu verwechseln mit dem Codegenerator für die Proxies und den Stub). Die Ausgabe des Codegenerators ist Quellcode, beispielsweise eine Klasse, für Description.



(a) generiert (b) generisch

Abbildung 5.3 – *Automatisierte Erstellung von Description*

Eine weitere Möglichkeit ist eine generische Komponente, um Description zu realisieren. Dabei werden, wie in Abbildung 5.3(b), das Schema und die Dienstbeschreibung als Eingabe für die generische Komponente verwendet. Die generische Komponente prüft das Schema gegen die Dienstbeschreibung. Passt die Dienstbeschreibung zum Schema, werden die Objekte, die die Eigenschaften repräsentieren, von der generischen Komponente dynamisch erzeugt. Die generische Komponente ist für beliebige JSON-Dokumente entworfen. Erst mit dem Schema und der Beschreibung für namenszentrische Dienste wird daraus die

Description-Komponente. Eine automatische Erstellung von Description wie in Abbildung 5.3 hat den Nachteil, dass nur die Informationen aus dem Schema und der Dienstbeschreibung herangezogen werden. Dieser Nachteil zeigt sich beispielsweise bei zusammengesetzten Datentypen in der Dienstbeschreibung. JSON-Eigenschaften sind laut Standard ungeordnet. Damit werden zum Beispiel von JSON-Parsern die Eigenschaften in eine andere Ordnung gebracht, als sie im JSON-Dokument haben. Für Methoden in der Dienstbeschreibung ist die Ordnung unproblematisch, für zusammengesetzte Datentypen allerdings nicht, denn die meisten Programmiersprachen wie C oder Java verlangen, dass ein Datentyp vor Benutzung definiert werden muss. Wird die Ordnung der Typen in der beispielhaften Typendefinition der Dienstbeschreibung in Quelltext 5.5 durch den JSON-Parser umgekehrt, also `C`, `B`, `A`, und würde diese Reihenfolge auch so bei der Codegenerierung angewendet, wäre das problematisch, weil Typen `A` und `B` vor `C` definiert sein müssen.

```
1  "types": {
2    "A": {
3      "v": { "type": "u8" }
4    },
5    "B": {
6      "v": { "type": "u8" }
7    },
8        "C": {
9      "v1": { "type": "A" },
10     "v2": { "type": "B" }
11   },
12 },
```

QUELLTEXT 5.5 – *Zusammengesetze Datentypen in einer Dienstbeschreibung für namenszentrische Dienste*

Eine Möglichkeit, wie man das Problem der Ordnung in JSON beheben könnte, wäre die explizite Angabe einer Ordnung mit einer Eigenschaft, wie es beispielsweise bei JSON-WSP vorgeschlagen wird. Allerdings würde eine Eigenschaft zur Angabe der Ordnung im Widerspruch zu einer schlanken Dienstbeschreibung für das Internet der Dinge stehen. In der Tat braucht man die Angabe der Ordnung nicht, denn es ist möglich, die Ordnung aus der Definition der Datentypen abzuleiten. Genau diese Ableitung der Ordnung ist bei der automatischen Erstellung von Description, so wie sie oben beschrieben wurde, nicht möglich. Daher wurde in dieser Arbeit die Description-Komponente manuell erstellt und dort eine Ableitung der Ordnung für die Datentypen implementiert.

Description wird in dieser Arbeit als Paket implementiert. Das heißt, dass sich Description aus mehreren, voneinander abhängigen Klassen zusammensetzt. Die Klassen von Description spiegeln den durch das Schema festgelegten Aufbau der Dienstbeschreibung wieder. Die Klasse in dem Paket, die die Schnittstelle für die Codegeneratoren implementiert heißt `Document`. `Document` repräsentiert somit den Inhalt zwischen der ersten geöffneten geschweiften und der letzten schließenden geschweiften Klammer der Dienstbeschreibung. Klasse `Document`

benutzt einen JSON-Parser um das JSON-Dokument der Dienstbeschreibung einzulesen und die von dort eingelesenen Informationen in Datenstrukturen zu speichern. Für jede Eigenschaft der Dienstbeschreibung gibt es in `Document` eine Methode, um die Informationen aus der Dienstbeschreibung abzurufen. Die Eigenschaften `types` und `methods` der Dienstbeschreibung werden in Description von den Klassen `Types` beziehungsweise `Methods` repräsentiert. Auch diese Klassen haben Methoden, um auf die Eigenschaften von `types` und `methods` zuzugreifen. Das UML-Klassendiagramm in Abbildung 5.4 zeigt den Aufbau von Description. Die Klassen `Document`, `Types` und `Methods` sind im Paket `Description` zusammengefasst. Die Assoziationspfeile zeigen von `Document` hin zu `Types` und `Methods`, da `Document` die beiden Klassen beim Einlesen des JSON-Dokuments der Dienstbeschreibung erzeugt. Die Klassen `Types` und `Methods` bieten Iteratoren an, um auf die in der Dienstbeschreibung definierten Datentypen und Methoden zuzugreifen. Klasse `Types` implementiert die oben beschriebene Ordnung der Datentypen. Beim Zugriff über den Iterator sind die Datentypen bereits in der richtigen Reihenfolge. Datentypen und Methoden sind wiederum als Klassen implementiert. Deren Methoden wiederum erlauben den Zugriff auf die jeweiligen Eigenschaften von Datentypen und Methoden, die in der Dienstbeschreibung definiert sind. Implementierungsdetails der Datentypen und Methoden in Description werden im Anhang in Abschnitt A.3.3 dargestellt.

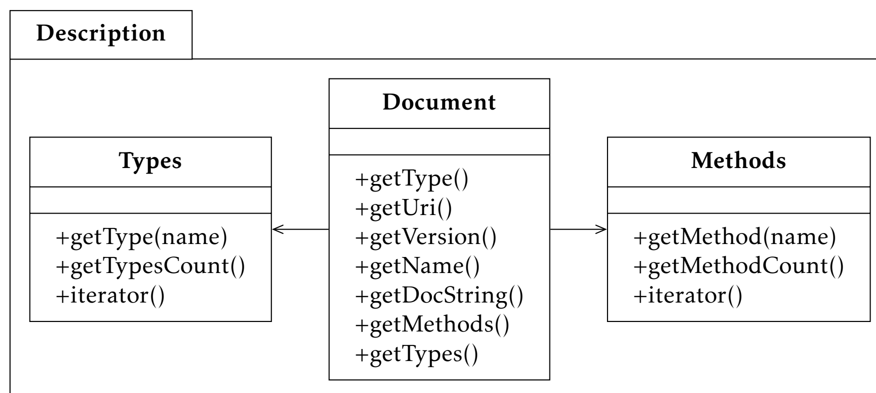


Abbildung 5.4 – *Aufbau von Description*

In allen Codegeneratoren, die während der Arbeit entstanden sind, wird die Description-Komponente verwendet. Aus diesem Grunde ist die Ordnung für Datentypen hier implementiert worden. Die Codegeneratoren werden im folgenden Abschnitt vorgestellt.

CODEGENERATOREN

Das in dieser Arbeit implementierte Werkzeug zur Codegenerierung ist in der Lage, Code für Dienstanbieter und Dienstnutzer zu generieren. Es enthält somit mehrere Codegeneratoren: einen *(i)* Dienstanbieter-Proxy-Generator, einen *(ii)* Dienstanbieter-Stub-Generator und einen *(iii)* Dienstnutzer-Proxy-Generator. Alle Codegeneratoren verwenden die Description-Komponente. Abbildung 5.5 zeigt in dem UML-Klassendiagramm den prinzipiellen Aufbau des Werkzeugs

zur Codegenerierung mit den drei oben aufgezählten Komponenten für die Codegeneratoren, die unter der Description-Komponente angeordnet sind. Für die drei Komponenten im Klassendiagramm wird die englische Bezeichnung sowie die bei Klassennamen übliche Notation mit Binnenmajuskeln verwendet. Der Assoziationspfeil zeigt an, dass die Codegeneratoren die Description-Komponente benutzen. Die Komponenten sind, in Abhängigkeit der Implementierung des Werkzeugs, als Klassen oder Module ausgeführt.

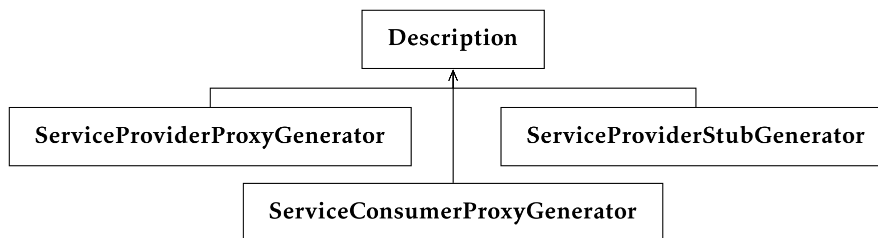


Abbildung 5.5 – *Prinzipieller Aufbau eines Werkzeugs zur Codegenerierung*

Die vom Dienstnutzer-Proxy-Generator generierten Proxies erstellen Anfragen mit den entfernten Methodenaufrufen und versenden diese. Generierte Dienstanbieter-Proxies empfangen diese Anfragen und rufen daraufhin Methoden beim Dienstanbieter auf. Die serialisierten Methoden in den Nachrichten sind Teil des Namens, der wiederum Bestandteil jeder inhaltszentrischen Nachricht ist. Sind die Methodennamen lang, führt das zu langen Namen und das wiederum zu langen Nachrichten, die wiederum eine Herausforderung für ressourcenbeschränkte Geräte im Internet der Dinge darstellen (vgl. Herausforderungen bei der Anwendung des inhaltszentrischen Ansatzes im Internet der Dinge in Abschnitt 3.2.1).

Die in dieser Arbeit entwickelten Codegeneratoren generieren Proxies, die das Problem der langen Methodennamen lösen, indem sie die Methodennamen durch kürzere Hashwerte ersetzen. Dazu ist es notwendig, dass Dienstanbieter- und Dienstnutzer-Proxy die gleichen Methoden auf die jeweiligen Hashes abbilden, denn der Methodenaufruf wird beim Dienstnutzer-Proxy in einen Hash umgewandelt und muss entsprechend beim Dienstanbieter-Proxy zum richtigen Methodenaufruf führen. Ein Dienstanbieter- und ein dazu passender Dienstnutzer-Proxy haben als Grundlage eine Dienstbeschreibung und das Werkzeug generiert daraus Proxies, die aufgrund der Dienstbeschreibung auch gleiche Hashes für die Methoden generieren. Ein generierter Dienstnutzer-Proxy ersetzt einen Methodennamen bei einer Anfrage in einen Hashwert. Abbildung 5.6 verdeutlicht das Prinzip der generierten Proxies, die Methodennamen durch Hashes ersetzen.

In Abbildung 5.6 werden zuerst die Proxies von einer Dienstbeschreibung erstellt. In der Dienstbeschreibung wird eine Methode mit dem Namen `method_name()` beschrieben. Der Dienstnutzer-Proxy-Generator erstellt Code, der `method_name()` auf den Hash 3752 abbildet und der Dienstanbieter-Proxy-Generator entsprechend Code, den Hash 3752 wieder auf `method_name()` abbildet. (Methodennamen werden einfachheitshalber hier ohne führende Command-Marker oder Typ-Längen-Felder dargestellt.) Wird am Dienstnutzer die Methode

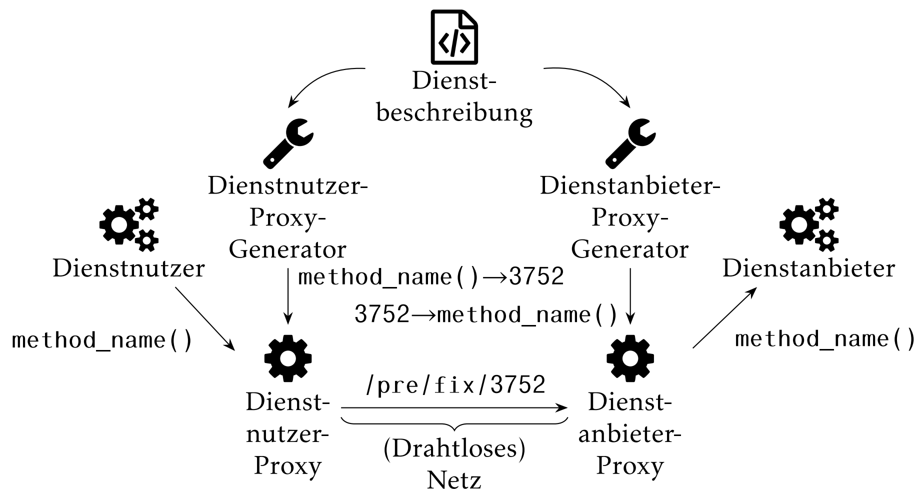


Abbildung 5.6 – *Caption*

method_name() aufgerufen, so wandelt der Dienstnutzer-Proxy den Methodennamen in den Hash um. Dieser Hash steht anstatt des langen Methodennamens im Namen des Interests (hier /pre/fix/method_name), der über das (drathlose) Netz verschickt wird. Der Dienstanbieter-Proxy erkennt den Hash und ruft entsprechend die Methode mit dem Namen method_name() auf. Angenommen, in diesem Beispiel benötigt der Hash vier Byte und der Methodenname elf Byte, so wurden durch die Maßnahme, Methodennamen durch Hashes zu ersetzen, insgesamt sieben Byte in der Nachricht eingespart.

Zur Generierung der Proxies und des Stub-Codes wurden im Rahmen der Arbeit Codegeneratoren für die Programmiersprachen Java und C++ entwickelt. Der generierte Java-Code baut auf der Java-Schnittstelle zu CCNx der Version 0.7.0 auf, der generierte C++-Code auf der CCN-IoT Implementierung für die Sensorknoten (vgl. Implementierung in Abschnitt 4.3). Implementierungsdetails der Codegeneratoren werden im Anhang in Abschnitt A.3.4 vorgestellt, da diese Details an dieser Stelle zu weit führen würden. Weiterhin werden im Anhang Quellcodebeispiele für Dienstanbieter-Proxy und -Stub vorgestellt, die mit den Codegeneratoren erstellt wurden.

5.3.3 EVALUATION WERKZEUGUNTERSTÜTZUNG

Dieser Abschnitt präsentiert eine quantitative Evaluation der Implementierung der Werkzeugunterstützung. Die Evaluation zeigt, dass die automatische Codegenerierung vorteilhaft ist, da manuelle Erstellung von komplexem Code eingespart wird. Eine Evaluation von Laufzeit, CPU- und Speicherauslastung der Codegenerator-Implementierungen ist hier nicht sinnvoll, da eine Codegenerierung für aktuelle PC-Hardware keine besondere Herausforderung darstellt. Ein Vergleich von manuell erstelltem und generierten Code wird hier ebenfalls nicht betrachtet, da hier keine signifikanten Unterschiede zu erwarten sind. Stattdessen erfolgt ein Vergleich von Dienstbeschreibung mit dem generierten Code. Dazu werden Techniken aus der statischen Codeanalyse wie die *zykloma-*

tische Komplexität sowie die *logischen Codezeilen* (engl. Logical Lines of Code, kurz LLOC) verwendet. Diese Evaluationsmethode wurde bereits in der Veröffentlichung „Tool Chain for Application Development with Name-Centric Services“ [4] (Teubler, Hellbrück) vorgestellt.

KOMPLEXITÄT VON PROXY- UND STUB-CODE

Die zyklomatische Komplexität, beschrieben von Thomas McCabe in „A Complexity Measure“ [106], ist die Anzahl der linear unabhängigen Pfade im Kontrollflussgraphen. Je höher die Anzahl der linear unabhängigen Pfade im Kontrollflussgraphen, desto schwieriger ist es, den Code zu verstehen. Bei der Betrachtung der Komplexität des generierten Codes treten die Methoden zur Serialisierung und Deserialisierung in den Proxies besonders hervor, da sie den Großteil der Funktionalität bereitstellen. Aus diesem Grunde wird im Folgenden die *maximale* (zyklomatische) Komplexität betrachtet. Die maximale Komplexität ist die Komplexität der komplexesten Methode. Je höher die maximale Komplexität, desto höher ist die Komplexität des jeweiligen Proxies.

Als weitere Metrik wird die *durchschnittliche Tiefe* betrachtet. Die durchschnittliche Tiefe ist die Summe der Anweisungen, gewichtet mit der Einrückungstiefe geteilt durch die Summe der Anweisungen (siehe hierzu „Verification and Validation for Modeling and Simulation“ von Jeffrey Strickland [158]). Je höher die durchschnittliche Tiefe, desto mehr Anweisungen gibt es in verschachtelten Blöcken und damit ist der Code schlechter les- und wartbar.

Für die Evaluation wurde Beispielcode für Dienstanbieter und Dienstnutzer generiert. Dienstanbieter und Dienstnutzer definieren jeweils einen zusammengesetzten Datentyp mit einem Feld. Jede Methode hat einen Parameter und gibt den zusammengesetzten Datentyp zurück. Die Untersuchung wurde für eine, drei und fünf Dienstmethoden durchgeführt.

Tabelle 5.2 zeigt die maximale Komplexität und die durchschnittliche Tiefe des generierten Codes. Die Abkürzungen DA und DN in der Tabelle stehen für Dienstanbieter beziehungsweise Dienstnutzer. Weiterhin wird in der Tabelle noch nach C++- und Java-Code unterschieden. Der C++-Code wird für die Implementierung von namenszentrischen Diensten auf den Sensorknoten verwendet, der Java-Code für die Implementierung auf den Gateways.

Nach Steve McConnell [107] liegen die oberen Grenzen für gut zu wartenden Code bei 8 für die maximale Komplexität und 1,0 für die durchschnittliche Tiefe. Wie aus Tabelle 5.2 hervorgeht, liegt insbesondere der C++-Code ab drei Dienstmethoden deutlich über den oberen Grenzen für gut zu wartenden Code.

Die Komplexität des Java-Codes ist konstant. Aus diesem Grunde nimmt auch die durchschnittliche Tiefe bei steigender Anzahl von Dienstmethoden ab, da mehr Anweisungen gibt, die auf gleicher Tiefe sind. Obwohl die Metriken für den Java-Code näher an den oberen Grenzen für gut zu wartenden Code sind, gilt auch hier, dass mit steigender Anzahl an Dienstmethoden die Anzahl der logischen Codezeilen zunimmt, wie im Folgenden gezeigt wird.

Anzahl Methoden	Maximale Komplexität				Durchschnittliche Tiefe			
	C++		Java		C++		Java	
	DA	DN	DA	DN	DA	DN	DA	DN
1	4	5	6	7	1,24	1,36	1,56	1,73
3	10	11	6	7	1,81	1,78	1,32	1,54
5	16	17	6	7	2,15	1,99	1,10	1,38

Tabelle 5.2 – *Maximale Komplexität und durchschnittliche Tiefe des generierten C++- und Java-Codes für Dienstanbieter (DA) und Dienstnutzer (DN). Obere Grenzen für gut zu wartenden Code: maximale Komplexität=* 8, *durchschnittliche Tiefe=* $1,0$

KOMPLEXITÄT DER DIENSTBESCHREIBUNG

Einer der Gründe für die Verwendung der Dienstbeschreibung ist, dass die Beschreibung weniger komplex ist als der daraus generierte Code. Damit vereinfacht die hier vorgeschlagene Lösung der Dienstbeschreibung und Werkzeugunterstützung die Entwicklung von namenszentrischen Diensten.

Wie in Abschnitt 5.2 beschrieben, ist die Dienstbeschreibung in JSON-Syntax. Die JSON-Syntax ist eine Untermenge der Programmiersprache JavaScript und wird als Datenaustauschformat eingesetzt. Die Dienstbeschreibung ist kein Programm, da sie keine Ausdrücke und keinen Kontrollfluss hat. Eine Anwendung der oben genannten Code-Metriken ist somit nicht möglich, da beispielsweise die für die zyklomatische Komplexität konstant der Wert Eins ermittelt werden würde.

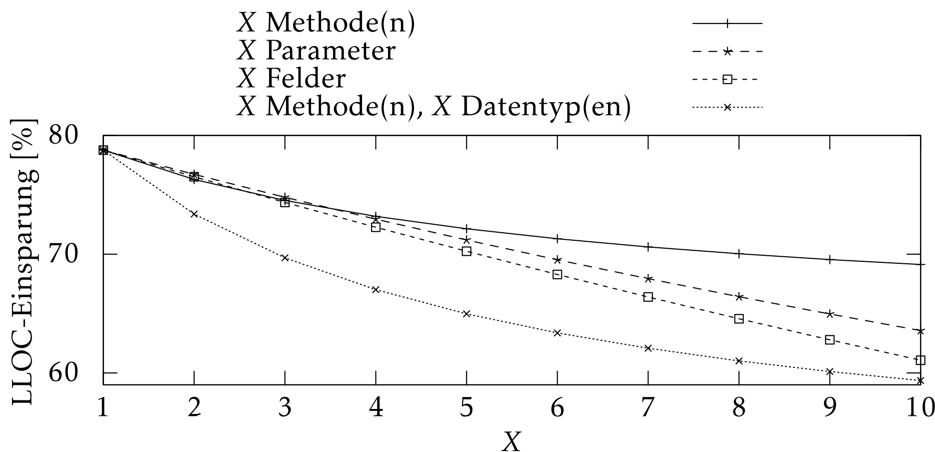
Aus diesem Grunde wird hier die Metrik der logischen Codezeilen (LLOC) verwendet, um den generierten Code mit der Dienstbeschreibung zu vergleichen. Bei der Betrachtung der LLOC wird für jede atomare Anweisung im Code eine Zeile vorgesehen. Die LLOC-Metrik wird hier verwendet, um zu zeigen, wie viele logische Codezeilen durch die Dienstbeschreibung im Vergleich zum generierten Code eingespart werden.

Für die Evaluation mit der LLOC-Metrik werden Beispiel-Dienstbeschreibungen erstellt. Aus jeder Beispiel-Dienstbeschreibung wird mit der Werkzeugunterstützung Code generiert. Dann wird die Anzahl der logischen Codezeilen jeder einzelnen Dienstbeschreibung ermittelt und mit der Anzahl der logischen Codezeilen des dazugehörigen generierten Codes verglichen. Eine Dienstbeschreibung hat weniger logische Codezeilen als der daraus generierte Code. Damit spart die Dienstbeschreibung logische Codezeilen gegenüber dem generierten Code ein. Diese Einsparungen werden in der Evaluation in Prozent angegeben und nach der Formel $\left(1 - \#LLOC_{Desc}/\#LLOC_{Code}\right) \cdot 100$ berechnet, wobei $\#LLOC_{Desc}$ die Anzahl der logischen Codezeilen der Dienstbeschreibung und $\#LLOC_{Code}$ die Anzahl der logischen Codezeilen des generierten Codes ist. Je größer die Einsparung, desto effizienter ist es, eine Dienstbeschreibung zu erstellen und den Code für Proxy und Stub mit der Werkzeugunterstützung zu generieren, da der generierte Code sonst manuell hätte erstellt werden müssen.

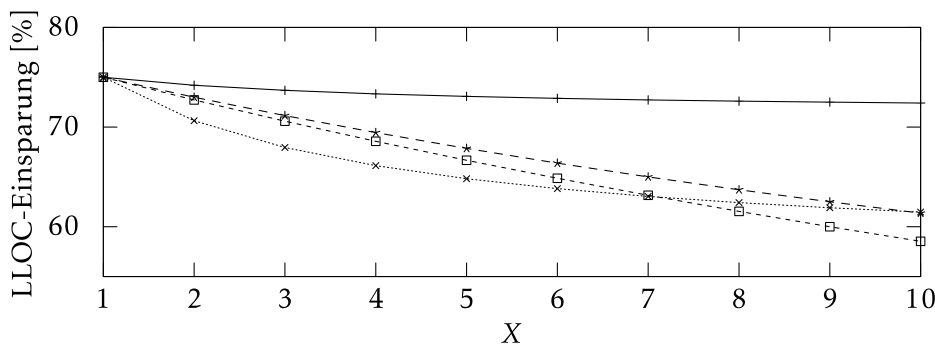
Die Einsparungen an logischen Codezeilen für den generierten Java- und C++-Code sind in Abbildung 5.7 exemplarisch für Dienstanbieter-Proxies dargestellt.

Auf der x-Achse in Abbildung 5.7 ist die Anzahl der korrespondierenden JSON-Objekte aufgetragen. Das „*X*“ in der Legende der Plots bezieht sich auf die Anzahl der JSON-Objekte. Jede Kurve in den Plots bezieht sich dabei auf JSON-Objekte der Dienstbeschreibung mit der Anzahl *X*. Für *X* sind Werte von eins bis zehn auf der x-Achse aufgetragen. In der Praxis werden im Internet der Dinge Dienste mit mehr als zehn Methoden, Methoden mit mehr als zehn Parametern oder Rückgabetypen mit mehr als zehn Feldern eher selten erwartet. Jede Kurve variiert nur die Anzahl bestimmter JSON-Objekte in der Dienstbeschreibung, die Anzahl der anderen JSON-Objekte wird auf eins gesetzt.

Zusammengesetze Datentypen werden nur als Rückgabewerte verwendet. Wenn die Anzahl der zusammengesetzten Datentypen erhöht wird, ist es daher sinnvoll, gleichzeitig die Anzahl der Methoden zu erhöhen, da es sonst ungenutzte zusammengesetzte Datentypen in der Dienstbeschreibung gäbe.



(a) C++



(b) Java

ABBILDUNG 5.7 – *LLOC-Einsparungen der Dienstbeschreibung gegenüber dem (generierten) Quellcode für den Dienstanbieter-Proxy*

Auf der y-Achse sind die Einsparungen an logischen Codezeilen in Prozent aufgetragen. Beim C++-Code in Abbildung 5.7(a) werden bei steigender Methodenanzahl noch 69 % logische Codezeilen eingespart, was immer noch viel

ist. Kommen neben den Methoden noch die zusammengesetzten Datentypen dazu, so werden nach zehn Methoden und Datentypen immer noch um die 59 % logische Codezeilen mit der Dienstbeschreibung gegenüber dem generierten Code eingespart.

Die Einsparungen werden für die *Parameter* der Methoden und der *Felder* der zusammengesetzten Datentypen in Abbildung 5.7 geringer. Das ist nicht verwunderlich, da Parameter und Felder in der Dienstbeschreibung und im generierten Code in ähnlicher Gestalt vorkommen. So ist eine Strukturvariable in C++ oder eine einfache Klasse in Java ähnlich aufgebaut wie ein zusammengesetzter Datentyp in der Dienstbeschreibung.

Abbildung 5.7(b) zeigt die Einsparungen an logischen Codezeilen der Dienstbeschreibung gegenüber dem generierten Java-Code. Hier ergibt sich ein ähnlicher Verlauf wie in Abbildung 5.7(a), wobei die LLOC-Einsparungen zwischen einer und zehn Methoden weniger stark abfällt als beim C++-Code. Dieser weniger stark abfallende Verlauf kommt daher, dass der Java-Code einen geringeren Overhead an logischen Codezeilen als der C++-Code hat. So ist in Abbildung 5.7(a) bei einer Methode der Anteil des Overheads am gesamten Code noch relativ hoch, nimmt aber mit steigender Methodenanzahl ab. Beim Java-Code in Abbildung 5.7(b) werden bei steigender Methodenanzahl 72 % logische Codezeilen eingespart. Werden mit den Methoden auch die zusammengesetzten Datentypen erhöht, gehen die Einsparungen im Verlauf wie beim C++-Code um etwa 10 % auf 62 %zurück.

Die Werkzeugunterstützung zusammen mit der Dienstbeschreibung hilft Entwicklern von namenszentrischen Diensten, manuelle Codeerstellung einzusparen. Entwickler erstellen anstatt der Proxies und der Stub-Codes nur die Dienstbeschreibung manuell. Proxies und Stub-Codes werden dann durch Codegeneratoren erstellt. Für die Erstellung der Dienstbeschreibung sind weniger logische Codezeilen als für Proxies und Stub-Codes notwendig, woraus eine Arbeitsersparnis für Entwickler resultiert. Die Ergebnisse in Abbildung 5.7 beziehen sich auf die aktuelle Implementierung von CCN-IoT beziehungsweise der Java-Schnittstelle zu CCNx. Damit die Messungen in Abbildung 5.7 aussagekräftig sind, vermeiden die Implementierungen Wiederholungen durch Kapselung von Funktionalitäten. Gäbe es Wiederholungen im Code, wären auch die Einsparungen von logischen Codezeilen höher, da mehr Code generiert werden müsste, um die gleiche Funktionalität zu realisieren.

5.4 ZUSAMMENFASSUNG

Dieses Kapitel stellte die Dienstbeschreibung und Werkzeugunterstützung der namenszentrischen Dienste vor. Eine Dienstbeschreibung ist ein strukturiertes Dokument, welches die Schnittstelle eines Dienstes beschreibt. Für die Strukturierung der Dienstbeschreibung wurde das JSON-Format gewählt. Der Aufbau einer Dienstbeschreibung ist in einem JSON-Schema definiert. Das JSON-Schema ist eine generische Beschreibung der Dienstbeschreibung. Mit der Werkzeugunterstützung werden aus der formalen Dienstbeschreibung Dienstanbieter- und Dienstnutzer-Proxies sowie Stub-Code automatisch generiert.

Ziel von Dienstbeschreibung und Werkzeugunterstützung ist es, dem Entwickler die Aufgaben wie Serialisierung und Deserialisierung von Nachrichten abzunehmen. Dies ist insbesondere bei den namenszentrischen Diensten wichtig, da es sich um ein neues Paradigma handelt, das Umdenken und Einarbeitung von den Entwicklern erfordert (vgl. Dienstentwicklung in Abschnitt 3.2.1 auf Seite 39). Wie in Kapitel 7 gezeigt wird, haben die Entwickler trotz der Abstraktion durch die namenszentrischen Dienste weiterhin Zugriff auf CCN-spezifische Merkmale, wie Nachrichten und Namen.

Mit der Evaluation der Werkzeugunterstützung in diesem Kapitel wurde gezeigt, dass sich die Werkzeugunterstützung in zweierlei Hinsicht lohnt: So wurde zum einen gezeigt, dass der Code für die Proxies komplex ist. Geht man des Weiteren davon aus, dass die Dienstbeschreibung von Entwicklern manuell erstellt wird, so ergibt sich eine Arbeitsersparnis. Die Arbeitsersparnis besteht darin, dass Dienstbeschreibungen, die in der Praxis erwartet werden, ungefähr 60 bis 70 % weniger logische Codezeilen als der Code für die Proxies hat, der ohne Werkzeugunterstützung manuell erstellt werden müsste.

KAPITEL 6

NAMEN FÜR DIENSTE

NAMEN sind ein wesentliches Element von namenszentrischen Diensten. Sie adressieren Dienste und realisieren entfernte Methodenaufrufe. Bisher wurde nur die Syntax und der Aufbau von Namen in den Grundlagen in Abschnitt 2.4.1 sowie das Konzept der Dienstmethoden in Abschnitt 3.3.2 behandelt. Der Beitrag dieses Kapitels ist Vergabe von Namen beziehungsweise der Präfixe für Dienste. Das Konzept basiert auf einem *Lebenszyklusmodell* für Dienste, das in „A Solution for the Naming Problem for Name-Centric Services“ [5] (Teubler et al.) erstmals vorgestellt wurde.

Weiterhin wird in dem Kapitel der Einsatz von Gateways in inhalts-/namenszentrischen Internet der Dinge behandelt. Gateways sind ein Teil der Architektur des inhalts-/namenszentrischen Internet der Dinge, wie in Abbildung 4.1 auf Seite 66 dargestellt. In diesem Kapitel wird gezeigt, wie auf Gateways Namen für Dienste vergeben oder wie mit Diensten auf Gateways eine zustandsbehaftete Lösung zur Verkürzung von Namen ermöglicht wird.

Das Kapitel baut sich wie folgt auf: Verwandte Arbeiten werden in Abschnitt 6.1 vorgestellt. In Abschnitt 6.2 wird das Lebenszyklusmodell für namenszentrische Dienste eingeführt, das die Grundlage für das Konzept für Namen in Abschnitt 6.3 bildet. In Abschnitt 6.4 werden spezielle Namen und Vorschläge für standardisierte Namen eingeführt. Namen und Gateways werden in Abschnitt 6.5 vor der Zusammenfassung des Kapitels in Abschnitt 6.6 behandelt.

6.1 VERWANDTE ARBEITEN

Bei Content Centric Networking dienen Namen zu Adressierung von Daten. Daten in inhaltszentrischen Netzen lassen sich auch anders adressieren. „Directed Diffusion“ für inhaltszentrische drahtlose Sensornetze von Intanagonwiwat et al. [83] nutzt Name-Wert-Paare, um Daten semantisch zu annotieren und zu adressieren. Ein ähnlicher Mechanismus, Labeled Segment genannt, ist in CCNx 1.0 eingeführt worden (siehe „CCNx 1.0 Protocol Specifications

Roadmap“ von Marc Mosco [112]). Bei Labeled Segment URIs [113] wird den *Segmenten* – Segment ist ein neuer Terminus für Komponenten in CCNx 1.0 – ein Label (dt. Bezeichner) zugewiesen. Das Label für ein Segment wird dabei vor die Bezeichnung des Segments gestellt und mit einem Gleichheitszeichen (=) von diesem getrennt. Der Name `/name=foo/name=bar` ist ein Beispiel für einen Namen aus zwei gelabelten Segmenten, die Label sind bei beiden Segmenten `name`, was nach CCNx 1.0 das Standard-Label ist und üblicherweise weggelassen wird. Somit sind `/name=foo/name=bar` und `/foo/bar` äquivalent. Der Name `/foo/app=bar` ist verschieden von `/foo/bar`, da das Label des zweiten Segments `app` und nicht `name` ist. Labeled Segment URIs erlauben es, zwischen Ressourcen mit ähnlichen Namen zu unterscheiden, da Segmenten damit eine Semantik zugeordnet wird. Labeled Segment URIs sind der Ersatz für Command Marker in CCNx 1.0 und sind laut Mosco flexibler als Command Marker. Ein Nachteil von Labeled Segment URIs ist, dass die Namen durch gelabelte Segmente/Komponenten noch größer werden und somit das Problem der knappen Ressourcen im Internet der Dinge verschärfen.

Das in diesem Kapitel vorgeschlagene Lebenszyklusmodell für namenszentrische Dienste weist einzelnen oder aufeinanderfolgenden Komponenten, den Namensteilen, eine Bedeutung zu. Weiterhin werden in diesem Kapitel Standardnamen vorgeschlagen. Einen ähnlichen Ansatz verfolgen Braun et al. in „Service-Centric Networking“ [50] mit ihrem konsistenten Namensschema. Bei Braun et al. finden sich Beispiele für Namen, die Inhalt und Dienste adressieren. Eine Klassifikation von Namensteilen wie in diesem Kapitel findet bei Braun et al. nicht statt. Weiterhin gibt es bei Braun et al. keinen Hinweis auf einen Konfigurationsmechanismus für Namen. Ein Konfigurationsmechanismus für Namen ist wichtig, da eine manuelle Konfiguration der zahlreichen Geräte und Dienste im Internet der Dinge aufwändig ist.

Konfigurationsmechanismen sind kein Ersatz für Konventionen. Die Konvention, DNS-Namen als Präfix für Namen in CCNx zu verwenden, wurde mit den ersten Entwürfen von CCNx eingeführt (z. B. bei Jacobson et al. [86]). DNS ist eine Abkürzung für Domain Name System [109], einem verteilten Verzeichnisdienst für Domainnamen im Internet. Domainnamen werden von sogenannten Registries an Organisationen, Institutionen, Unternehmen oder Einzelpersonen weltweit eindeutig vergeben. Beispiele hierfür sind `/ietf.org` oder `/fh-luebeck.de`. Neben der Empfehlung für Domainnamen als Präfix existierten in ersten CCNx-Versionen die oben erwähnten Command Marker. Command Marker geben Komponenten eine Bedeutung, wie etwa die Bedeutung eines entfernten Methodenaufrufs (vgl. CCNx-Kommando in Abbildung 3.5 auf Seite 47).

Im Named Data Networking (NDN) Projekt von Zhang et al. [178] werden ebenfalls Domainnamen als Präfix vorgeschlagen. Bei NDN wird die hierarchische Struktur des Internets konsequent auf die hierarchische Struktur des Namens abgebildet. So bilden die Top Level Domain (TLD), gefolgt von einem weltweit eindeutigen Domainnamen, den Präfix. Ist `de` die TLD und `fh-luebeck` der Domainname, so ist der Präfix bei NDN `/de/fh-luebeck` (und nicht `/fh-luebeck.de`). Für Dienste, die aus dem inhalts-/namenszentrischen Internet erreichbar sind, wird in dieser Arbeit der Präfix wie bei NDN gewählt, da dieser Präfix der Hierarchie des Internets folgt.

6.2 LEBENSZYKLUSMODELL

Im Verlauf der Arbeit stellte sich die wichtige Frage, wie Namen für Dienste vergeben werden. Für die letzte Komponente im Namen, die Dienstmethode, lässt sich die Frage leicht beantworten: der Entwickler eines Dienstes beziehungsweise der Dienstbeschreibung vergibt die Namen für Methoden. Der Begriff Entwickler bezeichnet hier entweder eine Einzelperson, ein Unternehmen oder eine Organisation. Um weitere Personen oder Organisationen zu identifizieren, die an der Vergabe von Namen beteiligt sind, bietet es sich an, einen beispielhaften Softwarelebenszyklus zu betrachten: Nimmt man zum Beispiel den Lebenszyklus einer netzwerkbasierten Anwendungssoftware, die auf Büro-PCs einer Firma oder Organisation zum Einsatz kommt, so sind von der Entwicklung bis zum Einsatz der Software verschiedene Personen oder Organisationen beteiligt. In dem Beispiel entwickelt im ersten Schritt ein Softwarehersteller die Anwendungssoftware. Diese Software wird im zweiten Schritt von einem IT-Dienstleister auf den Rechnern der Organisation, die die Anwendungssoftware einsetzen möchte, installiert und gegebenenfalls nach den Anforderungen der Endanwender konfiguriert. Im dritten Schritt wird der mit der Anwendungssoftware installierte Rechner in das Netzwerk der Organisation integriert. Zusammenfassend besteht der oben aufgeführte Lebenszyklus aus drei Schritten, oder um in der Terminologie des Softwarelebenszyklus [65] zu bleiben, aus drei *Phasen*.

Diese Phasen für den Lebenszyklus lassen sich auch auf namenszentrische Dienste im Internet der Dinge übertragen. In der *ersten Phase* vergibt der Entwickler des namenszentrischen Dienstes die Namen für Methoden. Methodennamen und die Signatur der Methode werden in der Dienstbeschreibung und in der Implementierung definiert. Die Dienstbeschreibung wird mit der Software des Dienstes ausgeliefert. Die *zweite Phase* umfasst die Ausbringung der Dienste auf die Hardware. In dieser Phase werden die Dienste in Kategorien gruppiert. Die *dritte Phase* ist die sogenannte *Dienstintegration*. Bei der *Dienstintegration*, bringt die Organisation, die die namenszentrischen Dienste im Internet der Dinge anbieten möchte, die Hardware aus. Anschließend konfiguriert die Organisation ein (oder mehrere) Gateway(s), um die auf der Hardware ausgeführten Dienste in das drahtlose Netz der Organisation und damit auch in das Internet zu integrieren. In jeder der Phasen wird ein Abschnitt des Namens hinzugefügt.

Im nächsten Teil des Kapitels wird ein Konzept eingeführt, welches die Zuweisung der Namensteile zu den verschiedenen Phasen beinhaltet. Abbildung 6.1 fasst die drei Phasen des Lebenszyklusmodells (*i*) Entwicklung, (*ii*) Ausbringung und (*iii*) Integration zusammen. Sie sind in zeitlicher Abfolge von links nach rechts dargestellt.

6.3 KONZEPT FÜR NAMEN

Das Ziel des Konzeptes ist es, den Teilen des Dienstnamens eine Bedeutung oder Semantik zuzuweisen. Als Teil eines Namens werden im Folgenden eine Komponente oder mehreren aufeinanderfolgende Komponenten eines Namens bezeichnet. Das Konzept für Dienstnamen wird in diesem Abschnitt an einem Beispiel erklärt.

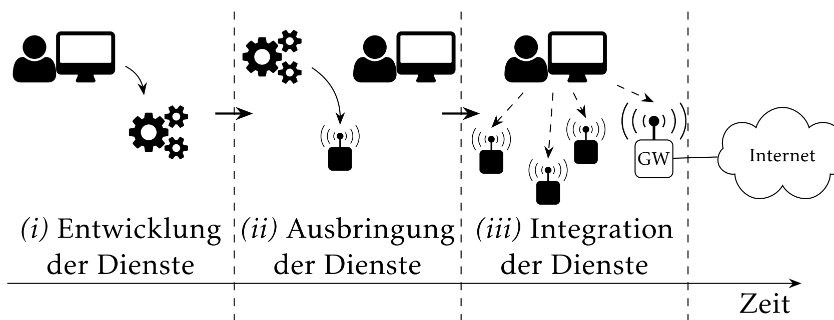


ABBILDUNG 6.1 – *Lebenszyklusmodell für namenszentrische Dienste im Internet der Dinge*

In dem Beispiel wird, der Reihenfolge der Phasen in Abbildung 6.1 folgend, der Name von hinten aufgebaut. Die letzte Komponente eines Dienstnamens ist die Methode, hier im Beispiel `/get_temp`. Methode `/get_temp` gibt den aktuellen Wert eines Temperatursensors zurück. (Methodennamen werden einfachheitshalber hier ohne führende Command-Marker oder Typ-Längen-Felder dargestellt.)

Als Präfix der Methode werden in dieser Arbeit eine oder mehrere Komponenten vorgeschlagen, die die Kategorie des Dienstes beschreiben. In dem Beispiel ist es die Komponente `/climate`. Sie beschreibt klimabezogene Dienste, wie Temperatur und Feuchtigkeit. Die Komponente oder die Komponenten für die Kategorie werden in der zweiten Phase, der Ausbringung der Dienste auf die Hardware, festgelegt.

Als Präfix vor der Kategorie kommt die sogenannte *Site*, in diesem Beispiel `/building18/floor2/room14`. In diesem Beispiel adressiert die Site Dienste, die sich auf einem Campus im Gebäude 18, im zweiten Stock, Raumnummer 14 befinden. Eine Site ist eine generische Adresse und adressiert nicht notwendigerweise einen Ort, wie in dem Beispiel. Der Campus wird durch den global eindeutigen Domainnamen `/de/fh-luebeck` adressiert. In diesem Beispiel gehört der Campus zur Fachhochschule Lübeck. Site und Domainnamen werden in der dritten Phase, der Integration der Dienste in das inhalts-/namenszentrische Internet, gesetzt.

Domainname, Site, die Kategorie des Dienstes und die Methode (in der Reihenfolge) bezeichnen in dieser Arbeit den voll qualifizierten Dienstnamen (engl. *Fully-Qualified Service Name*, kurz FQSN). Diese Bezeichnung ist im Rahmen der Arbeit in Anlehnung an den *Fully-Qualified Host Name* (FQHN) aus RFC 1153 [168] entstanden. Der FQHN benennt einen Host in einem knotenzentrischen IP-Netz eindeutig. Da bei den namenszentrischen Diensten nicht die Knoten bei der globalen Adressierung im Internet betrachtet werden, sondern die Dienste, wird der Begriff „Host“ durch „Service“ ersetzt.

Jede Phase des Lebenszyklus in Abbildung 6.1 ist mit einem Abschnitt des Namens assoziiert. Diese Assoziation wird in Abbildung 6.2 dargestellt. Die Abbildung zeigt die drei Abschnitte des Dienstnamens an einem Beispiel. Für jeden Abschnitt des Namens wird die Phase aus Abbildung 6.1 angegeben, in der er hinzugefügt wird.

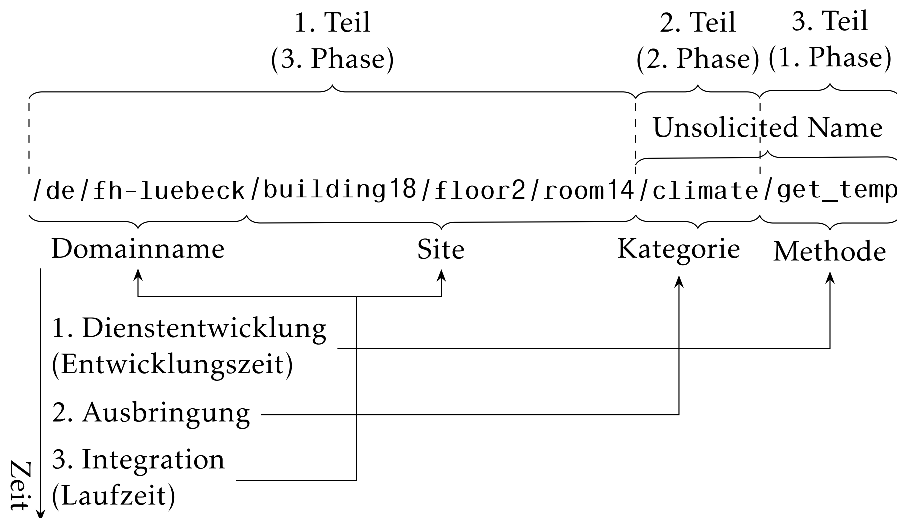


Abbildung 6.2 – *Aufteilung des Dienstnamens an einem Beispiel*

In der ersten Phase, der Entwicklung von Diensten, wird der Methodenname durch den Entwickler oder der Organisation vergeben, die den Dienst entwickelt. Die vom Entwickler unter Benutzung der Werkzeugunterstützung in der ersten Phase vergebenen Namensbestandteile werden in der Arbeit als *dritter Teil* (des Namens) bezeichnet.

Während der Ausbringung auf die Hardware werden die Dienste in Kategorien eingeteilt. Die Namen der Kategorien bilden den als *zweiten Teil* bezeichneten Namensbestandteil des Dienstes. Sind erster und zweiter Teil eines Dienstnamens gesetzt, ist der Dienst im lokalen Netz erreichbar, aber nicht im inhalts-/namenszentrischen Internet. Dienstnamen, bei denen der zweite und dritte Teil des Namens gesetzt ist, werden als *unsolicited* (dt. ungebeten) bezeichnet. Die Bezeichnung „ungebeten" erklärt sich wie folgt: In einem lokalen Netz gibt es unter Umständen mehrere Instanzen, die denselben Dienst anbieten und bei denen folglich der zweite und der dritte Teil des Namens identisch ist. Jeder Dienstanbieter antwortet auf eine Anfrage mit einem Unsolicited Service Name von einem Dienstnutzer. Nach dem inhaltszentrischen Paradigma ist es so, dass nur eine Antwort den Dienstnutzer erfolgreich erreicht. Von welchem Dienstanbieter die empfangene Antwort kam, lässt sich durch den Dienstnutzer nicht feststellen. Somit handelt es sich aus der Sicht des Dienstnutzers um eine ungebetene Antwort, weil es vielleicht nicht die gewünschte Instanz des Dienstanbieters war, die geantwortet hat. Wie die gewünschte Instanz eines Dienstanbieters erreicht wird, wird im folgenden Abschnitt behandelt.

6.4 NAMEN MIT BESONDERER BEDEUTUNG

Im Kontext der inhaltszentrischen Netze beziehungsweise der namenszentrischen Dienste ist es von Vorteil, Namen mit besonderer Bedeutung einzuführen. In CCN hat beispielsweise hat der *Root Name* eine besondere Bedeutung, da er Präfix aller Namen ist und damit bei der Weiterleitung von Interests eine

besondere Rolle spielt, wie in Abschnitt 6.4.1 erklärt wird. Für die Konfiguration von namenszentrischen Diensten bietet es sich an, einen Namen zu haben, mit dem sich Dienste auf einem bestimmten Gerät ansprechen lassen, wie mit dem *Link-Local Name*, der in Abschnitt 6.4.2 vorgestellt wird. Weiterhin ist es vorteilhaft, *Standardnamen* zu haben, unter denen Inhalte, wie Dienstbeschreibungen, im inhaltszentrischen Netz abgerufen werden können. Standardnamen werden in Abschnitt 6.4.3 vorgeschlagen.

6.4.1 ROOT NAME

Dieser Abschnitt erklärt, warum der Root Name, eingeführt in den Grundlagen in Definition 2.8 auf Seite 27, bei der Weiterleitung von Interests eine besondere Rolle spielt. Damit ist der Root Name ein Name mit besonderer Bedeutung.

Bei der Verarbeitung eines Interests (vgl. Interest-Verarbeitung im Dämon in Abbildung 2.9 auf Seite 30) wird der Interest mit den Einträgen in der Forwarding Information Base verglichen. Der Vergleich beginnt beim ersten Eintrag und endet bei dem letzten Eintrag. Ein Eintrag in der Forwarding Information Base besteht aus einem Namen und einer Information zu den Faces, zu denen ein passender Interest weitergeleitet wird (vgl. Aufbau der Forwarding Information Base in Abbildung 2.8(c) auf Seite 29). Die Einträge in der Forwarding Information Base sind in absteigender Shortlex-Ordnung ihres Namens (vgl. Definition 2.9) sortiert.

Ein Eintrag mit dem Root Name steht ganz am Ende der Forwarding Information Base. Mit diesem Eintrag werden alle Interests weitergeleitet, wenn kein vorheriger Eintrag gepasst hat. Der Eintrag mit dem Root Name in der Forwarding Information Base ist daher vergleichbar mit dem Eintrag des Standardgateways in der IP-Routingtabelle. In IP-basierten Netzen werden alle Nachrichten für Knoten, die nicht im gleichen Subnetz oder für die kein passender Eintrag in der IP-Routingtabelle existiert, zum Standardgateway geschickt. Das Standardgateway versucht dann, die Nachrichten an den Empfänger weiterzuleiten. In dieser Arbeit verweisen Einträge mit dem Root Name auf Faces zu lokalen Kommunikationsdiensten, die zum Beispiel Zugriff auf die Funkschnittstelle haben. Interests werden über diese Faces ebenfalls mit dem Ziel weitergeleitet, einen passenden Content oder Dienstanbieter zu finden.

6.4.2 LINK-LOCAL NAME

Der Link-Local Name löst das in Abschnitt 6.3 angesprochenen Problem, die richtige Instanz eines Dienstes im selben Netz anzusprechen, wenn Dienste noch nicht über einen Fully-Qualified Service Name erreichbar sind. Angenommen, es befinden sich identische Instanzen eines Dienstes, deren Unsolicited Name ebenfalls identisch ist, auf mehreren Knoten im Netz. Diese Dienste antworten, wenn sie mit dem Unsolicited Name angesprochen werden. Um nun einen Dienst auf einem bestimmten Knoten anzusprechen, wird der Link-Local Name verwendet. Der Link-Local Name wurde in „A Solution for the Naming Problem for Name-Centric Services“ [5] (Teubler, Hellbrück) eingeführt.

Beim Link-Local Name wird der Unsolicited Name um die Knotenadresse beziehungsweise um die Adresse der Netzwerkschnittstelle erweitert, wie in Abbildung 6.3 dargestellt.

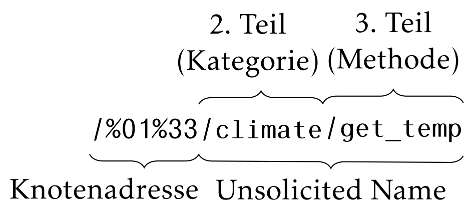


ABBILDUNG 6.3 – *Beispiel für einen Link-Local Dienstnamen*

Im Rahmen der Arbeit wurde die 16 Bit Knotenadresse der Sensorknoten verwendet. Während der Untersuchung war dies ausreichend. Sollten sich inhalts-/ namenszentrische Paradigmen durchsetzen, ist als Standard die 64 Bit Link-Layer Adresse bei 802.15.4 die bessere Wahl.

Auf den ersten Blick scheinen Link-Local Names im Widerspruch zum inhalts-/ namenszentrischen Paradigma zu stehen, da numerische Adressen wie beim knotenzentrischen Paradigma verwendet werden. Nun sind Knoten, wie Dienste, Entitäten im System und daher sollen diese auch mit Namen adressierbar sein. Somit sind die numerische Adressen im Namen kein Widerspruch zum inhalts-/ namenszentrischen Paradigma. Weiterhin sind Link-Local Names wichtig für die Konfiguration von Knoten. Link-Local Names werden in der zweiten Phase des Lebenszyklus für namenszentrische Dienste vergeben. Sie sind dann die einzige Möglichkeit, während der Laufzeit (dritte Phase), einzelne Knoten im Rahmen der Konfiguration gezielt über das inhaltszentrische Netz anzusprechen.

Der Begriff Link-Local Name wurde deswegen gewählt, weil ein Dienstanbieter, der die Anfrage mit einem Namen wie in dem Beispiel in Abbildung 6.3 erhält, diese nur von einem Dienstnutzer aus dem selben – lokalen – Ad-hoc-Netz erhalten hat und nicht aus dem Internet. Die Funktionalität, Anfragen an den Link-Local Name zu verarbeiten, wird von dem Basisdienst mit dem Namen *ID Service* realisiert, der im nächsten Kapitel in Abschnitt 7.4 vorgestellt wird.

6.4.3 STANDARDNAMEN

Wenn man herausfinden möchte, welche Dienste es im Netz gibt, braucht man eine Anlaufstelle, an der man die angebotene Dienste im Netz in Erfahrung bringen kann. Für die Auffindbarkeit von SOAP basierten Diensten wurde ein zentrales Verzeichnis mit dem Namen Universal Description Discovery and Integration (UDDI) [39] vorgeschlagen, welches sich aber nicht durchgesetzt hat. Statt einer Auffindbarkeit über UDDI werden bei SOAP basierten Diensten den Dienstnutzern die Dienstanbieter in der Regel fest zugewiesen. Eine feste Zuweisung von Namen ist auch bei den namenszentrischen Diensten möglich. Damit Anwender oder Entwickler in der Lage sind Site- und Kategorienamen zu erschließen, werden Standardnamen vorgeschlagen, unter denen entsprechende Informationen zu finden sind. Standardnamen werden als bekannt vorausgesetzt.

Standardnamen sind konzeptionell mit den Portnummern von UDP und TCP, den sogenannten Well-known Ports, (RFC 6335) [172] oder der Well-known URI in CoAP (RFC 6690) [150] vergleichbar. Die Well-known Ports identifizieren bekannte Dienste im heutigen Internet. Neben der IP-Adresse benötigt man, um einen entfernten Dienst zu adressieren, die Portnummer des Dienstes. Die Portnummer zum Beispiel für einen Webserver ist standardmäßig 80. CoAP bietet unter dem Präfix `<Servername oder Adresse>/.well-known/core` – wobei `.well-known/core` als wohlbekannt vorausgesetzt wird – ein Dokument an, das Aufschluss über die Dateistruktur des Servers gibt.

Für die Struktur der Präfixe von namenszentrischen Diensten sind ebenfalls wohlbekannte Namen denkbar, durch die sich deren Struktur erschließt. Ein Präfix aus Top Level Domain und Domainnamen, beispielsweise `/de/oceancarrier` für ein (fiktives) Seefrachtunternehmen namens „Oceancarrier", wird als bekannt vorausgesetzt. Bei CoAP wird die IP-Adresse beziehungsweise die URL eines Knotens ebenso als bekannt vorausgesetzt.

Alle Standardnamen, die sich explizit auf namenszentrische Dienste beziehen, beginnen mit dem Präfix `/ncs` Unter der URI `/de/oceancarrier/ncs/site` befindet sich beispielsweise ein Dienst, der die Site-Namen zurückgibt, die zu der Domain gehören. Der Standardname in dem obigen Beispiel ist der Suffix `/ncs/site`. Angenommen, eine Site hat den Präfix `/position`, dann sind weitere Unter-Sites unter dem Dienst mit dem Namen `/de/oceancarrier/position/ncs/site` verfügbar. Eine mögliche Unter-Site von `/position` könnte `/olc` sein. OLC ist eine Abkürzung für Open Location Code, einer kompakten Darstellung für geografische Positionen [129, 87]. Unter `/de/oceancarrier/position/olc` könnte dann der Hinweis kommen, dass die Komponente sich in Abhängigkeit der Position eines Frachtschiffs dynamisch ändert. Unter-Sites von `/olc` aus dynamisch ändernden Open Location Codes sind beispielsweise `/9F6JHCRR+36` oder `/X3P8+XG`. Diese Open Location Codes stehen für den Mittelpunkt eines imaginären Kreises, in dem sich beispielsweise ein Frachtschiff befindet.

Gibt es keine weitere Unter-Site, wird mit dem Suffix `/ncs/ctgy` nach den Kategorien unter der Site gefragt. Der Name für die Anfrage nach den Kategorien würde allerdings nicht `/de/oceancarrier/position/olc/X3P8+XG/ncs/ctgy` lauten, denn eine Komponente in dem Namen ändert sich dynamisch in Abhängigkeit der Position, die Kategorien sind aber unabhängig von der Position. Das Problem der sich dynamisch ändernden Komponenten wird mit Pseudo-Site-Namen umgangen, die als Platzhalter für Unter-Site-Namen eingesetzt werden. Dieser Platzhalter ist der Standardname `/ncs/subs`, wobei die Komponente `/subs` für Sub-Sites (dt. Unter-Sites) steht. Eine Anfrage mit dem Namen `/de/oceancarrier/position/ncs/position/site` liefert auch den Pseudo-Site-Namen `/ncs/subs` zurück, um anzuzeigen, dass die Unter-Sites ab `/position` durch den Pseudo-Site-Namen substituiert werden. Der Name für die Anfrage nach den Kategorien lautet dann `/de/oceancarrier/position/ncs/subs/ncs/ctgy`. In dem Beispiel substituiert `/ncs/subs` alle Unter-Sites von `/position` und somit auch die Unter-Sites `/olc/X3P8+XG` und `/olc/9F6JHCRR+36`.

Mit dem Standardnamen `/ncs/desc`, der als Suffix an die Kategorie angehangen wird, wird die Beschreibung für Dienste unter der Kategorie abgerufen.

Angenommen, eine Kategorie ist `/container`, dann lautet der vollständige Name zum Anfragen der Dienstbeschreibung `/de/oceancarrier/position/ncs/subs/container/ncs/desc`.

Ein weiterer wichtiger Standardname `/ncs/gw`, der Dienste auf Gateways identifiziert. Dieser Standardname ist insbesondere für die automatische Konfiguration der namenszentrischen Dienste wichtig. Die Bedeutung von Gateways und Namen wird im folgenden Abschnitt behandelt. Tabelle 6.1 fasst die oben vorgeschlagenen Standardnamen der namenszentrischen Dienste noch einmal zusammen.

Name	Beschreibung
`/ncs/site`	(Unter-)Site-Namen
`/ncs/subs`	Pseudo-Unter-Site (substituiert Unter-Sites)
`/ncs/ctgy`	Kategorien
`/ncs/desc`	Dienstbeschreibung
`/ncs/gw`	Gateway

TABELLE 6.1 – *Standardnamen für namenszentrische Dienste*

6.5 NAMEN UND GATEWAYS

Damit namenszentrische Dienste in einem drahtlosen Netz aus dem Internet erreichbar sind, brauchen sie einen Fully-Qualified Service Name. Fully-Qualified Service Names haben jedoch das Problem, dass sie zu unerwünscht langen Nachrichten in drahtlosen Netzen führen und Speicher auf ressourcenbeschränkten Geräten verbrauchen. In drahtlosen Netzen müssen lange Namen daher verkürzt werden. Die Vergabe von Präfixen und die Verkürzung von Namen wird mit Diensten auf dem Gateway realisiert: Der *Name Solicitation Service* vergibt Präfixe an Dienste und der *Alias Name Service* verkürzt die vergebenen Präfixe und somit auch die Nachrichten im drahtlosen Netz. Name Solicitation Service und Alias Name Service werden auf Gateways ausgeführt. Beide Dienste helfen dabei, den Übergang zwischen drahtgebundenen und drahtlosen Netzen zu realisieren (vgl. Überblick über die Gesamtarchitektur des inhalts-/namenszentrischen Internets Abbildung 4.1 auf Seite 66).

6.5.1 NAME SOLICITATION SERVICE

Um namenszentrische Dienste im gesamten Internet zu erreichen, werden eindeutige Präfixe, bestehend aus dem ersten und zweiten Teil des Namens, wie bei dem Beispieldienstnamen in Abbildung 6.2 auf Seite 125, benötigt. Der erste Teil des Namens besteht aus der Top Level Domain, dem Domainnamen und dem Site-Namen, der zweite Teil des Namens besteht aus den Komponenten für die Kategorie. Mit dem Methodennamen, dem dritten Teil, hat man den Fully-Qualified Service Name.

Der Name Solicitation Service (NSS) vergibt eindeutige Präfixe an namenszentrische Dienste, damit diese aus dem Internet unter ihrem Fully-Qualified Service

Name erreichbar sind. Im Lebenszyklusmodell in Abbildung 6.1 kommt der Name Solicitation Service bei der Integration der Dienste zur Anwendung. Der Dienstanbieter des Name Solicitation Service läuft auf einem Gateway, der Dienstnutzer auf den Geräten im drahtlosen Netz.

Die Funktionsweise des Name Solicitation Service wird im Folgenden an einem Beispiel erklärt. In dem Beispiel wird ein drahtloses Netz angenommen, das über ein Gateway in das Internet integriert ist. Auf dem Gateway wird der Name Solicitation Service angeboten. Das Beispiel zeigt, wie die Forwarding Information Base auf den Gateways organisiert ist und welche Dienste neben dem Name Solicitation Service noch bei der Anfrage nach einem Präfix beteiligt sind. Abbildung 6.4 zeigt das drahtlose Netz mit den Knoten und dem Gateway, welches in dem Beispiel verwendet wird. Neben dem Gateway ist in der Abbildung die Forwarding Information Base des Gateways zu sehen. In der Spalte Faces in der Forwarding Information Base tragen der Übersichtlichkeit halber die Faces die Kurzbezeichnungen der Dienste, die sie an den Dämon anbinden. Die Kurzbezeichnung ID steht dabei für den ID Service und NSS für den Name Solicitation Service. Unter den Knoten im drahtlosen Netz sind deren Adressen angegeben.

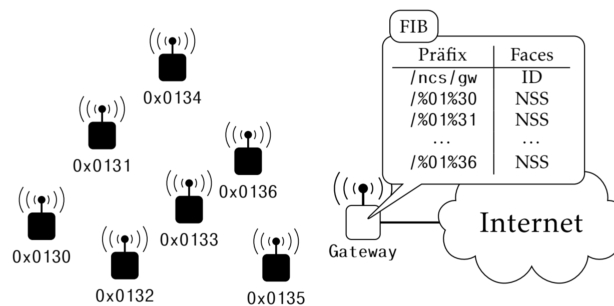


ABBILDUNG 6.4 – *Beispiel eines drahtlosen Netzes mit einem Name Solicitation Service auf einem Gateway*

Der Dienstanbieter des Name Solicitation Service auf dem Gateway vergibt Präfixe für die Knoten im drahtlosen Netz. Wird der Name Solicitation Dienstnutzer auf einem Gerät im drahtlosen Netz gestartet, wird eine Anfrage mit dem Gateway-Präfix `/ncs/gw` an den Name Solicitation Service versendet. Der Suffix der Anfrage setzt sich aus einem Link-Lokal Name ohne Methode (z. B. `/%01%33/climate`), des Dienstes dessen Präfix angefragt wird, sowie der Dienstmethode des Name Solicitation Service zur Anfrage eines Präfixes (`ncs.nss.req()`) zusammen. Im Folgenden wird der Link-Lokal Name ohne Methode als *Link-Local Präfix* bezeichnet. Knotenadresse und Kategorie im Link-Local Präfix werden dabei nicht als Parameter der Dienstmethode übergeben, sondern sind Teil des Präfixes der Anfrage.

Wird die Anfrage, die durch das drahtlose Netz geflutet wird, an dem Gateway empfangen, so wird die Anfrage zuerst an den ID Service weitergeleitet, da dieser mit `/ncs/gw` in der Forwarding Information Base registriert ist. Der ID Service entfernt den Gateway-Präfix `/ncs/gw` der Anfrage und versendet die Anfrage wieder. Nach der Entfernung des Gateway-Präfixes ist der neue Präfix

des Interests eine Knotenadresse. In der Forwarding Information Base gibt es einen Eintrag mit dem Präfix der Adresse dieses Knotens und dieser Eintrag sorgt dafür, dass der Interest an den Name Solicitation Service weiterleitet wird. Der Name Solicitation Service versendet daraufhin eine Antwort mit dem neuen Präfix. Über die Einträge in den Pending Interest Tables wird die Antwort zum Ursprung der Anfrage zurückgeschickt.

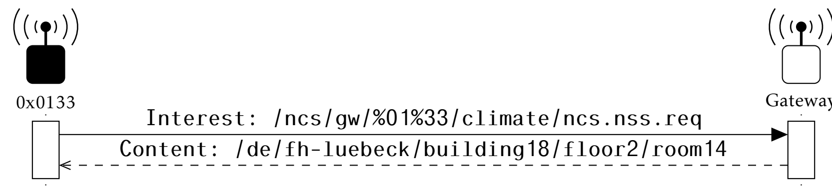


Abbildung 6.5 – *Nachrichtenaustausch zwischen Dienstnutzer und Dienstanbieter des Name Solicitation Service*

Abbildung 6.5 zeigt den Nachrichtenaustausch zwischen Dienstnutzer und Dienstanbieter beim Name Solicitation Service. Die Anfrage von dem Knoten mit der Adresse `0x0133` enthält den Präfix `/ncs/gw`, gefolgt von dem Link-Local Präfix bestehend aus Knotenadresse und der Kategorie des Dienstes, für den ein Präfix angefragt wird. Die Dienstmethode der Anfrage ist `/ncs.nss.req` und diese gibt, in Abhängigkeit der Knotenadresse und der Kategorie, einen Präfix zurück.

6.5.2 Verkürzung von Namen

Namen in inhalts-/namenszentrischen Netzen sind für gewöhnlich – der Präfix `/de/fh-luebeck/building18/floor2/room14` ist 39 Byte groß – länger als eine IPv6 Adresse mit 16 Byte. In drahtlosen Netzen sind, wie in Abschnitt 3.2.1 lange Namen aufgrund der kleinen Paketgrößen eine Herausforderung. Als Alternative zu mathematischen Methoden zur Verkürzung oder Kompression von Namen, wie beispielsweise der Ansatz mit Bloom-Filtern in Abschnitt 4.3.5 oder bei Muñoz et al. [115], werden in diesem Abschnitt zustandsbehaftete Verfahren vorgeschlagen. Diese zustandsbehafteten Verfahren zur Verkürzung von Namen werden mit Hilfe von speziellen Diensten auf Gateways realisiert.

Verkürzung während der Weiterleitung

Die Präfixe der Namen in inhalts-/namenszentrischen Netzen sind unterschiedlichen organisatorischen Verantwortungsbereichen zugeordnet, wie im Lebenszyklusmodell in Abbildung 6.1 und den daraus abgeleiteten Namensbestandteilen in Abbildung 6.2 dargestellt. In dieser Arbeit werden die organisatorischen Verantwortungsbereiche weiter in technische Verantwortungs- oder Geltungsbereiche unterteilt. Ein technischer Geltungsbereich ist zum Beispiel das lokale Netz einer Organisation. Das lokale Netz untergliedert sich wiederum in weitere Subnetze und in die drahtlosen Netze. Jedem dieser technischen Geltungsbereiche ist ein Gateway zugeordnet. Wenn ein Interest durch das Netz weitergeleitet wird, so vergleichen die Gateways die Komponenten des Namens im Interest mit

ihren Einträgen der Forwarding Information Base und treffen so ihre Weiterleitungsentscheidungen. Abbildung 6.6 zeigt ein Beispiel für die technischen und organisatorischen Bereiche des Netzes einer Organisation und des Internets. In dem Beispiel wird aus dem Internet ein Interest mit dem Präfix `/de/fh-luebeck/building18/floor3/room14` empfangen.

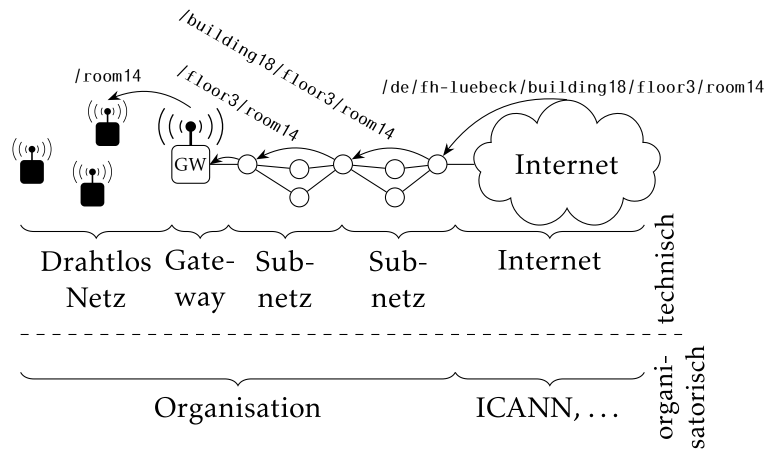


Abbildung 6.6 – *Technische und organisatorische Bereiche aus Sicht einer Organisation*

Das Gateway am Übergang vom Internet in das lokale Netz der Organisation hat in seiner Forwarding Information Base einen Eintrag mit der Top Level Domain und dem Domainnamen `/de/fh-luebeck`. Top Level Domains und Domainnamen werden von der Internet Corporation for Assigned Names and Numbers (ICANN) und weiteren Organisationen verwaltet. Alle Nachrichten mit diesem Präfix `/de/fh-luebeck` werden von dem Gateway in das interne Netz weitergeleitet. Das Gateway ist somit für den Geltungsbereich mit diesem Präfix zuständig. Gleiches gilt für Site-Namen, die hinter dem Domainnamen stehen, wie beispielsweise `/building-18/floor-2/room-14`. Ein Gateway in Gebäude 18 leitet beispielsweise alle Interests in entsprechende Sub- oder Drahtlosnetze in diesem Gebäude weiter.

In den Gateways sind die Präfixe, für die die Gateways zuständig sind, entsprechend konfiguriert. Um die Präfixe und damit die Nachrichten zu verkürzen, entfernen Gateways diese Präfixe bei eingehenden Interests. Nach dem Entfernen der Präfixe werden die Interests dann weiter Richtung drahtloses Netz weitergeleitet. In Abbildung 6.6 wird der Präfix bei der Weiterleitung des Interests über die Gateways schrittweise verkürzt: *(i)* `/de/fh-luebeck/building18/floor3/room14`, *(ii)* `/building18/floor3/room14`, *(iii)* `/floor3/room14` und schließlich *(iv)* `/room14`. Kommen die Interests bei den drahtlosen Netzen an, sind die Namen bereits bedeutend kürzer. Auf den Knoten im drahtlosen Netz werden, wenn die Verkürzung von Namen bei der Weiterleitung angewendet wird, auch nicht mehr die langen Präfixe gespeichert, sondern nur die kurzen. Ebenso würde der Name Solicitation Service einen kurzen Präfix und keinen langen Präfix vergeben (vgl. Nachrichtenaustausch Name Solicitation Service in Abbildung 6.5 auf der vorherigen Seite).

Namen von Content Objects, die als Antwort auf einen Interest geschickt werden, werden bei der Weiterleitung an den zuständigen Gateways um den Präfix erweitert, der dem jeweiligen Geltungsbereich zugeordnet ist. Mit der Strategie des Wegnehmens und Hinzufügens von Präfixen an den Grenzen der Geltungsbereiche entspricht der Name eines Content Objects beim Übergang vom Geltungsbereich der Organisation ins Internet wieder dem Namen des Interests, auf den das Content Object geschickt wurde. Das Wegnehmen und Hinzufügen der Präfixe auf den Gateways wird ebenfalls von dem ID Service realisiert.

Diese Strategie der Verkürzung von Namen funktioniert nur in eine Richtung, nämlich nur für Interests, die aus dem Internet Dienste in dem drahtlosen Netz anfragen. Für den umgekehrten Weg, aus dem drahtlosen Netz in das Internet, ist das nicht möglich, da eine Anfrage in das Internet nur mit einem Fully-Qualified Service Name möglich ist. Geht man davon aus, dass die relevanten Dienste im Internet der Dinge auf den „Dingen" angeboten werden, so kommen die Anfragen aus dem Internet und die Strategie der Verkürzung von Namen bei der Weiterleitung von Interests ist gut anwendbar.

ALIASPRÄFIXE

In dem Beispiel in Abbildung 6.6 wird davon ausgegangen, dass alle Dienste in dem drahtlosen Netz den gleichen Präfix haben, weil das drahtlose Netz von einer Organisation betrieben wird. Diese Organisation wird durch einen Präfix aus Top Level Domain und Domainnamen identifiziert. Wenn in einem drahtlosen Netz Dienste von unterschiedlichen Organisationen betrieben werden, haben die Dienste wahrscheinlich auch unterschiedliche Präfixe. Gibt es unterschiedliche Präfixe für die Dienste in einem drahtlosen Netz, versagt die oben eingeführte Strategie der Verkürzung von Namen, da die Präfixe entfernt werden und eine Zuordnung zu einem entfernten Präfix nicht möglich ist.

Anstatt die Präfixe zu entfernen, bietet es sich an, die Präfixe zu verkürzen. Eine Möglichkeit wäre, den langen Präfix durch eine kurzen Präfix, beispielsweise bestehend aus einer Komponente, zu ersetzen. Einen langen Präfix durch eine kurze Komponente zu ersetzen hat den Nachteil, dass sich das Verhalten beim Vergleich der Namen von Interest und Content Object ändert. Angenommen man hat im Content Store ein Content Object mit dem Namen `/alpha/omega` gespeichert. Dieses Content Object würde auf die Interests mit den Namen `/alpha` und `/alpha/omega` zurückgegeben werden, da jeder der Namen ein Präfix des Namens des Content Objects ist. Würde man die Namen der Interests beispielsweise durch die kürzeren Komponenten `/%01` und `/%02` ersetzen, wäre das Content Object nur mit einem Interest mit dem Namen `/%02`, der `/alpha/omega` entspricht, zu erhalten. Eine Anfrage mit einem unspezifischen Interest mit dem Namen `/alpha` ist in dem obigen Beispiel nicht vorgesehen, da jeweils die zwei Originalkomponenten in einer Komponente zusammengefasst sind.

Eine Alternative zur Ersetzung eines langen Präfixes durch eine kurze Komponente ist die Ersetzung der Komponenten eines langen Präfix durch kürzere Komponenten, die im Folgenden *Aliaskomponenten* genannt werden. `/alpha` wird durch `/%01` und `/omega` wird durch `/%02` ersetzt. Der Name `/alpha/omega`

des Content Objects wäre dann verkürzt /%01/%02 und mit Interests mit den Namen /%01 oder /%01/%02 würde man das Content Object erhalten. Der aus Aliaskomponenten zusammengesetzte Präfix heißt *Aliaspräfix*.

Um im drahtlosen Netz Präfixe und damit Nachrichten zu verkürzen und Speicherplatz auf Knoten zu sparen, ersetzt ein Gateway am Übergang vom drahtgebundenen zum drahtlosen Netz lange Komponenten durch kurze Aliaskomponenten. Die Zuordnung von langen Komponenten zu Aliaskomponenten wird im Gateway konfiguriert. Für die Konfiguration und der Ersetzung von Komponenten zu Aliaskomponenten und zurück wird in dieser Arbeit ein Dienst, der *Alias Prefix Service*, vorgeschlagen. Der Alias Prefix Service wird wie der Name Solicitation Service auf dem Gateway ausgeführt und die Aliaspräfixe werden im drahtlosen Netz mit dem Name Solicitation Service konfiguriert. Dazu fragt der Name Solicitation Service die Aliaspräfixe bei dem Alias Prefix Service an.

Kommen während der Laufzeit, also während des Betriebs des drahtlosen Netzes, neue Namen hinzu, ergibt sich unter Umständen das Problem, dass die langen Präfixe eine andere Ordnung haben als die ihnen zugeordneten kurzen Präfixe. Dieses Problem wird im Folgenden an einem Beispiel erläutert. Angenommen, das Gateway verwaltet die langen Präfixe /alpha/alpha und /alpha/omega und ordnet diesen die kurzen Aliaspräfixe /%01/%01 und /%01/%02 zu. Die kurzen Aliaspräfixe werden im gesamten drahtlosen Netz an Stelle der langen Präfixe benutzt, weil sie durch den Name Solicitation Service im drahtlosen Netz konfiguriert wurden. Nun kommt zu der Konfiguration des drahtlosen Netzes der Präfix /alpha/gamma hinzu. Der Komponente /alpha ist bereits die Aliaskomponente /%01 zugeordnet, der Komponente /gamma wird nun die Aliaskomponente /%03 zugeordnet. Weiterhin wird angenommen, dass es in einem Content Store auf einem Knoten im drahtlosen Netz die Content Objects mit den Aliaspräfixen /%01/%01, /%01/%02 und /%01/%03 gibt. Ein Interest mit dem Namen /%01 liefert das Content Object /%01/%03 zurück, weil immer das erste Content Object mit dem passenden Präfix aus Content Store zurückgegeben wird. Es ist deswegen das erste Content Object, weil die Prüfung auf ein passendes Content Object im Content Store in absteigender Shortlex-Ordnung der Content-Object-Namen erfolgt, wie in Abschnitt 2.4.4 Nachrichtenverarbeitung beschrieben. Der Präfix /%01/%03 ist /alpha/gamma zugeordnet. Wären die Präfixe nicht so durch Aliaspräfixe ersetzt, würde das Content Object mit dem Präfix /alpha/omega zurückgegeben werden, weil es das erste Content Object im Content Store ist.

Abbildung 6.7 fasst die Zuordnung der Präfixe zu den Aliaspräfixen aus dem obigen Beispiel zusammen. Die Präfixe sind in der Abbildung in absteigender Shortlex-Ordnung, wie in einem Content Store, dargestellt. Zuordnungen in Abbildung 6.7 der Präfixe zu den Aliaspräfixen und umgekehrt werden durch Doppelpfeile angezeigt. Die Zuordnung aus dem Beispiel wird als *naive Zuordnung* bezeichnet, da die kurzen Aliaspräfixe nach der naiven Vorgehensweise einer aufsteigenden Nummerierung vergeben werden.

Abbildung 6.7(a) zeigt die Ausgangssituation, in der die Präfixe /alpha/alpha und /alpha/omega den Aliaspräfixen /%01/%01 und /%01/%02 zugeordnet sind. Kommt wie in Abbildung 6.7(b) zur Ausgangssituation nun der Präfix /alpha /gamma mit dem Aliaspräfix /%01/%03 dazu, wäre im drahtlosen Netz in einem

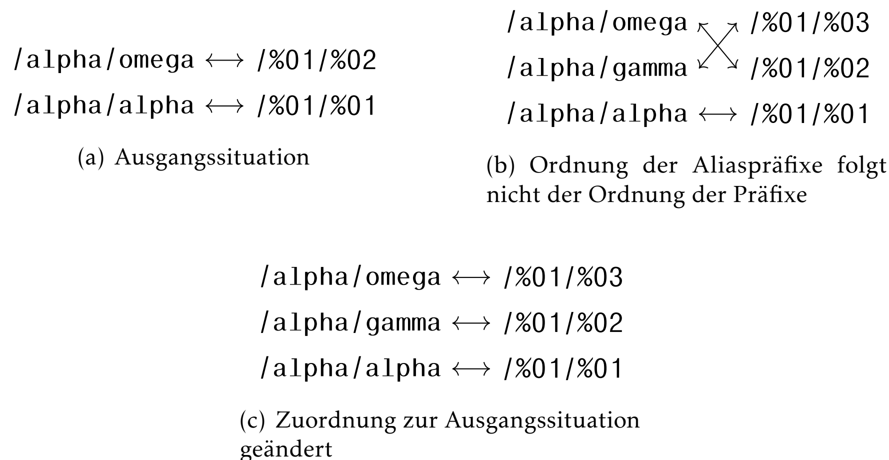


ABBILDUNG 6.7 – *Beispiel für eine naive Zuordnung von Aliasnamen*

Content Store das Content Object mit dem Präfix /alpha/gamma das erste Content Object, was auf einen Interest mit dem Namen /alpha beziehungsweise mit seinem Alias /%01 zurückgegeben werden würde. Das /alpha/gamma zurückgegeben wird, ist aber nicht richtig, da in der Ordnung der Präfixe /alpha/omega zuerst kommt.

Um die richtige Zuordnung wie in Abbildung 6.7(c) zu erhalten, müssten aber nicht nur die Zuordnungen der Präfixe auf dem Gateway getauscht werden, sondern alle Aliaspräfixe in den Datenstrukturen im drahtlosen Netz. Der Tausch der Zuordnungen im drahtlosen Netz ist mit Aufwand verbunden. Im drahtlosen Netz würde /alpha/omega dann dem Aliaspräfix von /%01/%02 zu /%01 /%03 getauscht werden, damit /alpha/gamma dann den Aliaspräfix /%01/%02 bekommen könnte.

Die Lösung, Nummern als Aliaskomponenten zu vergeben, scheint vielversprechend. Anstatt jedoch naiv die Nummern aufsteigend zu vergeben, wird die erste Nummer in der Mitte eines vorher definierten Wertebereichs vergeben. In dem Beispiel wird der ein Byte breite Wertebereich für die Aliaskomponenten /%00 bis /%FF ($FF_{16} = 255_{10}$) angenommen. Angenommen, für die Komponente /alpha wird als erstes ein Alias gesetzt, dann ist die Aliaskomponente /%7F. Der Wert $7F_{16}$ liegt in der Mitte des Wertebereichs. Ist eine Aliaskomponente vergeben, teilt sie den Wertebereich auf, in einen Wertebereich unterhalb und oberhalb der Aliaskomponente. In diesen Wertebereichen werden neue Aliaskomponenten wieder jeweils in der Mitte der Wertebereiche eingefügt. Wird also neben /alpha eine Komponente eingefügt, die größer ist, ist die Aliaskomponente %/BF. Aliaskomponente /%BF liegt in der Mitte des Wertebereichs, beginnend mit /%7F (Aliaskomponente von /alpha) und /%FF (Ende des gesamten Wertebereichs). Ist die eingefügte Komponente kleiner als /alpha, dann ist die Aliaskomponente /%3F, was in dem Mitte des Wertebereichs /%00 (Beginn des gesamten Wertebereichs) und /%7F (Aliaskomponente von /alpha) liegt. Dadurch, dass Aliaskomponenten immer in der Mitte des jeweiligen Wertberei-

che platziert werden, ist vor und hinter jeder Aliaskomponente noch Platz, um weitere Aliaskomponenten einzufügen.

Die Vergabe von Aliaskomponenten durch Aufteilen des Wertebereichs wird dabei in Abhängigkeit des Präfix, welcher vor der Komponente steht, vorgenommen, wie im Folgenden an einem Beispiel erklärt wird. Angenommen, ein Gateway ist für die Präfixe *(i)* /alpha/alpha, *(ii)* /omega/kappa, *(iii)* /alpha/omega, *(iv)* /alpha/gamma und *(v)* /omega/alpha zuständig. Die Reihenfolge der Präfixe gibt den Zeitpunkt des Einfügens an. Zuerst wird der Präfix /alpha/alpha eingefügt. Als Aliaskomponente für den Präfix /alpha wird /%7F gewählt und für den Suffix /alpha ebenfalls, da beide Komponenten die ersten sind, denen eine Aliaskomponente zugewiesen wird. Der zweite Präfix, der hinzugefügt wird ist /omega/kappa. Komponente /alpha hat bereits den Aliaspräfix /%7F, also bekommt /omega den Aliaspräfix /%BF, da /omega nach Shortlex-Ordnung größer ist als /alpha. Für den Präfix /omega, beziehungsweise dessen Aliaspräfix /%BF, ist /kappa der erste Suffix, daher wird /kappa die Aliaskomponente /%7F zugewiesen. Dann werden die Präfixe /alpha/omega und /alpha/gamma eingefügt. Suffix /omega wird die Aliaskomponente /%BF zugewiesen, da dem Suffix /alpha, er als erstes eingefügt wurde, /%7F zugewiesen wurde. Suffix /gamma liegt in der Shortlex-Ordnung zwischen /alpha und /omega und bekommt daher den Aliaspräfix /%9F in der Mitte des Wertebereichs von /%7F bis /%BF zugewiesen. Als letztes wird /omega/alpha eingefügt, wobei der Suffix /alpha den Aliaspräfix /%3F bekommt, da /alpha nach der Shortlex-Ordnung vor /kappa kommt.

Abbildung 6.8 visualisiert das obige Beispiel noch einmal in einer Baumstruktur. Die Ziffern an den Kanten des Baumes stehen für die Einfügereihenfolge beziehungsweise für die Reihenfolge der Vergabe der Aliaspräfixe. Über den Komponenten sind deren Aliaspräfixe angegeben.

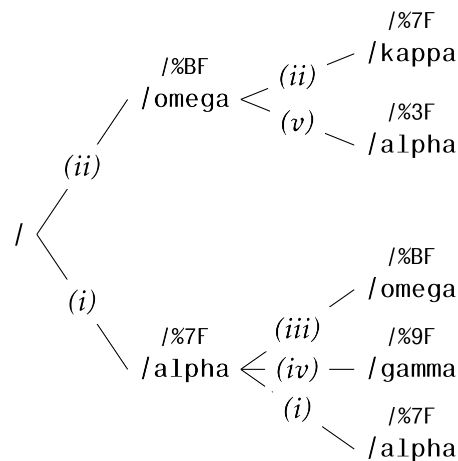


ABBILDUNG 6.8 – *Komponenten eines Namens als Baumstruktur*

In Abbildung 6.8 wird deutlich, dass die Aliaskomponenten für die Suffixe nicht eindeutig sind. Für die Präfixe /alpha und /omega sind den Suffixen /alpha die Aliaskomponenten /%3F beziehungsweise /%7F zugewiesen. Die Aliaspräfixe

wiederum sind eindeutig. Dass die Suffixe nicht eindeutig sind, stellt kein Problem dar, da bei der Nachrichtenverarbeitung stets Präfixe verglichen werden und keine Suffixe.

Die Einschränkungen der Vergabe von Aliaskomponenten durch Aufteilen des Wertebereichs sind einmal die Endlichkeit des Wertebereichs und die Tatsache, dass die Effizienz von der Einfügereihenfolge der Namen abhängt. Bei einer ungünstigen Einfügereihenfolge wird der Wertebereich jedes mal halbiert und damit ist der Wertbereich schnell erschöpft. Um der schnellen Erschöpfung des Wertebereich entgegenzuwirken, bietet es sich an, längere Aliaskomponenten zu verwenden, beispielsweise zwei Byte anstatt einem wie in dem Beispiel oben. Ebenso spart Vorgehensweise, Aliaskomponenten in Abhängigkeit von dem Präfix zu vergeben, Aliaskomponenten ein, da Werte für Aliaskomponenten wiederverwendet werden.

6.6 ZUSAMMENFASSUNG

In diesem Kapitel wurde ein Konzept für den Aufbau von *Namen für Dienste* vorgeschlagen. Das Konzept für den Aufbau von Namen für Dienste folgt einem Lebenszyklusmodell für namenszentrische Dienste im Internet der Dinge. Dieser Lebenszyklus besteht aus drei Phasen: der *Entwicklung*, der *Ausbringung* und der *Integration* der Dienste. Die Phase der Entwicklung umfasst die Softwareentwicklung des Dienstes (*Entwicklungszeit*). Hier werden die *Methodennamen* durch den Entwickler festgelegt. Die Methodennamen bilden den sogenannten *dritten Teil* des Dienstnamens. Bei der Ausbringung werden die Dienste in Kategorien eingruppiert. Jede Kategorie erhält einen Präfix. Die Präfixe der Kategorien bilden den sogenannten *zweiten Teil* des Dienstnamens. Der Präfix mit Domain- und Site-Name, der dem zweiten Abschnitt des Dienstnamens vorangestellt wird, wird als *erster Teil* bezeichnet. Er wird während der Integration der Dienste in das Netz der Organisation, welche die Dienste betreibt, durch die Organisation selbst vergeben.

Namen mit besonderer Bedeutung sind der *Root Name*, der *Link-Local Name* und die vorgeschlagenen *Standardnamen*. Hier wurde insbesondere dargelegt, wo die Namen eingesetzt werden. Der *Root Name* spielt in der Forwarding Information Base eine wichtige Rolle. Ein Eintrag in der Forwarding Information Base mit dem Root Namen als Präfix leitet alle Interests weiter. Der *Link-Local Name* hat als Präfix eine eindeutige Knotenadresse und dient dazu einen Knoten direkt anzusprechen. Die *Standardnamen* dienen dazu, Site- und Kategorienamen in Erfahrung zu bringen. Aus den Site- und Kategorienamen werden die Namen zusammengesetzt, unter denen die Dienstbeschreibungen abgerufen werden.

Mit Gateways werden drahtlose Netze an das Internet angebunden. Gateways führen auch Dienste zur Konfiguration oder zur Verkürzung von Namen aus. Dienste zur Verkürzung von Namen entfernen Präfixe beim Routing über die Gateways im Netz einer Organisation, so dass Namen im drahtlosen Netz kürzer sind. Beim Zurücksenden des Content Objects werden die Präfixe wieder angefügt. Werden in einem drahtlosen Netz Dienste unterschiedlicher Organisationen verwendet, funktioniert die Strategie des Verkürzens durch Entfernen

nicht mehr. Das liegt daran, dass die Zuordnung zu den Organisationen, die im Präfix enthalten ist, fehlt. Um trotzdem eine Verkürzung von Namen zu erreichen, werden sogenannte Aliaspräfixe eingeführt. Bei einem Aliaspräfix werden die Komponenten durch kürzere Komponenten, den sogenannten Aliaskomponenten, ersetzt und so die Namen insgesamt verkürzt. Werden Aliaspräfixe zur Laufzeit hinzugefügt, muss sichergestellt werden, dass die Aliaspräfixe der Ordnung der Präfixe weiterhin folgen. Aliaskomponenten werden daher aus der Mitte eines festgelegten Wertebereichs gewählt und so bleibt oberhalb und unterhalb einer Aliaskomponente Platz, um weitere Aliaskomponenten einzufügen.

KAPITEL 7

BASISDIENSTE

ALS Basisdienste werden in dieser Arbeit Dienste bezeichnet, die für die Funktion eines Netzes benötigt werden. Bei Basisdiensten handelt es sich nicht um einen definierten Satz an Diensten mit einer festgelegten Funktionalität, sondern jede Hardware oder Architektur braucht andere Basisdienste, um zu funktionieren. Der Fokus dieses Kapitels liegt auf Basisdiensten für drahtlose Netze im Internet der Dinge. Basisdienste wurden auf der NetSys 2017 im Rahmen der Demonstration mit dem Titel „Name-Centric Services for the Internet of Things“ (Teubler, Hellbrück) [2] präsentiert.

Wie im Konzept in Kapitel 3 beschrieben, gibt es in einem drahtlosen Netz Dienste für Kommunikation über Knotengrenzen hinweg. Im folgenden Abschnitt 7.1 wird mit dem *Simple-Radio Service* ein sehr einfacher Dienst zur drahtlosen Kommunikation über Knotengrenzen hinweg vorgestellt. Ein Nachteil des Simple-Radio Service ist, dass er Content Objects im gesamten Netz verteilt.

Mit dem *Backpath-Radio Service* in Abschnitt 7.2 wird eine Verbesserung des Simple-Radio Service vorgeschlagen. Diese Verbesserung verhindert die Verteilung von Content Objects im gesamten Netz, indem das Content Object auf einem Pfad zurückgeschickt wird, den eine Kopie des entsprechenden Interests vorher genommen hat. Der Backpath-Radio Service braucht für seine Funktion Informationen darüber, zu welchen Knoten bidirektionale Links bestehen. Um die bidirektionale Links zu ermitteln, greift der Backpath-Radio Service auf den *Neighborhood Service* zurück, der in Abschnitt 7.3 erläutert wird.

Nach dem Neighborhood Service wird der *ID Service* in Abschnitt 7.4 eingeführt. Wie im vorangegangen Kapitel erwähnt, werden vom ID Service Anfragen an den Link-Local Name verarbeitet. Der ID Service ist auch ein Dienst, der auf Gateways ausgeführt wird. Gateways verbinden drahtlose und drahtgebundene Netze, wie in der Systemarchitektur in Abschnitt 4.2.1 dargestellt. Namenszentrische Dienste, die auf dem Gateway ausgeführt werden, werden in

Abschnitt 7.5 vorgestellt. Am Ende des Kapitels werden in Abschnitt 7.6 Dienste zur Authentifizierung kurz skizziert.

7.1 SIMPLE-RADIO SERVICE

Der Simple-Radio Service ist ein einfacher Dienst für eine Kommunikation über die Funkschnittstelle. Die Aufgabe des Simple-Radio Service ist, über die Funkschnittstelle empfangene Interests oder Content Objects an den Dämon weiterzuleiten und vom Dämon empfangene Interests oder Content Objects über die Funkschnittstelle zu versenden.

Der Simple-Radio Service hat über eine Hardwareabstraktion Zugriff auf die Funkschnittstelle. Beim Empfang eines Paketes von der Funkschnittstelle prüft der Simple-Radio Service, ob es sich um einen Interest oder um ein Content Object handelt und leitet die Nachricht ohne weitere Verarbeitung an den Dämon weiter. Handelt es sich bei der Nachricht um einen Interest, so leitet der Dämon den Interest anhand der Einträge in seiner Forwarding Information Base weiter. Empfängt der Dämon ein Content Object vom Simple-Radio Service, so prüft er, ob ein entsprechender Eintrag in der Pending Interests Table vorhanden ist und leitet das Content Object entsprechend weiter. Interests oder Content Objects, die das Simple-Radio vom Dämon empfängt, werden über die Funkschnittstelle als Broadcast Nachricht versendet. Das heißt, dass alle Knoten, die die Nachricht empfangen, diese auch im Dämon, wie in den Grundlagen in Abschnitt 2.4.4 auf Seite 29 beschrieben, verarbeiten. Die grundlegende Architektur mit Funkschnittstelle, Simple-Radio Service und Dämon auf einem Knoten zeigt Abbildung 7.1. Dabei steht der Block mit dem Namen *Radio* und der angedeuteten Antenne links für die nicht näher differenzierten Hard- und Softwarekomponenten der Funkschnittstelle.

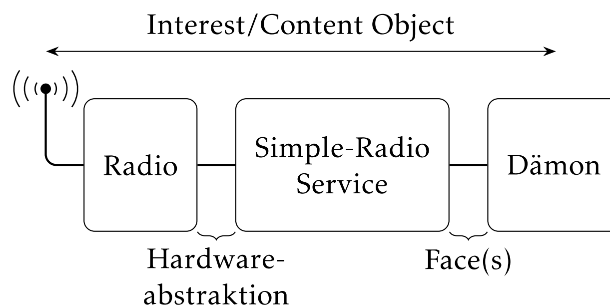


ABBILDUNG 7.1 – *Teil der Architektur aus Radio, Simple-Radio und Dämon*

Die Implementierung des Simple-Radio Service vereint einen Dienstanbieter und einen Dienstnutzer in einer Komponente, weil diese Komponente einerseits wie ein Dienstnutzer Interests sendet und Content Objects empfängt, andererseits wie ein Dienstanbieter Interests empfängt und Content Objects sendet. Aus diesem Grunde gibt es im Simple-Radio Service auch einen Dienstanbieter- und einen Dienstnutzer-Proxy. Ein wichtiges Merkmal, das in der Dienstanbieter/-nutzerkomponente des Simple-Radio Service implementiert ist, ist die

Beachtung des Interest-Scope. Mit dem Interest-Scope wird die Ausbreitung des Interests sichergestellt. Ist die Verbreitung des Interests auf die Dienste beschränkt, so darf der Simple-Radio Service den Interest nicht über die Funkschnittstelle versenden.

Damit der Simple-Radio Service Interests empfangen und dann über die Funkschnittstelle verschicken kann, wird sein Dienstanbieter-Proxy mit dem Root-Präfix in der Forwarding Information Base registriert. Gibt es für einen Interest in der Forwarding Information Base keinen passenden Eintrag mit einem längeren Präfix, so passt der Root-Präfix. Der Interest wird dann von der Default-Methode des Simple-Radio Service verarbeitet und über die Funkschnittstelle versendet.

Abbildung 7.2 zeigt die Architektur des Simple-Radio Service im Detail. Links in Abbildung 7.2 findet sich wie in Abbildung 7.1 die Funkschnittstelle mit der Bezeichnung Radio. In dem Block mit der Bezeichnung Simple-Radio Service sind links die Dienstanbieter/-nutzerkomponente und rechts davon die Proxies dargestellt. Der Nachrichtenaustausch über die Dienstmethoden-Schnittstelle (Default-Methode) und der Face-Schnittstelle ist mit Pfeilen visualisiert. Für die formale Darstellung von Komponenten und Schnittstellen sei auf die Softwarearchitektur in Abschnitt 4.2.2 verwiesen.

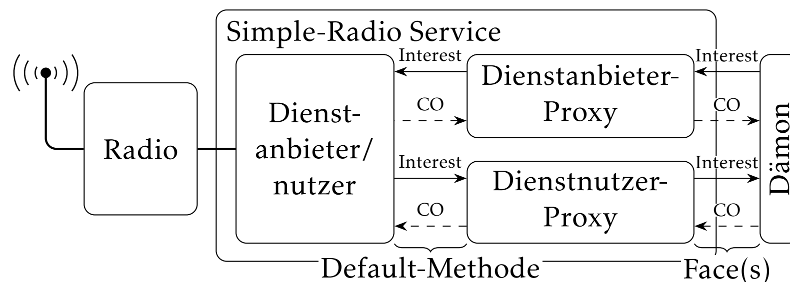


ABBILDUNG 7.2 – *Detaillierte Architektur des Simple-Radio Service*

Aus Gründen der Übersichtlichkeit wurde in Abbildung 7.2 auf die Darstellung von Dienstmethoden zur Konfiguration der Funkschnittstelle verzichtet. Der Kanal der Funkschnittstelle wird über eine Dienstmethode konfiguriert. Um den Dienst zur Konfiguration des Funkkanals anzubieten, wird der Dienstanbieter-Proxy mit dem Präfix `/rf` (engl. Abkürzung für Radio Frequency) in der Forwarding Information Base registriert.

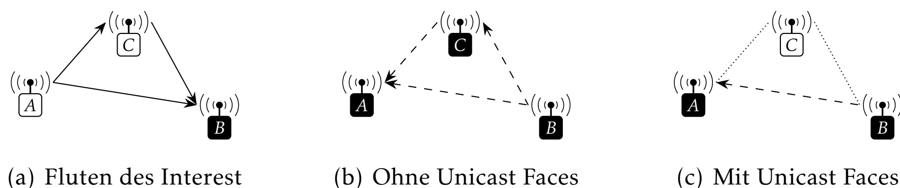
Das Versenden der Nachricht über die Funkschnittstelle entspricht konzeptionell dem Fluten der Nachricht im drahtlosen Netz. Aufgrund der Funktion von CCNx ist es so, dass ein Content Object an alle Knoten des drahtlosen Netzes verteilt wird, wenn Interest und Content Object geflutet werden. Warum das so ist und warum die Verteilung ein Problem darstellt, wird im folgenden Abschnitt erläutert. Darüber hinaus wird im folgenden Abschnitt eine Lösung des Problems präsentiert.

7.2 UNICAST FACES UND BACKPATH-RADIO

Wie im vorherigen Abschnitt beschrieben, verteilt der Simple-Radio Service die Content Objects im drahtlosen Netz. Dieses Verhalten ist an sich unerwünscht, da CCNx eigentlich vorsieht, dass die Content Objects an einem Pfad zurückgeschickt werden, der durch die Interests in den Pending Interest Tables vorgegeben ist. Weiterhin benötigt die Verteilung der Content Objects unnötig Bandbreite im drahtlosen Netz. Das Problem, dass Content Objects im drahtlosen Netz verteilt werden, wird mit Unicast Faces und dem Backpath-Radio Service gelöst. Unicast Faces und Backpath-Radio Service wurden erstmals in der Veröffentlichung „Efficient Data Aggregation with CCNx in Wireless Sensor Networks“ (Teubler et al.) [6] vorgestellt.

UNICAST FACES

Die Notwendigkeit von Unicast Faces wird am Beispiel eines einfachen Ad-hoc-Netzes mit drei Knoten deutlich. Alle Knoten in dem Ad-hoc-Netz sind gegenseitig in Funkreichweite. Ein Knoten schickt einen Interest für ein Content Objekt, das auf einem Knoten gespeichert ist. Die Situation in dem Ad-hoc-Netz mit den drei Knoten zeigt Abbildung 7.3. In der Abbildung werden Knoten, die einen Interest mit einem Content Object beantworten können, in Schwarz dargestellt. Alle anderen Sensorknoten sind in Weiß dargestellt.



(a) Fluten des Interest (b) Ohne Unicast Faces (c) Mit Unicast Faces

ABBILDUNG 7.3 – *Verbreitung von CCNx Nachrichten in drahtlosen Ad-hoc Netzen*

Das Schicken eines Interests ist in Abbildung 7.3(a) dargestellt, wo Knoten A einen Interest sendet, was in der Abbildung mit durchgezogenen Pfeilen angedeutet wird. Knoten B und C empfangen den Interest und speichern diesen in der Pending Interest Table. C versendet den Interest erneut, da der Interest von keinem Dienst auf Knoten C beantwortet wird. Auf den Empfang es Interests folgend, sendet Knoten B ein Content Object als Antwort. Da alle Knoten gegenseitig in Funkreichweite sind, empfangen sowohl A und C das Content Object, wie die gestrichelten Pfeile in Abbildung 7.3(b) zeigen. Da der Interest auch in der Pending Interest Table von Knoten C gespeichert wird, speichert er auch das Content Object. Folglich verbreiten sich die Content Objects im drahtlosen Netz, was ein unerwünschtes Verhalten darstellt. Das gewünschte Verhalten, was mit sogenannten Unicast Faces erreicht wird, zeigt Abbildung 7.3(c), wo das Content Object nur an A geschickt wird.

In der Implementierung, die im Rahmen der Arbeit entstanden ist, wählen die Unicast Faces einen bestimmten Pfad für das Content Object zur Quelle des Interests. Die Auswahl des Pfades funktioniert wie folgt: In drahtlosen Ad-hoc-

Netzen mit vermaschten Topologien wird ein Interest mehrmals kopiert. Es existieren also mehrere Instanzen des gleichen Interests im Netz. Somit treffen auch mehrere Kopien eines Interest bei den Knoten ein, die unterschiedliche Pfade genommen haben. Die Länge eines Pfades, die ein Interest genommen hat, wird durch den Hop-Count ermittelt. Die hier implementieren Unicast Faces wählen den Nachbarn, an den sie das Content Object zurückschicken, anhand der kürzesten Pfadlänge aus. Für das Beispiel in Abbildung 7.3(c) bedeutet das, dass ein Unicast Face auf Knoten *B* das Content Object nur an *A* und nicht an *C* schickt, weil die Kopie des Interests, die *B* von *C* erhalten hat, einen geringeren Hop-Count als die Kopie von *A* hat. In einem größeren Netz wird so der kürzeste Pfad, bezüglich des Hop-Counts, zurück zur Quelle des Interests gewählt.

Unicast Faces heißen so, weil sie sich wie Faces verhalten: Ein Interest kommt von einem Face und das Content Object als Antwort auf den Interest wird wieder an dieses Face zurückgeschickt. Bei Unicast Faces wird das Content Object ebenso zum Sender des Interests – einem benachbarten Knoten – zurückgeschickt. Unicast Faces nutzen die Knotenadresse der Sicherungsschicht, um den Empfängerknoten des Content Objects zu adressieren. Alternativ zum Hop-Count sind noch andere Metriken für die Auswahl möglich, wie beispielsweise die Güte der Verbindung zu dem Nachbarn.

Unicast Faces sind Dienstinstanzen, die einen Dienstnutzer implementieren. Ein Unicast Face besteht aus einem Dienstnutzer-Proxy und einer Dienstnutzer-Komponente, die Zugriff auf die Funkschnittstelle hat. Der Aufbau eines Unicast Face ist somit vergleichbar mit Dienstanbieter-Teil des Simple-Radio Service in Abbildung 7.2. Unicast Faces werden beim Empfang einer Nachricht vom sogenannten *Backpath-Radio Service* erzeugt und von diesem am Dämon registriert. Der Backpath-Radio Service wird im Folgenden Abschnitt erklärt.

BACKPATH-RADIO SERVICE

Der Begriff Backpath (engl. Rückweg) bedeutet, dass dieser Dienst das Zurückschicken von Content Objects an einem Pfad anbietet, den der zum Content Object korrespondierende Interest durch das Netz genommen hat. Der Backpath-Radio Service besteht aus einem Dienstanbieter-Proxy und einer Dienstanbieter-Komponente und ist daher vergleichbar mit dem Dienstanbieter-Teil des Simple-Radio Service. Backpath-Radio Service und Unicast Faces zusammen ermöglichen einem Knoten das Senden und Empfangen von Nachrichten und stellen somit eine Alternative zum Simple-Radio Service dar.

Die Arbeitsweise des Backpath-Radio Service wird im Folgenden an einem Beispiel mit zwei Knoten erklärt. Zur Unterscheidung werden die Knoten im Folgenden mit *A* und *B* bezeichnet, wobei diese Bezeichnungen auch stellvertretend für die Adressen der Sicherungsschicht der jeweiligen Knoten stehen. In dem Beispiel empfängt die Funkschnittstelle von Knoten *A* eine Interest-Nachricht von Knoten *B*. Bei der Verarbeitung der Interest-Nachricht prüft der Backpath-Radio Service, ob es bereits ein Unicast Face für den Sender *B* existiert. Gibt es kein Unicast Face für den Sender, wird ein neues Unicast Face erzeugt und am Dämon registriert. Bei Erzeugung wird dem Unicast Face die Knotenadresse des Senders zugewiesen, in diesem Falle *B*. Mit einem Methodenaufruf

übergibt der Backpath-Radio Service den empfangenen Interest an das Unicast Face. Daraufhin leitet das Unicast Face den Interest an den Dämon weiter, wobei ein Eintrag in der Pending Interest Table für das Unicast Face angelegt wird. Existiert bereits ein Unicast Face für den Sender, inkrementiert das Unicast Face bei jedem Weiterleiten eines Interests einen Zähler und weist sich selbst eine Lebensdauer zu, die der Lebensdauer des weitergeleiteten Interests entspricht.

Der Backpath-Radio Service prüft die Liste der Unicast Faces periodisch und meldet alle Unicast Faces, die ihre Lebensdauer überschritten haben, vom Dämon ab und gibt die Unicast Faces anschließend wieder frei. Wird ein Interest von einem Unicast Face weitergeleitet, wird die Lebensdauer des Unicast Faces entsprechend der Lebensdauer des weitergeleiteten Interests angepasst. Das Anpassen der Lebensdauer geschieht aber nur dann, wenn die Lebensdauer des weitergeleiteten Interests größer ist, als die aktuelle Lebenszeit des Unicast Faces. Damit wird sichergestellt, dass das Unicast Face so lange gültig ist wie der langlebigste Interest, der von diesem Unicast Face weitergeleitet worden ist.

Ein Content Object, das als Antwort auf den Interest erzeugt wird, wird aufgrund des Eintrages in der Pending Interest Table an das Unicast Face weitergeleitet, von dem der Interest kam. Das Unicast Face wiederum versendet das Content Object über die Funkschnittstelle, wobei als Empfänger-Knotenadresse diejenige gesetzt wird, die dem Unicast Face bei der Erstellung zugewiesen wurde (in diesem Falle B). Nach dem erfolgreichen Versenden eines Content Objects wird der Zähler des Unicast Faces dekrementiert. Steht der Zähler auf Null, wird das Unicast Face vom Dämon abgemeldet und gelöscht.

Abbildung 7.4 fasst die Abläufe des Backpath-Radio Service auf Knoten A beim Empfang eines Interests von Knoten B in einem Sequenzdiagramm zusammen. Die Sequenzen werden in der Abbildung mit Sprechblasen erklärt.

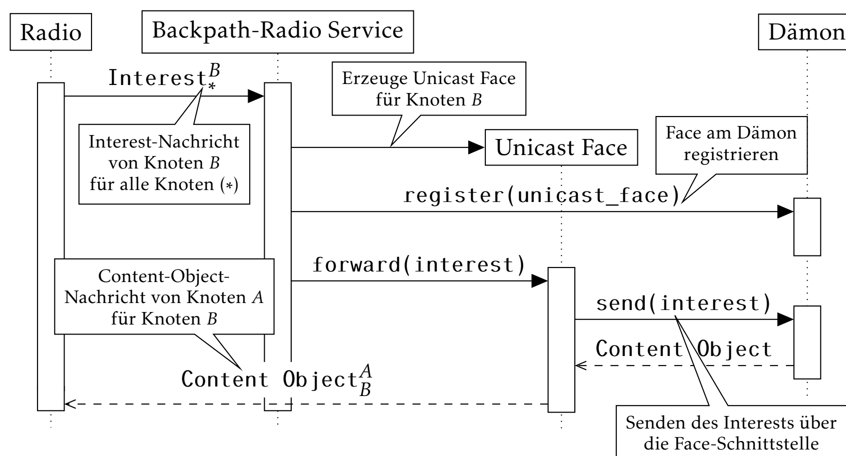


ABBILDUNG 7.4 – *Sequenzdiagramm der Abläufe im Backpath-Radio Service beim Empfang eines Interests an einem Knoten A*

Das Beispiel betrachtete nur den Empfang eines Interests. Bei einem ausgehenden Interest von Knoten A verhält sich das Backpath-Radio wie der Dienstanbieter-Teil des Simple-Radio Service und versendet den Interest über die Funk-

schnittstelle. Der Dienstanbieter-Proxy des Backpath-Radio Service ist wie der Simple-Radio Service mit dem Root-Präfix an der Forwarding Information Base registriert.

Der Backpath-Radio Service zusammen mit den Unicast Faces realisiert das Routing von Content Objects an einem Pfad, den ein Interest vorher genommen hat. Dass dieser Routing-Mechanismus nicht Teil des CCN-IoT-Dämons ist, sondern mit dem Backpath-Radio Service und den Unicast Faces realisiert wird, zeigt die Flexibilität der namenszentrischen Dienste.

7.3 NEIGHBORHOOD SERVICE

Beim Backpath-Radio Service wird gleich beim Empfang eines Interests ein Unicast Face für den benachbarten Knoten angelegt, um Content Objects über dieses Unicast Face zurückzuschicken. Damit wird implizit davon ausgegangen, dass benachbarte Knoten gegenseitig in Funkreichweite sind. In der Realität ist die Funkausbreitung selten kreisrund, wie in dem in den Grundlagen vorgestellten Ad-hoc-Netz in Abbildung 2.2 auf Seite 17. Angenommen die Funkausbreitung ist wie in Abbildung 7.5 elliptisch, dann ist ein Knoten *B* in Funkreichweite eines anderen Knotens *A*, aber der Knoten *A* ist nicht in Funkreichweite von *B*. Damit ergibt sich ein sogenannter *asymetrischer* oder *unidirektionaler Link*, wie in Abbildung 7.5.

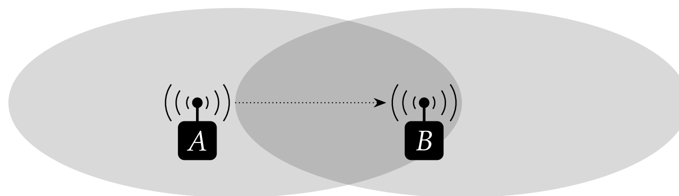


ABBILDUNG 7.5 – *Asymetrischer Link*

Angenommen, Knoten *B* richtet ein Unicast Face für *A* ein, dann wird das Zurückschicken eines Content Objects an diesem Link scheitern, da dieser Link asymmetrisch ist. Das Backpath-Radio braucht eine Möglichkeit, unidirektionale Links zu erkennen um diese nicht als Rückweg zu nutzen. Als Rückweg kommen nur symmetrische Links zu Nachbarknoten – hier als *symmetrische Nachbarschaften* bezeichnet – in Frage.

Der *Neighborhood Service* (dt. Nachbarschaftsdienst) ermittelt die symmetrischen Nachbarschaften für jeden Knoten und stellt diese symmetrischen Nachbarschaften zur Verfügung. Ermittelt werden die symmetrischen Nachbarschaften mit der Dienstmethode `mau()`. Der Name der Methode ist eine Abkürzung für *Mutual Announce and Update* (dt. Gegenseitiges Bekanntgeben und Aktualisieren). Für das Anfragen der Nachbarschaften gibt es die Dienstmethode `list()`, die eine Liste der symmetrischen Nachbarn zurückgibt.

Dass sich so eine Funktionalität, wie sie der Neighborhood Service realisiert, als namenszentrischer Dienst umsetzen lässt, ist nicht sofort ersichtlich. Aus diesem Grunde wird die Dienstbeschreibung der Methode `mau()` in Quelltext 7.1

angegeben. Die Methode hat zwei Parameter, der erste Parameter (`from`) enthält die Knotenadresse des Aufrufers. Der zweite Parameter von `mau()` (`neighbors`) ist ein Array der Adressen der Knoten, die bei dem Aufrufer wiederum `mau()` aufgerufen haben. Um Platz in der Nachricht zu sparen, werden die Differenzen aus Knotenadresse und erstem Parameter (Knotenadresse des aufrufenden Knotens) gespeichert. Methode `mau()` hat keinen Rückgabewert und wird, durch die Angabe des Verbreitungsgebietes des Interests (Scope), nur bei den Nachbarn aufgerufen.

```
1  mau : {
2    doc : "mau - Mutual Announce and Update (to be called
         periodically and with neighbor scope)",
3    params : {
4      from : {
5        doc : "Self node id",
6        type: "u16"
7      },
8      neighbors : {
9        doc : "Diff encoded neighbors (i-th id = from -
           neighbors.get(i))",
10       type : "i8[]"
11     }
12   }
13 }
```

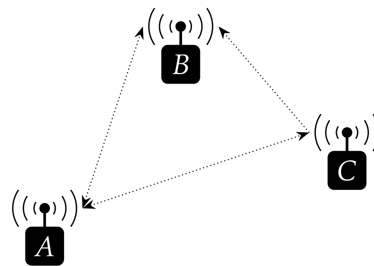
Quelltext 7.1 – *Beschreibung der Methode mau()*

In dem drahtlosen Ad-hoc-Netz in Abbildung 7.6(a) wird `mau()` von den Knoten periodisch aufgerufen. Die Funktionsweise von `mau()` ist in dem Beispiel in Abbildung 7.6 dargestellt. Alle Aufrufe von `mau()` und die darauf folgenden lokalen Methodenaufrufe in dem drahtlosen Netz aus Abbildung 7.6(a) zeigt das Sequenzdiagramm in Abbildung 7.6(b).

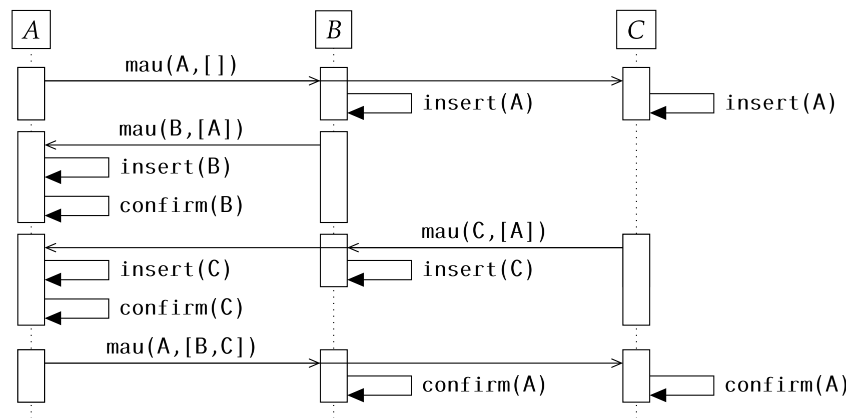
Zwischen Knoten A und B und zwischen A und C existieren symmetrische Links, wie in Abbildung 7.6(a) angezeigt. Der Link zwischen C und B ist unidirektional. Knoten C ist damit ein symmetrischer Nachbar von A.

Methode `mau()` wird nun von allen Knoten periodisch aufgerufen. Die initiale Aufrufsequenz von `mau()` der Knoten A, B, C, A zeigt Abbildung 7.6(b). Knoten A ruft `mau()` auf, Knoten B und C empfangen den Aufruf. Da A noch keine Nachbarn kennt, ist der zweite Parameter ein leeres Array. Knoten B und C speichern A vorerst als asymmetrischen Nachbarn (Aufruf der lokalen Methode `insert()`). Dann ruft B `mau()` auf, der Aufruf erreicht allerdings nur A. Nachdem B vorher den Aufruf von `mau()` von A empfangen hat, enthält das Array (zweiter Parameter von `mau()`) die Adresse von A.

Eigentlich enthält das Array die Differenzen der Adressen B und dem Nachbarknoten, wie oben beschrieben. Aus Gründen der Übersichtlichkeit wurde in Abbildung 7.6(b) auf die Differenzkodierung verzichtet; die Semantik von `mau()` ändert das Weglassen der Differenzkodierung nicht.



(a) Drahtloses Ad-Hoc-Netz mit zwei symmetrischen und einem asymmetrischen Link



(b) Sequenzdiagramm von vier Aufrufen von `mau()` im drahtlosen Ad-hoc-Netz aus (a)

Abbildung 7.6 – *Beispiel für Mutual Announce and Update*

Dadurch, dass A den Aufruf von `mau()` empfängt und sich in der Liste des zweiten Parameters findet, speichert Knoten A den Knoten B als symmetrischen Nachbarn (mit der lokalen Methode `confirm()`). Wie Knoten B ruft auch C `mau()` auf, in der Folge speichert Knoten A Knoten C als symmetrischen Nachbarn. Da Knoten A die Aufrufe von `mau()` von B und C empfangen hat, ruft A `mau()` mit den Knotenadressen von B und C im zweiten Parameter auf, was B und C veranlasst, A als symmetrischen Nachbarn einzutragen.

Methode `mau()` ist ein Beispiel einer Dienstmethode mit einem Seiteneffekt. Der Seiteneffekt von `mau()` ist sind die Aufrufe der lokalen Methoden `insert()` und `confirm()`. Methoden mit Seiteneffekten erlauben die Realisierung beliebiger Message Exchange Pattern. Kurzlebige Inhalte werden in den Parametern der Methode transportiert.

Andere Dienste nutzen die Methode `list()` des Neighborhood Service, um die symmetrischen Nachbarn eines Knotens abzurufen. Methode `list()` hat keine Parameter und liefert bei Aufruf eine `NeighborList` zurück. Der Rückgabewert von `list()` ist der zusammengesetzte Datentyp `NeighborList`, beschrieben in Quelltext 7.2.

```
NeighborList : {
  last : {
        doc : "The last node in the node list",
        type: "u16"
  },
  neighbors : {
        doc : "Other nodes in the list diff encoded (i-th id
             = last - neighbors.get(i))",
        type: "i8[]"
  }
}
```

Quelltext 7.2 – *Beschreibung von NeighborList, dem Rückgabetyp von* `list()`

`NeighborList` hat zwei Felder, `last` und `neighbors`. Feld `last` speichert die Adresse der Nachbarn, die wertmäßig am größten ist. In dem Array `neighbors` sind die anderen symmetrischen Nachbarn, ausgehend vom wertmäßig größten Nachbarn, Effizienzgründen differenzkodiert gespeichert.

Unicast Faces, Backpath-Radio und Neighborhood Service verdeutlichen die Flexibilität der namenzentrischen Dienste. Sie setzen Basisfunktionalitäten um, die beispielsweise von serviceorientierten Architekturen im knotenzentrischen Umfeld nicht umgesetzt werden. Weiterhin sind sie auch ein Beispiel für Dienste im Internet der Dinge, die mit kurzlebigen Inhalten arbeiten, denn Nachbarschaften sind kurzlebige Inhalte, da sie sich möglicherweise schnell ändern.

7.4 ID SERVICE

Der ID Service hilft bei der Adressierung von einzelnen Geräten oder Gruppen von Geräten in namenszentrischen Netzen. ID ist die Kurzform des englischen Begriffs *Identifier*. Die Bezeichnung ID Service wurde in dieser Arbeit gewählt, weil die Aufgabe des Dienstes die Identifikation ist. Mit dem ID Service wird zum Beispiel der Link-Local Name aus Abschnitt 6.4.2 realisiert. Der Link-Local Name identifiziert einen bestimmten Knoten, denn Interests, mit dem Link-Local Namen des Knotens im Präfix, werden nur von diesem bestimmten Knoten verarbeitet.

Ein ID Service speichert einen Präfix, der entweder zur Entwicklungszeit oder zur Laufzeit durch andere Dienste konfiguriert wird. Mit diesem Präfix wird der ID Service auch am Dämon registriert. Beim Link-Local Namen ist dieser Präfix die Knotenadresse. Empfängt der ID Service einen Interest, so entfernt er zunächst den Präfix des Interests, der identisch mit seinem konfigurierten Präfix ist. Dann schickt er den Interest wieder an den Dämon, von wo aus er zu anderen Faces auf dem Knoten weitergeleitet wird.

Der ID Service hilft dabei, Speicherplatz zu sparen und Konfigurationsaufwand minimieren: Angenommen, alle Dienste auf einem Knoten teilen sich den gleichen Präfix, der eine Knotenadresse oder Raumnummer repräsentiert. Mit dem ID Service entfällt die Duplizierung des Präfix in jedem Eintrag in der Forwar-

ding Information Base. Ändert sich der Präfix, beispielsweise wenn sich die Raumnummer ändert, muss diese Änderung nur in der Konfiguration des ID Service geändert werden. Denn wenn sich der Präfix des ID Service ändert, sind alle Dienste unter dem neuen Präfix erreichbar. Ein Knoten kann bei Bedarf auch mehrere ID Services ausführen, beispielsweise um den Link-Local Name und einen Präfix für eine Gruppe zu realisieren. Ein Beispiel für eine Gruppe sind beispielsweise alle Knoten, die mit einem Aktuator ausgestattet sind.

Die Abläufe beim Empfang eines Interests auf einem Knoten, wo der ID Service ausgeführt wird, zeigt das Beispiel in Abbildung 7.7. Der ID Service in dem Beispiel realisiert den Link-Local Name.

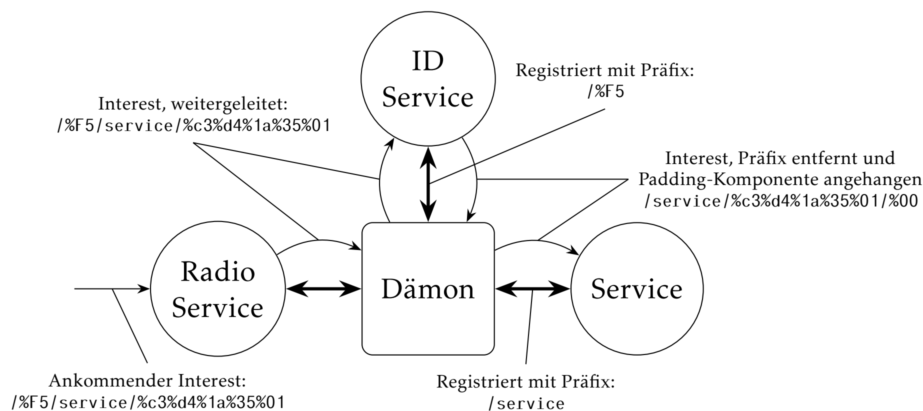


ABBILDUNG 7.7 – *ID Service auf dem Knoten*

Auf der linken Seite in Abbildung 7.7 wird an der Funkschnittstelle ein Interest mit dem Namen `/%F5/service/%c3%d4%1a%35%01` empfangen. Der Radio Service (Simple-Radio oder Backpath-Radio Service) leitet den Interest zum Dämon weiter und der wiederum leitet den Interest zum ID Service, da dieser mit dem Präfix `/%F5` (hier die Knotenadresse 245) am Dämon registriert ist.

Bei der Verarbeitung im ID Service wird der Präfix `/%F5` aus dem Namen des Interests entfernt und eine Padding-Komponente, bestehend aus Nullen als Suffix angehangen. Die Länge der Padding-Komponente entspricht der Länge des entfernten Präfixes. Das Padding verhindert, dass Dienste, die einen Interest nach dem ID Service erhalten und die ursprüngliche Länge des Namens nicht kennen, zu viele Daten in das Antwort-Content-Object speichern, so dass der ID Service anschließend nicht mehr in der Lage ist, den Präfix anzufügen. Nach dem Entfernen des ursprünglichen Präfix und dem Anhängen der Padding-Komponente wird der Interest wieder an den Dämon gegeben. Der Interest, jetzt mit dem Namen `/service/%c3%d4%1a%35%01/%00`, wird nun vom Dämon an den Dienst mit dem Präfix `/service` weitergeleitet. Informationen zum Padding finden sich auch im Anhang in Abschnitt A.1.2.

Ein Content Object, welches als Antwort auf den Interest gesendet wird, hat den gleichen Namen wie der Interest und nimmt durch die Einträge in der Pending Interest Table den umgekehrten Weg über die Dienste in der Reihenfolge: *(i) Service*, *(ii) ID Service* und *(iii) Radio Service*. Der ID Service entfernt die Padding-

Komponente wieder aus dem Namen des Content Objects und fügt den Präfix, der vorher beim Interest entfernt wurde, wieder an das Content Object an.

Das Prinzip des ID Service wird so auch bei der Verkürzung von Namen beim Routing angewendet, wie in Abschnitt 6.5.2 beschrieben. Der Unterschied ist, dass der Interest über mehrere Knoten versendet wird. Weiterhin ist die Paketgröße im drahtgebundenen Netz in der Regel größer als im drahtlosen Netz, daher wird beim Übergang am Gateway auf das Hinzufügen einer Padding-Komponente verzichtet.

7.5 BASISDIENSTE AUF DEM GATEWAY

Drahtlose Netze werden mit Gateways mit dem drahtgebundenen inhalts/namenszentrischen Internet verbunden. Beim Übergang vom drahtgebunden zum drahtlosen Netz findet neben dem Übergang auf ein anderes Übertragungsmedium auch eine *Paketkonvertierung* statt. Die Paketkonvertierung passt Pakete aus dem drahtgebundenen Netz für das drahtlose Netz an und umgekehrt und wird benötigt, da drahtlose Netze mit ressourcenbeschränkten Geräten in der Regel geringere Rahmengrößen als drahtgebundene Netze unterstützen. Eine Art von Paketkonvertierung ist zum Beispiel die Anwendung von Aliasnamen, wie in Abschnitt 6.5 beschrieben. Der Gateway Service wird auf einer speziellen Gateway-Hardwarekomponente ausgeführt. Diese Hardwarekomponente ist in der Lage, mit drahtlosen als auch mit drahtgebundenen Netzen zu kommunizieren.

Der Gateway Service gleicht im Aufbau dem Simple-Radio Service in Abbildung 7.2 auf Seite 141, nur mit dem Unterschied, dass Dienstanbieter- und Dienstnutzerkomponenten des Gateway Service neben dem Weiterleiten von Nachrichten noch eine Paketkonvertierung vornehmen. Bei der Paketkonvertierung werden syntaktische Änderungen vorgenommen, wie beispielsweise Formatänderungen an Steuerungsfeldern der Nachricht.

Semantische Änderungen an den Nachrichten, wie beispielsweise das Ändern von Namen, wird nicht vom Gateway Service, sondern von einem ID Service (vgl. Abschnitt 7.4), vorgenommen. Entsprechend konfigurierte ID Services auf Gateways verkürzen so Namen während des Routings, wie in Abbildung 6.6 auf Seite 132 dargestellt. ID Services, die auf Gateways betrieben werden, hängen keine Padding-Komponenten an, da die Paketgröße im drahtgebundenen Netz in der Regel größer ist als im drahtlosen Netz. Gateway und ID Service sind Basisdienste für Gateways, da der Gateway Service Zugriff auf die Kommunikationshardware braucht und der ID Service Netzbereiche repräsentiert.

Weitere Basisdienste, die auf dem Gateway ausgeführt werden, sind Konfigurationsdienste (engl. Config Services). Konfigurationsdienste nehmen beispielsweise die Konfiguration von Aliasnamen vor. Im Gegensatz zu Gateway und ID Service ist es möglich, Konfigurationsdienste auch im Internet auszuführen. Allerdings ist es sinnvoll, Konfigurationsdienste direkt auf Gateways auszuführen, da Gateways die drahtlosen Netze in das Internet integrieren und nur korrekt konfigurierte Geräte tatsächlich integriert sind.

Abbildung 7.8 zeigt den Aufbau eines Gateways für das inhalts-/namenszentrische Internet der Dinge. Die Darstellung orientiert sich am Knotenmodell für namenszentrische Dienste in Abbildung 3.9 auf Seite 51. Gateway, Config und ID Service sind um den Dämon herum angeordnet, die Doppelpfeile sind die Faces. Der Dämon hat ein Face zu einer Netzwerkschnittstelle (engl. Network Interface Card, kurz NIC), die mit dem Internet verbunden ist.

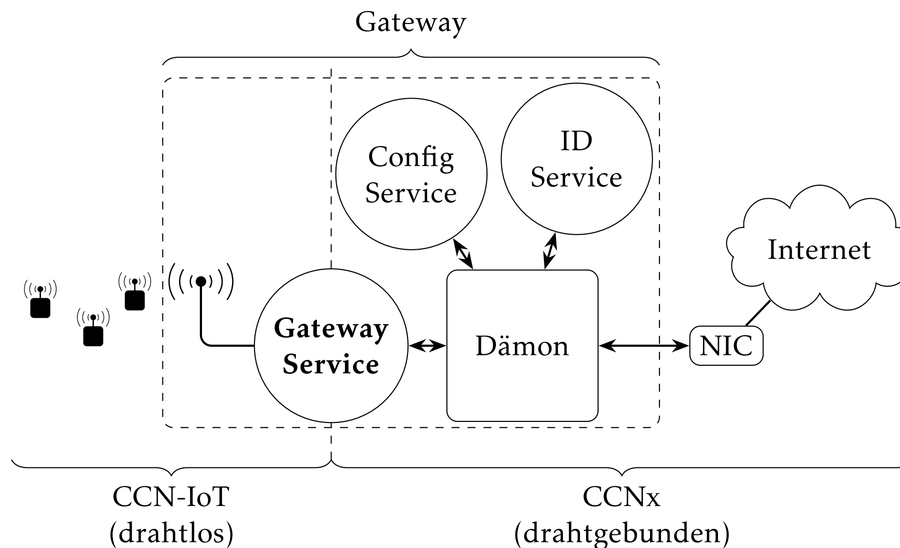


ABBILDUNG 7.8 – *Gateway Service im inhalts-/namenszentrischen Internet der Dinge*

Die Integration ressourcenbeschränkter Geräte in das Internet ist definitiv eine Basisfunktionalität des Internet der Dinge. Nicht minder wichtig sind Authentifizierungsmechanismen, die Dienstnutzer vor falschen Dienstanbietern schützen und die unberechtigte Nutzung von Diensten unterbindet. Authentifizierungsmechanismen mit namenszentrischen Diensten werden im folgenden Abschnitt skizziert.

7.6 DIENSTE ZUR AUTHENTIFIZIERUNG

Das Internet der Dinge birgt die Gefahr, dass beispielsweise durch den böswilligen Austausch von Geräten auch Dienste ausgetauscht werden, mit dem Ziel, Dienstnutzer zu täuschen. Der Dienstanbieter muss sich also gegenüber dem Dienstnutzer authentifizieren. Umgekehrt muss es die Möglichkeit geben, Dienstanbieter vor der unberechtigten Nutzung der angebotenen Dienste zu schützen. Authentifizierungsmechanismen lassen sich, wie im Folgenden gezeigt wird, mit namenszentrischen Diensten umsetzen.

Authentifizierung im Internet basiert auf Verfahren zur Verschlüsselung und zum Schlüsselaustausch. Details zu diesen Verfahren werden in dieser Arbeit nicht diskutiert, da es hier mit dem CCNx Key Exchange Protocol [174] bereits ein Verfahren zum Schlüsselaustausch gibt, welches auf dem Diffie-Hellman-Schlüsselaustausch [63] beruht.

Im Folgenden wird zunächst die Authentifizierung eines Dienstanbieters gezeigt. Dabei wird mit einem *Authentifizierungs-Dienst* und einem *Schlüsselaustausch-Dienst* eine Challenge-Response-Authentifizierung umgesetzt. Beim Challenge-Response Verfahren sind dem Dienstanbieter und dem Dienstnutzer ein Geheimnis, beispielsweise ein Schlüssel s, bekannt. Der Dienstnutzer schickt mit der Anfrage eine Challenge (dt. Herausforderung) x mit und berechnet mit Hilfe einer Funktion $f(s,x)$, deren Parameter der Schlüssel und die Challenge sind, das Ergebnis y. Der Dienstanbieter berechnet nach dem Empfang der Anfrage mit der Challenge ebenfalls $f(s,x) = y'$ und schickt diese in Antwort (engl. Response) wieder zurück. Ist die Response falsch, also $y \neq y'$, weiß der Dienstnutzer, dass sich ein Angreifer als Dienstanbieter ausgegeben hat. Abbildung 7.9 zeigt Authentifizierung des Dienstanbieters beim Methodenaufruf bei der Challenge-Response Authentifizierung.

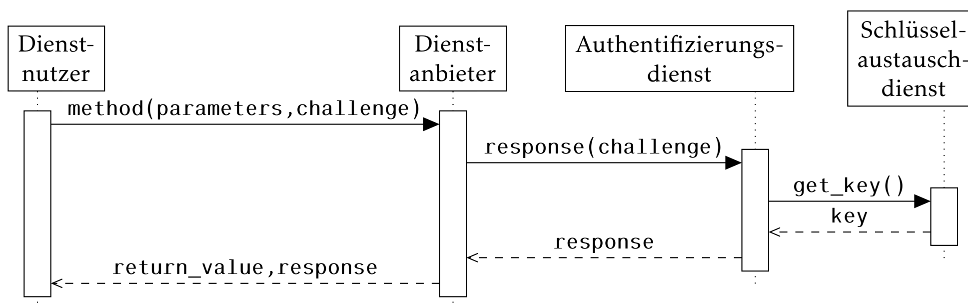


ABBILDUNG 7.9 – *Authentifizierung des Dienstanbieters beim Methodenaufruf*

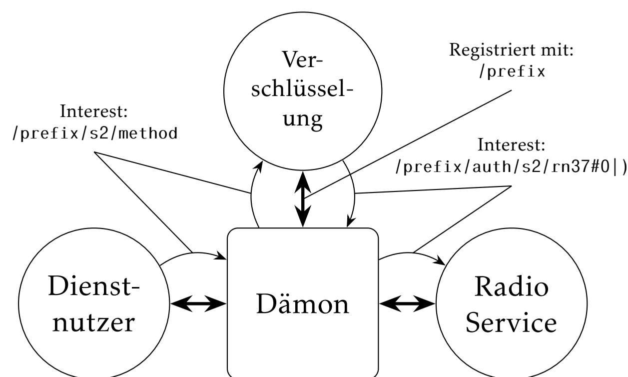
Ein Dienstnutzer ruft eine Methode beim Dienstanbieter auf. Der Methodenaufruf enthält als zusätzlichen Parameter eine Challenge, die der Dienstanbieter löst und sich so beim Dienstnutzer authentifiziert. Um die Response zur Challenge zu berechnen nutzt der Dienstanbieter einen Authentifizierungs-Dienst. Der Authentifizierungs-Dienst fragt das gemeinsame Geheimnis, den Schlüssel(`key`), zwischen Dienstnutzer- und Dienstanbieter vom Schlüsselaustausch-Dienst an. Das gemeinsame Geheimnis wurde vorher in einem separaten Schlüsselaustausch ermittelt.

Mit dem Schlüssel errechnet der Authentifizierungs-Dienst die Response und gibt diese an den Dienstanbieter zurück (`response` in Abbildung 7.9). Der Dienstanbieter gibt den Rückgabewert (`return_value`) und die Response an den Dienstnutzer zurück. Beim Empfang der Rückgabe vom Dienstanbieter prüft der Dienstnutzer die Response, ob sie dem erwarteten Wert entspricht. Der Dienstnutzer weiß somit, ob er es mit dem richtigen Dienstanbieter zu tun hat, oder nicht.

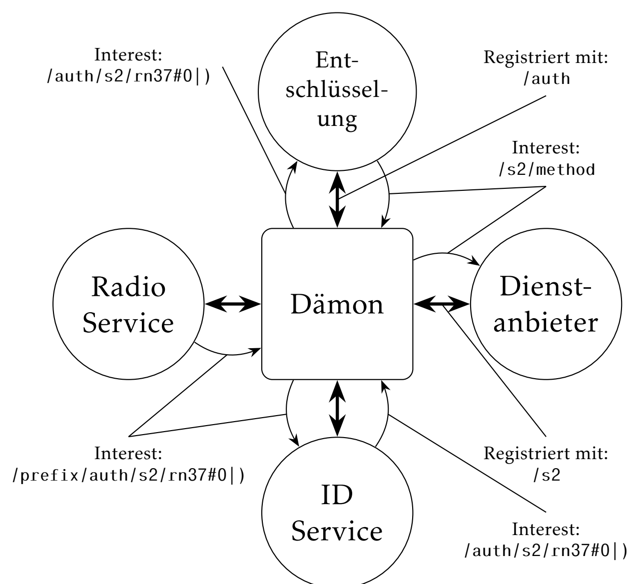
Eine andere Fragestellung ist die Authentifizierung eines Dienstnutzers am Dienstanbieter. Mit der Authentifizierung eines Dienstnutzers am Dienstanbieter wird verhindert, dass Dienste unrechtmäßig genutzt werden. Die Idee bei der Authentifizierung eines Dienstnutzers ist, dass die Dienstmethode in der Anfrage vom Dienstnutzer mit einem Schlüssel verschlüsselt wird, der Dienstanbieter und Dienstnutzer bekannt ist. Alternativ lässt sich hier ein asymmetrisches Verschlüsselungsverfahren einsetzen, wobei die Methode mit im dem privaten

Schlüssel des Dienstnutzers verschlüsselt wird. Die öffentlichen Schlüssel der Dienstnutzer, die zur Dienstnutzung autorisiert sind, werden vom Dienstanbieter gespeichert. Der Dienstnutzer verschlüsselt die Methode in der Anfrage und der Dienstanbieter entschlüsselt die Methode wieder, führt die Methode aus und gibt das Ergebnis zurück.

Das Beispiel Abbildung 7.10 zeigt ein Verfahren zur Authentifizierung eines Dienstnutzers am Dienstanbieter mit namenszentrischen Diensten. In dem Beispiel wird angenommen, dass die Methodenaufrufe unter dem Präfix zu authentifizieren sind.



(a) Dienstnutzerseite



(b) Dienstanbieterseite

ABBILDUNG 7.10 – *Authorisierter Methodenaufruf*

In Abbildung 7.10(a) möchte der Dienstnutzer den Dienstanbieter in Abbildung 7.10(b) nutzen. Der Dienstanbieter ist unter dem Präfix `/prefix` erreichbar.

Ein Dienst, der die Methode verschlüsselt, registriert sich auf der Dienstanbieterseite mit dem Präfix /prefix am Dämon. Interests mit dem Präfix /prefix werden zu dem Dienstanbieter geschickt, der die Authentifizierung vornimmt, indem er die Methode verschlüsselt und den Präfix um /auth erweitert. Nach der Verschlüsselung der Methode wird der Dienst wieder an den Dämon geschickt. Der Dämon schickt den Interest nicht wieder an den Verschlüsselungsdienst zurück, da er den Interest von da erhalten hat. Der einzige Präfix, der jetzt noch passt, ist der Root-Präfix des Radio Service.

Beim Empfang auf der Dienstnutzerseite in Abbildung 7.10(b) beispielsweise durch den ID Service bearbeitet. Der Interest hat nun den Präfix /auth und wird an den Dienst zur Entschlüsselung der Methode weitergeleitet, da dieser mit dem Präfix /auth registriert ist. Bei der Entschlüsselung der Methode wird auch der Präfix /auth entfernt und der Interest wird so an den Dienstanbieter weitergeleitet.

Content Objects gehen den Weg durch die Verarbeitungskette der Dienste zurück, der durch die Einträge in der Pending Interest Table gegeben ist. Der Dienst, der die Methode entschlüsselt hat, macht beim Zurücksenden des Content Objects seine Änderungen am Namen wieder rückgängig, genauso wie der ID Service.

In dem Beispiel zeigt sich wieder die Flexibilität namenszentrischer Dienste, da nicht der Dienstnutzer explizit eine Authentifizierung einleitet, sondern der Verschlüsselungsdienst auf der Dienstnutzerseite. In dem Beispiel wird davon ausgegangen, dass alle Anfragen unter dem /prefix einer Authentifizierung bedürfen. Der Verschlüsselungsdienst auf der Dienstnutzerseite verschlüsselt alle Methoden in Anfragen, die an Präfix /prefix gerichtet sind, unabhängig davon, von welchem Dienstnutzer die Anfrage kommt.

7.7 ZUSAMMENFASSUNG

Basisdienste setzen grundlegende Funktionalitäten im Internet der Dinge um. Die wohl wichtigste Grundfunktionalität im drahtlosen Netz ist die Kommunikation der Dienste untereinander im Netz. Radio Services stellen Dienste zur Kommunikation im Netz bereit.

In diesem Kapitel wurden die Radio Services *Simple-Radio* und der *Backpath-Radio* vorgestellt. Der Simple-Radio Service ist aus einem Dienstanbieter- und Dienstnutzer-Teil aufgebaut und realisiert ein einfaches Fluten von Interests und Content Objects. Der Nachteil des Simple-Radio Service ist, dass angefragte Content Objects stets im ganzen drahtlosen Netz verteilt werden, was mitunter ineffizient ist und nicht dem üblichen Verhalten von CCNx entspricht, wonach Content Objects einer bestimmten Route zum Ursprung des korrespondierenden Interests folgen.

Der Backpath-Radio Service implementiert – im Gegensatz zu Simple-Radio Service – ein Routing für Content Objects. Content Objects folgen beim Backpath-Radio Service der Route des korrespondierenden Interests. Für das Routing erstellt der Backpath-Radio Service temporär *Unicast Faces* beim Empfang eines

Interests. Unicast Faces sind Dienstinstanzen, die vom Backpath-Radio Service am Dämon registriert werden. Sie leiten den Interest zum Dämon weiter und speichern die Sicherungsschichtadresse des Nachbarn, von dem ein Interest empfangen wurde. Bei Content Objects, die als Antwort auf den Interest gesendet werden, ändern Unicast Faces die Sicherungsschichtadresse der Nachricht in die gespeicherte Adresse, so dass nur der Nachbar, von dem der Interest kam, adressiert wird.

Der Backpath-Radio Service funktioniert nur zuverlässig, wenn es seine symmetrischen Nachbarn kennt. Um die Zuverlässigkeit des Backpath-Radio Service zu erhöhen, ermittelt der *Neighborhood Service* die symmetrischen Nachbarschaften. Für den Neighborhood Service wurde eine Dienstbeschreibung angegeben, die zeigt, dass auch dieser Basisdienst mit dem Werkzeug der namenszentrischen Dienste realisiert wird.

Weitere namenszentrische Basisdienste sind der *Gateway Service*, der auf Gateways drahtgebundene und drahtlose Netze miteinander verbindet, in dem er eine Paketkonvertierung vornimmt. Das Kapitel der Basisdienste schließt mit den *Diensten für Authentifizierung*, wo Beispielarchitekturen für Dienste zur Authentifizierung angegeben werden.

KAPITEL 8

ZUSAMMENFASSUNG UND AUSBLICK

ZIEL dieser Arbeit war die Umsetzung und Evaluation von Diensten im inhaltszentrischen Internet der Dinge. Zunächst wurde analysiert, ob sich etablierte Verfahren aus den serviceorientierten Architekturen anwenden lassen und welche Anpassungen an den Verfahren zur Erreichung des Ziels nötig sind. Der Fokus lag dabei auf ressourcenbeschränkten Geräten, da diese in einem heterogenen Internet der Dinge der bestimmende Faktor für ein durchgängiges Kommunikationsparadigma sind. Mit der Evaluation wurde gezeigt, dass der inhaltszentrische Ansatz auch auf ressourcenbeschränkten Geräten funktioniert und dass Dienstbeschreibung und Werkzeugunterstützung die Entwicklung von namenszentrischen Diensten merklich vereinfachen.

Ein Ergebnis dieser Arbeit war das Herausarbeiten von vier grundlegenden Herausforderungen bei der Anwendung des inhaltszentrischen Ansatzes zur Entwicklung von Diensten im Internet der Dinge: *(i)* der konzeptionelle Unterschied zwischen inhaltszentrischem und knotenzentrischem Ansatz, *(ii)* die Dienstentwicklung mit dem inhaltszentrischen Ansatz, *(iii)* der Ressourcenverbrauch und die Verarbeitung von langen und sprechenden Namen sowie *(iv)* das starre Messaging Pattern des inhaltszentrischen Ansatzes.

Ein weiteres Ergebnis dieser Arbeit sind die Potenziale der Anwendung des inhaltszentrischen Ansatzes zur Entwicklung von Diensten im Internet der Dinge: *(i)* Das flexible Knotenmodell des inhaltszentrischen Ansatzes ermöglicht die Realisierung von lose gekoppelten Diensten, *(ii)* die Adressierung mit Namen erlaubt eine größere Flexibilität, *(iii)* ressourcenbeschränkte Geräte im Internet der Dinge profitieren vom Caching, da es neben den kurzlebigen auch langlebige Daten (z. B. Softwareupdates) gibt. *(iv)* Einfache Messaging Pattern wie In/Out-Only werden mit kurzlebigen Interests umgesetzt, wobei die Daten ein Bestandteil des Namens des Interests sind. Kurzlebige Interests sind deswegen geeignet, da man davon ausgeht, dass es im Internet der Dinge viele Dateneinheiten von geringer Größe und kurzer Lebensdauer geben wird.

Als wissenschaftlicher Beitrag entstand aus den Potenzialen das Konzept der namenszentrischen Dienste. Ausgehend von diesem Konzept der namenszentrischen Dienste wurden Komponenten sowie die Schnittstellen zwischen den Komponenten identifiziert und definiert, die für die Realisierung der namenszentrischen Dienste relevant sind. Bei den Komponenten wird zwischen Hard- und Softwarekomponenten unterschieden. Hardwarekomponenten im Internet der Dinge sind alle Geräte im Internet inklusive der ressourcenbeschränkten Geräte. Softwarekomponenten sind Dienstanbieter und Dienstnutzer sowie Dienstanbieter- und Dienstnutzer-Proxy. Dienstanbieter und Dienstnutzer interagieren jeweils über eine generierte API-Schnittstelle mit den Proxies. Die Proxies wiederum interagieren über die Face-Schnittstelle mit dem CCN-Dämon. Dienstanbieter und Dienstnutzer interagieren über entfernte Dienstmethodenaufrufe. Dienste und der Dämon werden auf Knoten oder Gateways ausgeführt.

Das Konzept ist die Grundlage für die Architektur und Implementierung von CCN-IoT. CCN-IoT ist eine leichtgewichtige Variante von CCN für ressourcenbeschränkte Geräte, die im Rahmen der Arbeit entstanden ist. Ein Ergebnis der Implementierung ist, dass Namen, Nachrichten und Dienstmethoden eine Basisklasse für serialisierbare Objekte zugrunde legt, wobei Namen die Basisklassen für Nachrichten und Dienstmethoden sind. Die Verarbeitung von Namen und Nachrichten erfolgt mithilfe von Iteratoren. Ein Ergebnis bei der Entwicklung der Iteratoren ist, dass eine Aufteilung der Iteratoren in nur lesend und lesend und modifizierend sinnvoll ist. So ist der nur lesende Iterator bei Vergleichen mehr als doppelt so schnell gegenüber dem lesenden und modifizierenden Iterator.

Weiterhin wurde im Rahmen der Implementierung auch eine Optimierung des Speicherverbrauchs der Datenstruktur Forwarding Information Base (FIB) vorgeschlagen, da Optimierungen der FIB in der Literatur bisher kaum betrachtet wurden. Ausgehend von der aktuellen FIB-Implementierung von CCN-IoT zeigt die Optimierung der FIB mittels Bloom-Filtern (FIB-BF) eine Kompressionsrate von 1/3 und die mit Hashes (FIB-Hash) von 1/4. Die Kompressionsrate von FIB-BF wird nur unter den Nebenbedingungen erreicht, dass Präfixe gleich auf die Faces verteilt sind und dass Einträge aus der FIB nicht entfernbar sind. Das schränkt die Flexibilität von FIB-BF ein. Für FIB-Hash gelten diese Nebenbedingungen nicht, damit ist FIB-Hash flexibler. Die Verwendung von Iteratoren für Namen und Nachrichten sowie die Speicheroptimierung der FIB sind Maßnahmen zur Verringerung des Ressourcenverbrauchs und effizienten Verarbeitung von Namen, die in dieser Arbeit identifiziert wurden.

Ein weiteres Ergebnis ist die Entwicklung einer leichtgewichtigen Dienstbeschreibung auf Basis von JSON. Diese erlaubt eine automatische Codegenerierung mittels eines Werkzeugs, was eine messbare Arbeitsersparnis für Entwickler von namenszentrischen Diensten darstellt. Die Arbeitsersparnis ergibt sich daraus, dass die Dienstbeschreibung weniger logische Codezeilen als der komplexe Code für die Proxies hat und somit der Gesamtaufwand der manuellen Erstellung geringer ist. Außerdem wird vom inhaltszentrischen Ansatz abstrahiert und die Entwicklung von Diensten erleichtert.

Das Lebenszyklusmodell für Dienste im Internet der Dinge ist ein weiterer wissenschaftlicher Beitrag dieser Arbeit. Auf Basis dieses Lebenszyklusmodells

wurde danach ein Konzept für Namen für Dienste entwickelt. Das Lebenszyklusmodell besteht aus drei Abschnitten: *(i)* der Dienstentwicklung, *(ii)* der Ausbringung und *(iii)* der Integration der Dienste. In jedem Abschnitt wird ein Teil des Namens hinzugefügt, wobei der Name rückwärts aufgebaut wird. Während der Dienstentwicklung wird der dritte Teil, der Methodenname, hinzugefügt. Bei der Ausbringung wird eine Kategorie als zweiter Teil des Namens und während der Integration der erste Teil des Namens hinzugefügt. Der erste Teil unterteilt sich in Domainname und Site-Name.

Bei Betrachtung der Namen für Dienste wurden zustandsbehaftete Verfahren zur Verkürzung von Namen entworfen und umgesetzt. Mit diesen Verfahren ist es möglich, insbesondere in drahtlosen Netzen, den Präfix des Namens in Nachrichten zu verkürzen. Die Verkürzung erfolgt durch Weglassen eines Präfix oder durch Ersetzung mit einem kürzeren Alias. Diese Verkürzungsstrategien von Namen sind ebenfalls eine Maßnahme, um mit langen Namen in drahtlosen Netzen aus ressourcenbeschränkten Geräten umzugehen.

Zum Abschluss wurden mit namenszentrischen Diensten Basisdienste realisiert. So wurden Basisdienste zur Kommunikation umgesetzt und ausgebracht, die dem Konzept der namenszentrischen Dienste folgen und aus Dienstanbieter, Dienstnutzer und Proxies bestehen. Für einen Dienst zur Ermittlung der Nachbarschaften wurde eine Dienstbeschreibung angegeben. Proxies und der Stub-Code für diesen Dienst wurden mit der Werkzeugunterstützung erstellt. Bei der Ermittlung der Nachbarschaften wurden entfernte Dienstmethoden gezeigt, die keine Antwort erwarten, sondern den Zustand des Empfängers mit einem Seiteneffekt ändern. Aufrufe von Dienstmethoden ohne Rückgabewert werden mit kurzlebigen Interests realisiert und zeigen, wie man In-Only Messaging Pattern umsetzt.

Zu Demonstrationszwecken wurde neben den Basisdiensten ein einfacher Konfigurationsdienst erstellt. Der Konfigurationsdienst integriert namenszentrische Dienste auf drahtlosen Sensorknoten in ein drahtgebundenes inhaltszentrisches Netz. Um die Integration zu testen, wurde ein Dienst zum Schalten der LEDs auf den Knoten implementiert.

Namenszentrische Dienste ermöglichen die Entwicklung von Diensten im inhaltszentrischen Internet der Dinge. Für die Zukunft bedarf es allerdings weiterer Untersuchungen, um zu zeigen, dass die namenszentrischen Dienste in der Lage sind, ein breites Anwendungsspektrum abzudecken.

Bei der Betrachtung von Anwendungen werden sich noch weitere Anforderungen an Namen ergeben. Namen folgen einer Semantik, die sich aus dem jeweiligen Kontext der Anwendung ergibt, beispielsweise Lokationsinformationen, wie eine geografische Position. Eine offene Fragestellung ist die Syntax von solchen Namen. Es durchaus vorstellbar, dass es mehrere Anwendungen gibt, die gleiche oder ähnliche Anforderungen an die Syntax haben. Diese Fragestellung lässt sich erweitern, in dem man fragt, welche Anforderungen es an die Namen gibt und welche Anwendungen diese Anforderungen haben. Ein Ziel der Erfassung dieser Anforderungen ist es, weiterführende Standards für Namen vorzuschlagen.

Im Zuge der vorgeschlagenen Standardnamen ist es sinnvoll, die Dienstbeschreibung und Werkzeugunterstützung zu erweitern, um dem Entwickler bei der Einhaltung der Standards zu unterstützen. Darunter fallen beispielsweise, dass Format und Wertebereich für Namenspräfixe in Übereinstimmung mit den vorgeschlagenen Standards eingehalten werden. Eine offene Frage ist hierbei, wie man den Mechanismus zur Einhaltung des Standards für Namenspräfixe in der Dienstbeschreibung abbildet und in den Proxies realisiert.

Ebenso offen ist eine vergleichende Leistungsbewertung der namenszentrischen Dienste mit den knotenzentrischen Ansätzen im Internet der Dinge. Diese Arbeit stellt mit CCN-IoT und der Implementierung der namenszentrischen Dienste eine Basis zur Verfügung, um eine Bewertung durchzuführen. Anwendungen, die einmal knotenzentrisch und als namenszentrische Dienste implementiert werden, helfen bei der Ermittlung der Leistungskriterien für den Vergleich der beiden Ansätze.

Weiterhin ist die Verallgemeinerung des Knotenmodells dahingehend denkbar, dass man Dienste in die Lage versetzt, Nachrichten mit beliebigen Formaten zu verarbeiten. Das Ziel dieser Verallgemeinerung ist es, in Abhängigkeit der Anwendungen sowohl knotenzentrische als auch inhaltszentrische Ansätze zu implementieren, je nachdem welcher Ansatz für die jeweilige Anwendung besser geeignet ist. Die Dienstbeschreibung wird bei dieser Erweiterung des Knotenmodells zu einem allgemeinen Beschreibungsformat für Nachrichten. Die Werkzeugunterstützung generiert dann Parser für diese Nachrichten.

Ein weiterer konsequenter Schritt ist die Erweiterung der Dienstbeschreibung zu einer plattformunabhängigen Sprache für das Internet der Dinge. Bisher wird von der Werkzeugunterstützung Quellcode für Proxies und die Stubs generiert, die Semantik des Dienstes wird bisher mit einer plattformabhängigen Programmiersprache realisiert. Bei der Erweiterung der Dienstbeschreibung hin zu einer plattformunabhängigen Sprache für das Internet der Dinge wird auch die Semantik des Dienstes beschrieben, was in einem heterogenen Internet der Dinge vorteilhaft wäre.

Zusammenfassend verbessert die in dieser Arbeit entwickelte Dienstbeschreibung und Werkzeugunterstützung die Entwicklung von Diensten im inhaltszentrischen Internet der Dinge, indem Teile des Quellcodes generiert werden, die bei manueller Erstellung fehlerträchtig und nicht Teil der Anwendung sind. Mit CCN-IoT wurde eine Plattform für das inhaltszentrische Internet der Dinge entwickelt, die ein Wegbereiter für die namenszentrischen Dienste ist. Damit ist die zukünftige Basis für eine effiziente Entwicklung von Anwendungen in einem inhalts-/namenszentrischen Internet der Dinge mit seinen heterogenen Plattformen geschaffen.

ANHANG **A**

ANHANG

DER Anhang enthält ergänzende Informationen zur Implementierung in Abschnitt A.1, den Untersuchungen zum NDN-Testbed in Abschnitt A.2 und zur Dienstbeschreibung und Werkzeugunterstützung in Abschnitt A.3. Die Abschnitte des Anhangs folgen der Reihenfolge ihrer erstmaligen Referenzierung in den Kapiteln.

A.1 IMPLEMENTIERUNG

Die effiziente Kodierung und Verarbeitung von Namen und Nachrichten ist ein wesentlicher Teil von CCN-IoT. Bei der Kodierung von Namen und Nachrichten werden Felder variabler Länge eingesetzt. Felder variabler Länge werden durch ein Typ-Längen-Feld beschrieben, das am Anfang des Felds variabler Länge steht. Details zur Implementierung und Kodierung von Typ-Längen-Feldern werden in Abschnitt A.1.2 bzw. Abschnitt A.1.1 vorgestellt. Abschnitt A.1.3 zeigt die Implementierungsdetails der Iteratoren mit Hilfe von UML-Klassendiagrammen. Ebenfalls wichtig für eine effiziente Nachrichtenverarbeitung sind die Datenstrukturen Forwarding Information Base, Pending Interest Table und Content Store. Die Implementierungen der CCN-Datenstrukturen in CCN-IoT werden in Abschnitt A.1.4 vorgestellt.

A.1.1 TYP-LÄNGEN-FELDER

Bei CCN-IoT werden vier Typen von Feldern variabler Länge definiert: *(i)* Flag, *(ii)* Ganzzahltyp (Integer), *(iii)* Array und *(iv)* Namenskomponente.

Sind die höherwertigen drei Bits eines Typ-Längen-Feldes gleich Null, so kodieren sie ein sogenanntes Flag. Flags werden für Steuerinformationen in Nachrichten genutzt. Beispielsweise wird die sogenannte AnswerOriginKind-Eigenschaft des Interests als Flag kodiert. Die AnswerOriginKind-Eigenschaft steuert, ob man auch veraltete Daten empfangen möchte.

Sind die höherwertigen drei Bits im Bereich von 001_2 bis 100_2, so kodieren sie einen Ganzzahltyp, wobei die Bits die Länge des Ganzzahltyps kodieren. Ganzzahltypen gibt es in den Längen ein bis vier Byte und sie werden ohne führende Null-Bytes serialisiert. Die Serialisierung ohne führende Null-Bytes ist dabei unabhängig vom Ursprungstyp. Das heißt, dass beispielsweise ein 16-Bit-Ganzzahltyp mit dem Wert 12 als 0x0C serialisiert wird und nicht als 0x000C. Der größte serialisierbare Ganzzahltyp hat 32 Bit und ist vorzeichenlos. Negative Ganzzahltypen werden in der Zweierkomplementdarstellung serialisiert, so wird der 16-Bit-Ganzzahltyp mit dem Wert -12 als 0xFFF4 serialisiert. Die Ganzzahltypen entsprechen den für Mikrocontroller-Plattformen üblichen Ganzzahltypen. (Eine Serialisierung von Gleitkommatypen wird nicht umgesetzt, da diese auf Mikrocontroller-Plattformen aus Performancegründen ohnehin kaum verwendet werden.)

Ein Beispiel für einen Ganzzahltyp ist der Hop-Zähler des Interests, für den der Typ 0x10 definiert ist. Abbildung A.1 zeigt ein Beispiel für einen serialisierten Hop-Count. Das Typ-Längen-Feld ist 0x30 ($0011\,0000_2$), wobei 0x20 ($0010\,0000_2$) die Länge und 0x10 ($0001\,0000_2$) der Typ für den Hop-Count ist ($0010\,0000_2 \lor 0001\,0000_2 = 0011\,0000_2$). Das folgende Byte, also der aktuelle Wert des Hop-Counts, ist 0x10 (16_{10}).

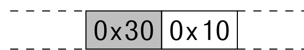


Abbildung A.1 – *Serialisierter Hop-Count*

Bei Arrays sind die höherwertigen drei Bits 101_2 und die niederwertigen fünf Bits kodieren die Länge des Arrays. Arrays bei CCN-IoT sind immer Bytearrays. Die im Rahmen der Arbeit in Kapitel 5 eingeführten namenszentrischen Dienste nutzen das erste Byte in einem Array, um den darin serialisierten Daten einen Datentyp zuzuweisen. Bei Namenskomponenten sind die höherwertigen drei Bits 110_2 und die niederwertigen fünf Bits kodieren die Länge der Namenskomponente plus eins. Ist als Länge für eine Namenskomponente fünf angegeben, so ist die Namenskomponente sechs Byte lang.

Die Kodierung der Typ-Längen-Felder des CCN-IoT-Buffers ist in Tabelle A.1 zusammengefasst. Bei den Typ-Längen-Feldern ist noch ein reservierter Bereich (höherwertige drei Bits gleich Eins) für zukünftige Erweiterungen vorgesehen.

Der Zugriff auf ein Feld variabler Länge erfolgt über das erste Feld variabler Länge. Ist das erste Feld variabler Länge bereits das gesuchte, so werden die Daten ausgelesen. Wenn es nicht das gesuchte Feld ist, dann wird das Feld übersprungen, da die Länge bekannt ist und es wird das nächste Typ-Längen-Feld geprüft. Abbildung A.2 zeigt den Zugriff auf die Felder variabler Länge. Das gesuchte Feld ist drei Felder vom Anfang der Felder variabler Länge entfernt. Es werden drei Schritte benötigt, um das gesuchte Feld variabler Länge zu finden.

A.1.2 BUFFER-IMPLEMENTIERUNGEN

In diesem Abschnitt werden die Buffer-Implementierungen *Name*, *Method*, *ServiceName*, *Interest* und *Content Object* detailliert vorgestellt.

<table>
<tr><th colspan="2"></th><th colspan="4">Bit</th></tr>
<tr><th colspan="2">Typ/Länge</th><th>7</th><th>6</th><th>5</th><th>4 … 0</th></tr>
<tr><td colspan="2">Flag</td><td>0</td><td>0</td><td>0</td><td>Typ = $t \in [00001_2, 11111_2]$</td></tr>
<tr><td rowspan="4">Integer</td><td>1 Byte</td><td>0</td><td>0</td><td>1</td><td rowspan="4">Typ = $t \in [00000_2, 11111_2]$</td></tr>
<tr><td>2 Byte</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>3 Byte</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>4 Byte</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="2">Array</td><td>1</td><td>0</td><td>1</td><td>Länge = $l \in [00000_2, 11111_2]$</td></tr>
<tr><td colspan="2">Namenskomponente</td><td>1</td><td>1</td><td>0</td><td>Länge = $1 + l \in [00000_2, 11111_2]$</td></tr>
<tr><td colspan="2">Reserviert</td><td>1</td><td>1</td><td>1</td><td>Reserviert</td></tr>
</table>

Tabelle A.1 – *Kodierung der Typ-Längen-Felder*

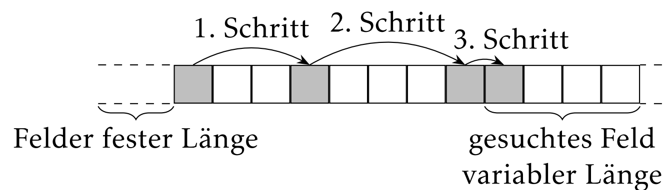


Abbildung A.2 – *Zugriff auf Felder variabler Länge*

NAME

Die Klasse `Name` repräsentiert einen CCN-Namen (vgl. Definition 2.8). Aufgrund der beschränkten Ressourcen von drahtlosen Sensornetzen ist die Länge der Komponenten und die Länge eines Namens insgesamt begrenzt. Die maximale Länge des Namens beträgt 60 Byte, die maximale Länge einer Komponente 32 Byte (vgl. Typ-Längen-Felder in Tabelle A.1).

In dieser Implementierung wird der Name nicht als Ganzes, sondern als Summe seiner Komponenten betrachtet. Komponenten werden einzeln als Felder variabler Länge serialisiert. Eine Besonderheit ist, dass die erste Komponente im Buffer am Anfang des Bereichs der Felder variabler Länge steht und dass alle Komponenten aufeinander folgen, wie im Beispiel in Abbildung A.3 dargestellt. Die Typ-Längen-Felder sind farblich abgesetzt.

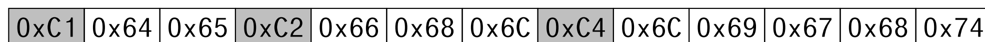


Abbildung A.3 – *Serialisierter Name* `/de/fhl/light`

Aufeinanderfolgende Komponenten erlauben einen schnellen Vergleich von Namen untereinander. Methoden/Operatoren zum Vergleich von Namen werden von der Klasse Name (vgl. Klassenhierarchie der Buffer-Implementierungen in Abbildung 4.18 auf Seite 81) implementiert.

SERVICE NAME

Die Klasse `ServiceName` implementiert sogenannte Service Namen (dt. Dienstnamen). Diese Klasse erweitert `Name` um Funktionalität, die für die namenszentrischen Dienste benötigt wird. Dazu zählt das Setzen und Auffinden von Methoden und Segmentmarkern im Namen. Methoden und Segmentmarker sind spezielle Namenskomponenten. Ein Segmentmarker kennzeichnet einen Teil von zusammenhängenden Daten (vgl. Abbildung 3.16 auf Seite 58). Methoden als Namenskomponenten sind beispielsweise CCNx-Kommandos, die in Abschnitt 3.2.2 in Abbildung 3.5 auf Seite 47 eingeführt wurden.

Eine weitere Funktion der Klasse `ServiceName` ist das Hinzufügen und Entfernen von Auffüll-Komponenten (engl. Padding) am Ende des Namens. Padding wird benötigt, um die ursprüngliche Länge des Namens zu erhalten, wenn beim Routing der Präfix einer Nachricht entfernt wird. Die ursprüngliche Länge des Namens beim Routing zu erhalten ist wichtig, da die Padding-Komponenten beim Rückweg wieder entfernt und die ursprünglich entfernten Präfixkomponenten wieder angefügt werden. Ohne Padding-Komponenten würden Dienste unter Umständen zu viele Daten in die Antwort speichern, so dass kein Platz mehr zum Anfügen für die Präfix-Komponenten wäre. Verkürzung der Namen beim Routing wird in Kapitel 6 behandelt.

INTEREST

Der Interest wird wie alle Nachrichten von den Klassen `Buffer` und von `Type` abgeleitet (vgl. Klassenhierarchie der Buffer-Implementierungen in Abbildung 4.18 auf Seite 81). Klasse `Type` implementiert die Funktionalität, um auf den Nachrichtentyp zuzugreifen. Der Nachrichtentyp bei CCN-IoT Nachrichten wird als Feld fester Länge vor dem Namen gespeichert. Den prinzipiellen Aufbau einer serialisierten Interests zeigt Abbildung A.4.

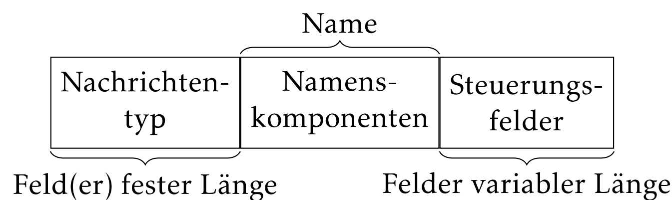


Abbildung A.4 – *Aufbau eines serialisierten Interests*

Am Anfang des Interests steht der Nachrichtentyp im Bereich der Felder fester Länge. Er ist ein Byte groß und hat bei einem Interest immer den Wert `0x1A`. Auf den Nachrichtentyp folgen die Namenskomponenten (Name) des Interests im Bereich der Felder variabler Länge. Im Bereich der Felder variabler Länge folgen dem Namen die Steuerungsfelder.

CCN-IoT setzt folgende Steuerungsfelder um: *AnswerOriginKind*, *Scope/HopCount*, *ChildSelector*, *Lifetime* und *Nonce*. Die Steuerungsfelder sind im Wesentlichen identisch zu den gleichnamigen Steuerungsfeldern, die bei CCNx für den Interest definiert werden. Einige Steuerungsfelder werden in drahtlosen

Sensornetzen nicht benötigt oder lassen sich nicht effizient umsetzen. Aus diesem Grunde wird auf sogenannte Excludes, die bestimmte Namen im Content Store vom Vergleich ausschließen, verzichtet. Der Verzicht auf Excludes wird in „Efficient Data Aggregation with CCNx in Wireless Sensor Networks" [6] begründet.

Mit AnswerOriginKind werden Content Objects im Cache als veraltet gekennzeichnet und ob auf einen Interest auch veraltete Content Objects akzeptiert werden. CCNx definiert für den AnswerOriginKind noch weitere Optionen, beispielsweise dass man nur generierten Content erhalten möchte. Da im Rahmen der Arbeit im drahtlosen Netz nur namenszentrischen Dienste zum Einsatz kommen, ist der Content praktisch immer generiert und gecachter Content hat für gewöhnlich eine kurze Lebensdauer (vgl. Herausforderungen Abschnitt 3.2.1), so dass die Option, nur generierten Content zu erhalten, von CNN-IoT nicht implementiert wird.

Der Scope steuert die Verbreitung von Interests. Die Verbreitung erstreckt sich entweder auf den *(i)* Dämon (der Interest gelangt nur in den Content Store oder die Pending Interest Table), auf die *(ii)* Anwendungen/Dienste oder auf die *(iii)* benachbarten Knoten. Standardmäßig gibt es keine Einschränkung bezüglich der Verbreitung. Der Scope in CCN-IoT ist als optionales Feld variabler Länge implementiert.

Hat der Scope den Wert 0, wird er nur an den Dämon weitergeleitet. Beim einem Wert von Eins wird der Interest nur an die lokalen Anwendungen/Dienste weitergeleitet. Ist kein Scope vorhanden, so wird der Interest nur an die benachbarten Knoten ausgeliefert. Das Weglassen des Scope in der Nachricht spart Platz in der Nachricht und damit letztendlich Bandbreite. Ist Scope > 0, so ist der Wert ein Hop-Count, der bei jeder Übertragung über einen drahtlosen Link vom Dienst für die drahtlose Kommunikation dekrementiert wird. Die möglichen Werte und deren Bedeutung für den Scope fasst Tabelle A.2 zusammen.

Wert	Verbreitung/Scope
0	Dämon
1	Anwendungen/Dämon
kein	Nachbarn
> 0	Keine Einschränkung/Hop Count

Tabelle A.2 – *Werte zur Steuerung der Verbreitung für Interests im drahtlosen Netz (Scope)*

CONTENT OBJECT

Das Content Object ist neben dem Interest der andere Nachrichtentyp von CCN. Bei CCN-IoT hat das Content Object eine maximale Länge von 121 Byte und ist, wie in Abbildung A.5 dargestellt, ähnlich dem Interest aufgebaut.

Content besteht aus Ganzzahltypen (Integer) und Arrays. Bei den Ganzzahltypen wird der Datentyp des Content in den Typ-Bits (Bits Null bis vier in Tabelle A.1)

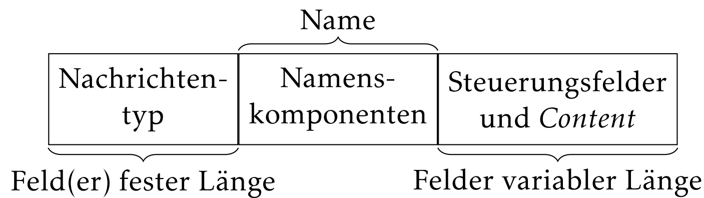


Abbildung A.5 – *Aufbau eines serialisierten Content Objects*

kodiert. Tabelle A.3 zeigt die Bedeutung der Typ-Bits.

Bedeutung		Typ-Bit
Breite	8 Bit	0
	16 Bit	1
	32 Bit	2
Vorzeichen	Negativ	3
	Vorzeichenbehaftet	4

Tabelle A.3 – *Typ-Bits bei Integer Content*

Die Bits Null bis Drei kodieren die Breite des Datentyps unär. Negative Integer werden nicht im Zweierkomplement dargestellt, sondern als Betrag, wobei ein Vorzeichenbit angibt, ob die Zahl negativ ist. Eine nicht vorzeichenbehaftete Zahl kann nicht negativ sein. Kombiniert man alle möglichen Kombinationen aus Tabelle A.3, so ergeben sich insgesamt neun Möglichkeiten für den Typ der Typ-Längen-Felder: Es gibt drei vorzeichenlose Integer-Typen, drei vorzeichenbehaftete Integer-Typen, wobei die vorzeichenbehafteten Typen positiv oder negativ sind.

Die Darstellung negativer Integer mit Vorzeichenbit und Betrag hat den Vorteil, dass die Integer kürzer sind, da führende Null-Bytes eingespart werden. Die Kodierung der Breite und der Vorzeicheneigenschaft zeigen an, ob der Content vom erwarteten Typ ist. Stimmen bei den Rückgabewerten der entfernten Methodenaufrufe die Typen nicht überein, stellt dies einen Fehlerfall dar, über den der Dienstnutzer informiert wird.

Strukturierte Daten als Rückgabewerte werden als Content nach ihrem Variablennamen sortiert in das Content Object in den Bereich der Felder variabler Länge geschrieben. Variablennamen werden nicht im Content Object übertragen, um Platz in der Nachricht zu sparen.

Neben Integer gibt es auch Integerarrays als Rückgabetypen. Namenszentrische Dienste speichern den Typ des Arrays im ersten Byte des Arrays, wobei die Kodierung aus Tabelle A.3 (außer dem dritten Bit) benutzt wird. Negative Integer im Array werden in Zweierkomplementdarstellung gespeichert, ebenso werden führende Null-Bytes gespeichert, da die Integer im Array eine feste Länge haben.

METHOD

Die Klasse `Method` repräsentiert die Methode in CCN-IoT. Methoden werden für die Übertragung serialisiert und die Architektur von CCN-IoT sieht vor, dass serialisierbare Objekte von der Klasse `Buffer` abgeleitet werden. Da Methoden einen Namen haben, werden sie konsequenterweise von der Klasse `Name` abgeleitet. Somit nutzt ein serialisierter Name in CCN-IoT keine Command-Marker-Syntax (vgl. Abbildung 3.5 auf Seite 47), wo der Methodenname mit „`%C1.`“ eingeleitet wird, sondern ein Typ-Längen-Feld für Namen, worin die Länge des Methodennamens kodiert ist. Durch die Ableitung von `Name` werden die Methoden zum Vergleich und Zugriff von Komponenten wiederverwendet. Die Klasse `Method` bietet darüber hinaus Funktionalität zum komfortablen Setzen der Methodennamen ohne explizite Angabe der Typ-Längen-Felder und zum Anhängen der Parameter an die serialisierte Methode.

Die Parameter nutzen dieselbe Kodierung wie der Content, wo Typ-Bits den Datentyp und das Vorzeichenbit negative Zahlen angibt (vgl. Typ-Bits bei Integer Content in Tabelle A.3). Parameter werden in der Methode anhand ihrer Position identifiziert; wobei die Parameter in der Reihenfolge ihrer Namen, wie sie in der Dienstbeschreibung angegeben sind, sortiert werden. Da Dienstanbieter und Dienstnutzer auf der Dienstbeschreibung basieren, können die Parameter wieder ihren Namen zugeordnet werden. Parameter von Methoden werden bei CCN-IoT anstatt im ASCII-Format, wie in der Command-Marker-Syntax, binär kodiert, um Platz in der Nachricht zu sparen. So benötigt die ASCII-Darstellung eines Parameters mit dem Wert `123` in der Command-Marker-Syntax drei Bytes, während die Binärdarstellung `0x7B` hingegen nur ein Byte benötigt.

Auf die Übertragung eines Rückgabetyps (vgl. Aufrufsyntax einer Dienstmethode Abbildung 3.12) in der serialisierten Methode wird verzichtet, da Dienstanbieter und Dienstnutzer der Rückgabetyp durch die Dienstbeschreibung bekannt ist. Auf den Verzicht der Angabe des Rückgabetyps in der serialisierten Methode wird darüber hinaus Platz in der Nachricht gespart. Methoden sind als Komponenten ein Teil des Namens und daher sind sie in CCN-IoT maximal 32 Byte groß (vgl. Kodierung der Typ-Längen-Felder in Tabelle A.1). Abbildung A.6 zeigt den Aufbau einer serialisierten Methode.

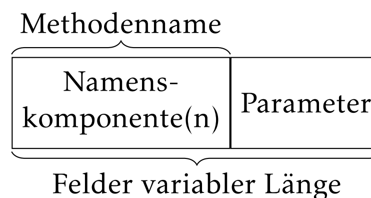


ABBILDUNG A.6 – *Aufbau einer serialisierten Methode*

A.1.3 BUFFER-ITERATOREN

Alle Buffer-Iteratoren sind als Klassen implementiert. Die Basisklasse aller Buffer-Iteratoren ist `FastBufferIterator`, da diese Klasse den minimalsten

Funktionsumfang für einen Buffer-Iterator implementiert. BufferIterator ist die Basis für alle Iteratoren, die in der Lage sind, den Buffer zu modifizieren. FastIteratorTlvBase und BufferIteratorTlvBase implementieren Methoden, um auf Integer oder Arrays in Typ-Längen-Feldern zuzugreifen. Die Klassenhierarchie der Buffer-Iterator Basisklassen ist in dem UML-Diagramm in Abbildung A.7 zusammengefasst.

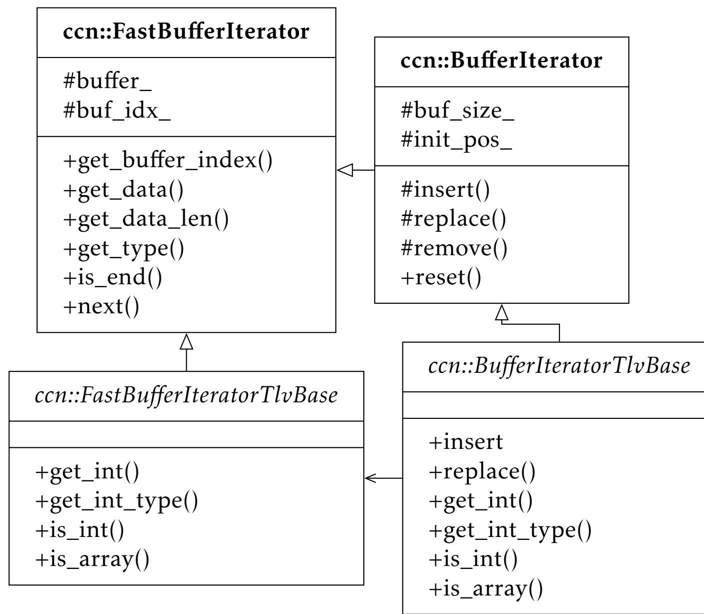


Abbildung A.7 – *Klassenhierarchie der Buffer-Iterator Basisklassen*

BufferIteratorTlvBase benutzt für den lesenden Zugriff die Implementierung von FastIteratorTlvBase, was der Assoziationspfeil ausdrückt. Diese Assoziation wird durch die Beschränkung bei der Entwicklung mit C++ auf eingebetteten Systemen notwendig, da keine Mehrfachvererbung und virtuelle Methoden möglich sind. Die Basisklassen in Abbildung A.7 sind abstrakt und alle konkreten Implementierungen für die Iteratoren für den schnellen, nur lesenden Zugriff werden von der Basisklasse FastBufferIterator abgeleitet.

Die von FastBufferIterator abgeleitete Klasse FastNameIterator greift auf die Namenskomponenten zu. Klasse FastParamIterator ist von FastIterator-TlvBase abgeleitet, da sie auf Parameter von Methoden zugreift. Beide Iteratoren überschreiben die Methoden next() und is_end() der Basisklasse, um in den jeweiligen Bereichen des Buffers von Namen und Parametern zu operieren. Das UML-Diagramm in Abbildung A.8 zeigt FastNameIterator und FastParam-Iterator in der Klassenhierarchie der Buffer-Iteratoren für den schnellen, nur lesenden Zugriff.

Neben den Buffer-Iteratoren für den schnellen, lesenden Zugriff gibt es noch die Buffer-Iteratoren für den modifizierenden Zugriff. Bei den Iteratoren für den modifizierenden Zugriff sind in der Klasse NameIterator Methoden zum Lesen und Schreiben des Namens implementiert. Dieser Iterator speichert auch die Länge des Namens und die Anzahl der Komponenten. Mit dem Param-

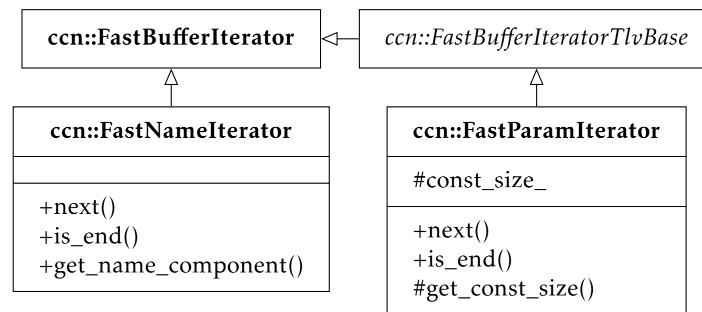


Abbildung A.8 – *Klassenhierarchie der Buffer-Iteratoren für den schnellen, nur lesenden Zugriff*

`Iterator` werden die Parameter von Instanzen der Klasse `Method` modifiziert. `ParamIterator` implementiert keine Methoden, da die Basisklasse `FastParamIterator` bereits alle benötigten Methoden Implementiert. Abbildung A.9 zeigt ein UML-Klassendiagramm der Iteratoren für den modifizierenden Zugriff.

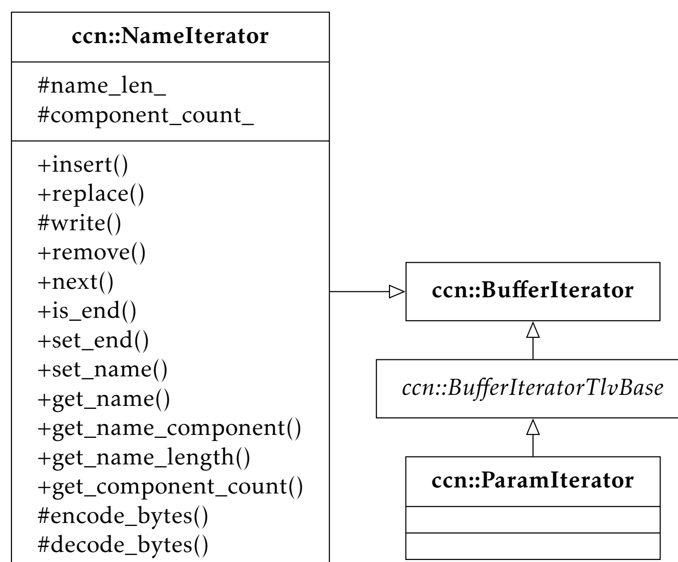


Abbildung A.9 – *Klassenhierarchie der Buffer-Iteratoren für den modifizierenden Zugriff*

A.1.4 CCN-DATENSTRUKTUREN IN CCN-IOT

Dieser Abschnitt führt zunächst die generischen Datenstrukturen Set und Vec ein, die Grundlage von Content Store, Pending Interest Table und Forwarding Information Base sind. Die Implementierungen von Content Store, Pending Interest Table und Forwarding Information Base werden im Anschluss vorgestellt.

SET UND VEC

Während der Entwicklung von CCN-IoT zeigte sich, dass eine sortierte Speicherung von Elementen notwendig ist. So werden Content Objects in Content

Store und Präfixe in der Forwarding Information Base sortiert gespeichert. Mit der Klasse `Set` wurde in CCN-IoT eine generische Container-Datenstruktur entwickelt, die die sortierte Speicherung von Elementen erlaubt. Eine generische Container-Datenstruktur nimmt beliebige Datentypen eines Typs auf, wobei der Typ, den sie aufnimmt, zur Entwicklungszeit festgelegt wird.

Die Sortierung von `Set` wird mittels *Einfügesortierung* [130] realisiert. Aus diesem Grunde implementieren die Klassen, deren Instanzen in ein `Set` einsortiert werden sollen, den Less-Operator (`operator<()`). `Set` stellt eine ähnliche Schnittstelle und Funktionalität wie die Klasse `set` aus der C++ Standard Template Library (STL) [128] zur Verfügung.

Das sortierte Speichern von Elementen ist mit einem gewissen Aufwand verbunden und auch nicht immer notwendig oder gewünscht. So ist es zum Beispiel für die Pending Interest Table nicht erforderlich, die Interests zu sortieren. Aus diesem Grunde wurde neben `Set` die generische Container-Datenstruktur `Vec` entwickelt. Sie ist aufgebaut wie ein Array und unterstützt Anfügen von Elementen am Anfang und am Ende und wird zur Realisierung von Queues [130] verwendet. Die Datenstruktur `Vec` hat eine ähnliche Schnittstelle wie die Klasse `vector` aus der STL.

Im Gegensatz zu den Pendants aus der STL ist es bei `Set` und `Vec` jedoch möglich, während des Iterierens über die Datenstrukturen, Elemente zu entfernen. Das Entfernen von Elementen während des Iterierens ist nützlich, da Elemente gleich während der Prüfung in der Schleife entfernt werden. Anderenfalls wäre es notwendig, die zu entfernenden Elemente zwischenzuspeichern und im Anschluss an den Schleifendurchlauf zu entfernen, was zusätzlichen Aufwand bedeutet.

Zum Entfernen von Elementen wird die Methode `erase()` verwendet. Die Methode `erase()` gibt einen Iterator zurück, der beim nächsten Schleifendurchlauf auf das Element zeigt, welches vor dem entfernten Element steht. Daher ist es wichtig, dass die Ausführung in der Schleife nach dem Entfernen unterbrochen und der Iterator zum nächsten Element bewegt wird. (Hinweis: Der Iterator von `Set` und `Vec` ist kein Buffer-Iterator; er entspricht in Schnittstelle und Funktionalität den Iteratoren aus der STL.) Quelltext A.1 zeigt ein Beispiel für das Entfernen von Elementen während des Iterierens am Beispiel der Instanz `my_set_` der Klasse `Set`. Ist die Bedingung `condition` in Zeile 4 erfüllt, dann wird das Element in Zeile 5 entfernt. In Zeile 6 wird die Ausführung der Schleife nach dem Entfernen des Elements unterbrochen und beim nächsten Element fortgesetzt.

Die Elemente, wie Interests oder Content Objects, die in `Set` oder `Vec` gespeichert werden, sind für ressourcenbeschränkte Geräte verhältnismäßig groß. Das Verschieben von großen Elementen im Speicher beim Hinzufügen und Entfernen von Elementen ist aufwändig. Aus diesem Grunde fügen `Set` und `Vec` die Elemente sequenziell in ein Array, das *Elementarray* ein, und belassen die Elemente an ihrer ursprünglichen Position. Die Position der Elemente, so wie sie der Anwender der Klassen `Set` oder `Vec` sieht, wird durch ein *Indexarray* realisiert. Im Indexarray werden die Indizes der Elemente im Elementarray der Sortierung der Elemente folgend im Indexarray abgelegt. Ungültige Indizes im Indexarray

```
1 for ( my_set_iter_t it = my_set_.begin();
2       it != my_set_.end(); ++it )
3 {
4     if ( condition ) {
5         it = my_set_.erase( it );
6         continue;
7     }
8     // ...
9 }
```

Quelltext A.1 – *Entfernen von Elementen aus der Datenstruktur* Set *während des Iterierens über die Datenstruktur*

zeigen hinter das Elementarray, die Belegung des Elementarrays mit gültigen oder ungültigen Einträgen wird mit einem (Belegungs-)Bitfeld angezeigt. Das Elementarray, das Indexarray, und das Belegungs-Bitfeld sind in Abbildung A.10 dargestelt. Die Elemente im Elementarray sind mit Großbuchstaben dargestellt. Element A steht zum Beispiel an Index 5 im Elementarray. Nach der alphabetischen Sortierung kommt Element A an erster Stelle, daher befindet sich der Index 5 an erster Stelle im Indexarray.

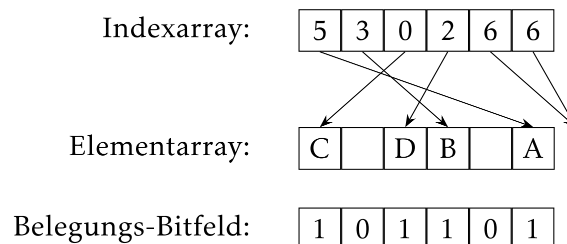


Abbildung A.10 – *Indexmenge, Elemente und Belegungs-Bitfeld von Set und Vec*

CONTENT STORE

Der Content Store ist als Klasse implementiert und speichert sogenannte *Local Content Objects* in einem Set. Local Content Objects erweitern Content Objects um die Zeit, wo die sogenannten *Freshness Seconds* des Content Objects das letzte mal geprüft wurden. Die Freshness Seconds werden bei der Suche nach einem Content Object im Content Store oder beim Hinzufügen eines Content Objects um die Differenz der aktuellen Zeit mit der Zeit der letzten Prüfung dekrementiert. Ist die Zeit auf null dekrementiert, ist das Content Object veraltet. Veraltete Content Objects werden bei Bedarf aus dem Content Store gelöscht, wenn ein neues Content Object eingefügt wird und kein Platz mehr im Content Store ist. Abbildung A.11 zeigt das Klassendiagramm des Content Store von CCN-IoT. Mit der Operation `lookup()` wird nach einem Content Object gesucht und mit `add()` wird ein Content Object hinzugefügt. Mit `mark_stale()` werden Content Objects im Content Store als veraltet gekennzeichnet, um Anwendungen die Möglichkeit zu geben, Content Objects aus dem Content Store zu entfernen.

Die `update()`-Methode wird bei jedem Zugriff aufgerufen und aktualisiert die Einträge des Content Store.

Die Klasse `LocalContentObject` repräsentiert die Klasse für die Local Content Objects. Die Vergleichsoperatoren `operator<()` und `operator==()` vergleichen die Namen der Content Objects die in `content_object_` des `LocalContentObject` gespeichert sind. Klassenvariable `last_check_time_` speichert die Zeit der letzten Prüfung.

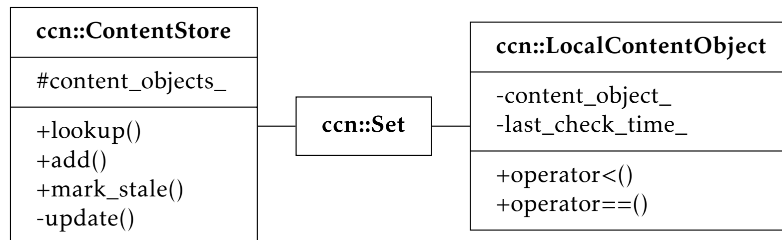


Abbildung A.11 – *Content Store*

PENDING INTEREST TABLE

Die Pending Interest Table (PIT) ist als Klasse implementiert und speichert die Einträge unsortiert in einem `Vec`. PIT-Einträge bestehen aus dem Interest und der Zeit, an dem der Interest vom Dämon empfangen wurde. Bei der Suche nach einem Präfix in der PIT werden alle PIT-Einträge geprüft, ob die Differenz zwischen aktueller Zeit und Ankunftszeit größer oder gleich der Lebenszeit des Interests ist. Ist das der Fall, wird der PIT-Eintrag gelöscht. Werden mehrere Interests mit dem gleichen Namen empfangen, wird aus Effizienzgründen nicht für jeden Interest ein PIT-Eintrag angelegt, sondern ein Bit in einem Bitfeld gesetzt. Jedes Bit steht dabei für ein Face auf dem Knoten. Abbildung A.12 zeigt das Klassendiagramm der Pending Interest Table von CCN-IoT. PIT-Einträge werden durch die Klasse `PitEntry` repräsentiert. In der Klassenvariable `arrival_time_` von `PitEntry` wird die Ankunftszeit des Interests gespeichert. Der Interest wird in der Klassenvariable `interest_` gespeichert. Mit der `lookup()`-Operation der Klasse `PendingInterestTable` wird geprüft, ob es einen Interest mit einem bestimmten Präfix gibt. Methode `unregister_face()` wird bei der Entfernung von Faces aufgerufen, um das Face aus den PIT-Einträgen zu entfernen. Wenn es den Interest nicht gibt, wird der Interest mit `lookup()` in die Pending Interest Table übernommen. Die Klassenvariable `face_mask_` von `PitEntry` ist das Bitfeld für die Faces, mit denen der Eintrag assoziiert ist.

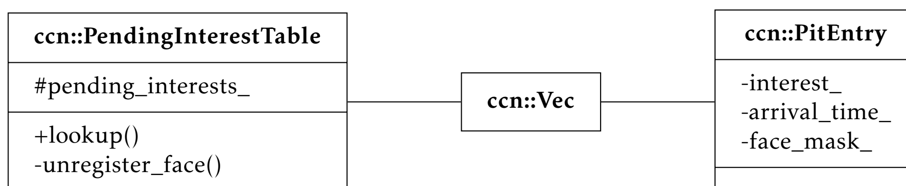


Abbildung A.12 – *Pending Interest Table*

FORWARDING INFORMATION BASE

Die Forwarding Information Base (FIB) speichert die Einträge anhand des Präfixes nach der Shortlex-Ordnung sortiert in einer Instanz der Klasse `Set`. Mehrere Faces teilen sich einen FIB-Eintrag, wenn sie sich für den gleichen Präfix registrieren, wobei hier die gleiche Strategie wie bei den PIT-Einträgen mit dem Bitfeld (Klassenvariable `face_mask_`) angewendet wird. Abbildung A.13 zeigt das Klassendiagramm der Forwarding Information Base von CCN-IoT. Mit der Methode `lookup()` werden die Faces ermittelt, zu denen ein Interest weitergeleitet wird. Anwendungen benutzen `register_face()` und `unregister_face()`, um Faces an der Forwarding Information Base an- und abzumelden. FIB-Einträge werden durch die Klasse `FibEntry` repräsentiert. Ein FIB-Eintrag hat eine Klassenvariable `prefix_`, wo der Präfix gespeichert wird, mit dem der Eintrag assoziiert ist.

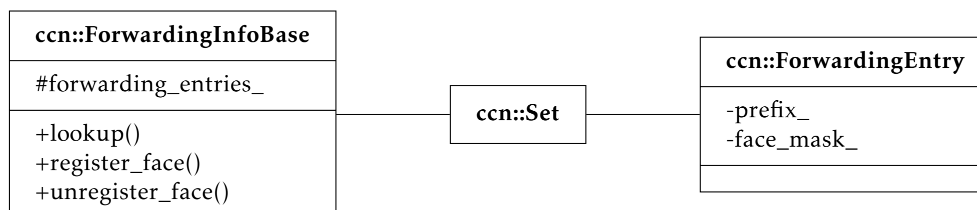


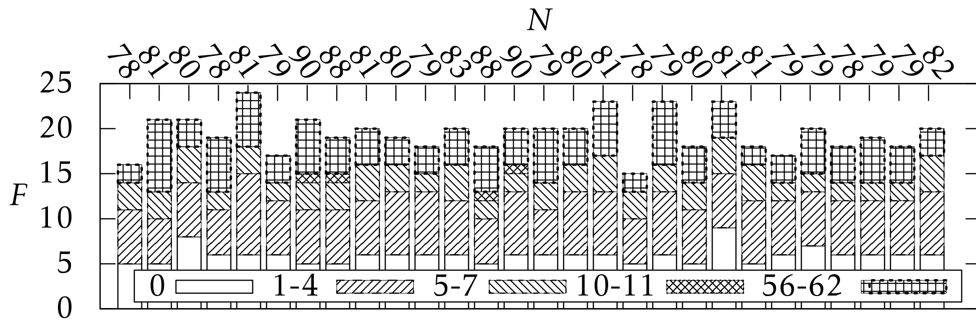
Abbildung A.13 – *Forwarding Information Base*

A.2 NDN-TESTBED

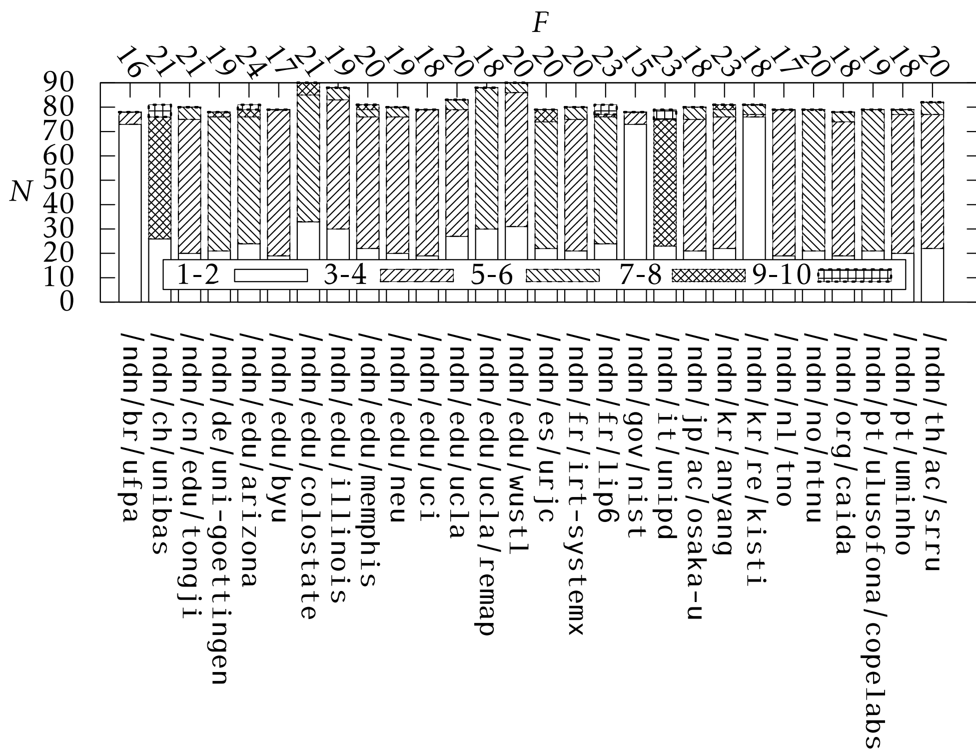
In diesem Abschnitt werden die Abbildungscharakteristik von Faces zu Präfixen und umgekehrt sowie die Länge der Präfixe im NDN-Testbed betrachtet. Die den Messungen zugrundeliegenden Informationen sind Stand November 2016 und stammen von der NDN-Webseite `https://named-data.net/ndn-testbed/`. Folgende Messergebnisse wurden in „Memory Efficient Forwarding Information Base for Content-Centric Networking“ [1] veröffentlicht.

A.2.1 ABBILDUNGSCHARAKTERISTIK

Unter dem Begriff *Abbildungscharakteristik* wird in dieser Arbeit die Verteilung der Präfixe auf die Faces beziehungsweise die Verteilung der Faces auf die Präfixe verstanden (vgl. Abbildungen von Präfixen zu Faces in der FIB in Abbildung 4.26). Die Abbildungscharakteristik ist wichtig für die Auswahl der Strategien zur Speicheroptimierung der Forwarding Information Base, vorgestellt in Abschnitt 4.3.5. Da es bisher kaum belastbare Aussagen gibt, wie die Abbildungscharakteristik in zukünftigen inhaltszentrischen Netzen aussieht, wurde die Abbildungscharakteristik im NDN-Testbed untersucht. Abbildung A.14 zeigt die im NDN-Testbed gemessenen Abbildungscharakteristika. Jede Säule in Abbildung A.14(a) repräsentiert einen der 28 Knoten im NDN-Testbed. Die Namen der Knoten sind aus Platzgründen nur auf der x-Achse von Abbildung A.14(b) aufgetragen. Die Höhe jeder Säule in Abbildung A.14(a) steht für die Anzahl der Faces F, die auf der y-Achse aufgetragen ist.



(a) Zuordnung von Präfixe zu Faces



(b) Zuordnung von Faces zu Präfixe

Abbildung A.14 – *FIB-Abbildungscharakteristika im NDN-Testbed*

Segmente der Säulen zeigen die Anzahl der Faces, denen eine bestimmten Anzahl an Präfixe zugeordnet ist. Der erste Knoten `/ndn/br/ufpa` hat fünf Faces, denen kein Präfix zugeordnet ist und zwei Faces, denen zwischen 56 und 62 Präfixe zugeordnet sind. Die obere x-Achse zeigt die Anzahl der Präfixe N in der Forwarding Information Base des jeweiligen Knotens.

Abbildung A.14(b) ist ähnlich zu Abbildung A.14(a), die y-Achse zeigt hier allerdings die Anzahl der Präfixe N und die obere x-Achse die Anzahl der Faces F. Die unterschiedlichen Segmente der Säulen zeigen die Anzahl der Präfixe, die einem bis zwei, drei bis vier, fünf bis sechs, sieben bis acht beziehungsweise neun bis zehn Faces zugewiesen sind. Bei der Forwarding Information Base des ersten Knotens in Abbildung A.14(b) gibt es 70 Präfixe, die einem oder zwei Faces zugeordnet sind.

Die Abbildungscharakteristika bestimmen sich wie folgt: *(i)* Ist $N \approx F$ und die Werte in den Säulen sind klein (ungefähr Eins), dann handelt es sich um eine 1-1-Abbildung. *(ii)* Ist $N \ll F$ und die Werte in den Säulen sind im ersten Graphen (Abbildung A.14(b)) klein und in dem zweiten Graphen (Abbildung A.14(a)) groß, dann handelt es sich um eine 1-n-Abbildung. *(iii)* Ist $N \gg F$ und sind die Werte in den Säulen den ersten Graphen (Abbildung A.14(b)) groß und im zweiten Graphen (Abbildung A.14(a)) klein, dann ist es eine n-1-Abbildung.

Eine Klassifizierung nach Abbildungscharakteristik wie in Abbildung 4.26 aus Abbildung A.14 abzuleiten, ist schwierig. Eine Teilmenge der Abbildungen von Präfixen zu Faces legen eine 1-1-Abbildung nahe, andere hingegen eine n-1-Abbildung. Aus diesem Grunde werden in Abschnitt Abschnitt 4.3.5 alle Abbildungscharakteristika betrachtet.

Bei den Evaluation der Strategien zur Speicheroptimierung der Forwarding Information Base in Abschnitt 4.3.6 wurden für die Anzahl der Faces F Werte zwischen 10 und 15 und für Präfixe N Werte zwischen 50 und 60 angenommen. Diese Werte liegen ungefähr in der Mitte der beobachteten Werte für die Anzahl der Faces (y-Achse in Abbildung A.14(a)) und Präfixe (y-Achse in Abbildung A.14(b)) im NDN-Testbed.

A.2.2 LÄNGE DER PRÄFIXE

Neben der Verteilung der Präfixe auf die Faces und umgekehrt wurden die Längen der Präfixe im NDN-Testbed untersucht. Die Motivation dieser Untersuchung ist, festzustellen, wie lang Präfixe in inhaltszentrischen Netzen werden. Je länger Präfixe in drahtlosen Netzen werden, desto effizienter sind die Strategien zur Speicheroptimierung der Forwarding Information Base in Abschnitt 4.3.5. Die Längen der Präfixe im NDN-Testbed zeigt das Histogramm in Abbildung A.15. Auf der x-Achse sind die Präfixlängen in Byte aufgetragen, auf der y-Achse die Häufigkeit der jeweiligen Präfixlängen, wie sie im NDN-Testbed betrachtet wurden. Es gibt zwei Häufungen an Präfixlängen im NDN-Testbed, eine bei 11 bis 16 Byte und eine bei 29 bis 34 Byte. Aufgrund der Häufung bei 11 bis 16 Byte wurde in der Evaluation in Abschnitt 4.3.6 eine Präfixlänge von 15 Byte angenommen. Auf eine Untersuchung mit Präfixlängen von beispiels-

weise 30 Byte wurde verzichtet, da die Strategien zur Speicheroptimierung der Forwarding Information Base mit steigender Präfixlänge effizienter werden und somit die Bewertung an Aussagekraft verliert.

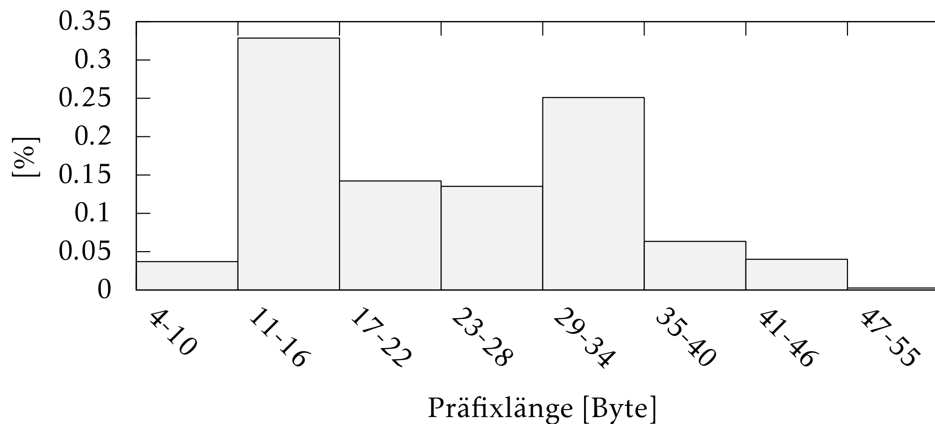


Abbildung A.15 – *Präfixlängen im NDN-Testbed*

A.3 DIENSTBESCHREIBUNG UND WERKZEUGUNTERSTÜTZUNG

Dieser Abschnitt enthält ergänzende Informationen zu der Dienstbeschreibung und Werkzeugunterstützung. Abschnitt A.3.1 zeigt das Schema der Dienstbeschreibung, das in Abschnitt 5.2 in Auszügen dargestellt ist, vollständig. In Abschnitt A.3.2 wird ein Beispiel für eine Dienstbeschreibung eingeführt. Abschnitt A.3.4 zeigt Implementierungsdetails der Codegeneratoren sowie ein Beispiel für generierten Quellcode.

A.3.1 VOLLSTÄNDIGES SCHEMA DER DIENSTBESCHREIBUNG

Das vollständige JSON-Schema der Dienstbeschreibung zeigt Quelltext A.2. Bei allen Quelltexten in dieser Arbeit ist zu beachten, dass Zeilennummern fortlaufend mit Eins beginnend vergeben werden. Somit sind die jeweiligen Nummern an den entsprechenden Zielen in den Quelltexten 5.1 bis 5.4 verschieden zu denen in Quelltext A.2.

Quelltext A.2 – *Vollständiges JSON-Schema der Dienstbeschreibung*

```
1 {
2   "type": "object",
3   "properties": {
4     "type": { "type": "string" },
5     "name": { "type": "string" },
6     "version": { "type": "string" },
7     "uri": {
8       "type": "string",
```

```
9         "pattern": "^(ccnx:)(/[A-Za-z0-9-._~\\?#]+)+$"
10      },
11      "doc": { "type": "string" },
12      "types": {
13        "type": "object",
14        "patternProperties": {
15          "^[A-Za-z_][A-Za-z_0-9]*$": {
16            "type": "object",
17            "patternProperties": {
18              "^[A-Za-z_][A-Za-z_0-9]*$": {
19                "type": "object",
20                "properties": {
21                  "doc": { "type": "string" },
22                    "type": {
23                      "type": "string",
24                      "pattern": "^[iu](8|16|32)(\\[\\])*$|^[A
                            -Za-z_][A-Za-z_0-9]*$"
25                    }
26                  },
27                  "additionalProperties": false,
28                  "required": [ "type" ]
29                }
30              },
31              "minProperties": 1
32            }
33          },
34          "additionalProperties": false
35        },
36        "methods": {
37          "type": "object",
38          "patternProperties": {
39            "^[A-Za-z_][A-Za-z_0-9]*$": {
40            "type": "object",
41            "properties": {
42              "doc": { "type": "string" },
43              "params": {
44                "type": "object",
45                "patternProperties": {
46                  "^[A-Za-z_][A-Za-z_0-9]*$": {
47                    "type": "object",
48                    "properties": {
49                      "doc": { "type": "string" },
50                      "type": {
51                        "type": "string",
52                        "pattern": "^[iu](8|16|32)(\\[\\])*$"
53                      }
54                    },
55                    "additionalProperties": false,
56                    "required": [ "type" ]
57                  }
58                },
59                "minProperties": 1
60              },
61              "ret_type": {
```

```
62                 "type": "string",
63                 "pattern": "^[iu](8|16|32)(\\[\\])*$|^[A-Za-z_
                      ][A-Za-z_0-9]*$"
64               }
65             }
66           }
67         },
68         "additionalProperties": false
69       }
70     },
71     "additionalProperties": false,
72     "required": [ "type", "name", "version", "uri" ],
73     "dependencies": { "types": [ "methods" ] }
74   }
```

A.3.2 DIENSTBESCHREIBUNG BEISPIEL

In diesem Abschnitt wird ein Beispiel für eine Beschreibung eines fiktiven namenszentrischen Dienstes für eine Klimaüberwachung und -regelung in Seecontainern präsentiert, die dem Schema der Dienstbeschreibung aus Quelltext A.2 folgt. Der Quelltext der Dienstbeschreibung wird zunächst in Auszügen dargestellt. Das vollständige Beispiel findet sich am Ende diese Abschnitts in Quelltext A.6. Die Teile der Dienstbeschreibung werden in der Reihenfolge *(i)* Metadaten, *(ii)* Datentypen und *(iii)* Methoden vorgestellt.

Metadaten sind Begleitinformationen eines Dienstes oder auch der Dienstbeschreibung selbst. Die Metadaten bestehen aus den fünf JSON-Eigenschaften: *(i)* `type`, *(ii)* `name`, *(iii)* `version`, *(iv)* `uri` und *(v)* `doc`, die in Abschnitt 5.2 erklärt werden. Quelltext A.3 zeigt die *Metadaten* der Dienstbeschreibung für den fiktiven Dienst zur Klimaüberwachung und -regelung in Seecontainern.

```
1  {
2    "type": "ncs/desc",
3    "name": "de.fhl.sensors",
4    "version": "0.1",
5    "uri": "ccnx:/de/oceancarrier/position/ncs/subs/container/
          ncs/desc",
6    "doc": "Acquire and controls conditions in a maritime
          container",
       ⋮
7
8  }
```

QUELLTEXT A.3 – *Beispiel Dienstbeschreibung: Metadaten*

Nach den Metadaten werden die *zusammengesetzten Datentypen* definiert. Der Dienst für die Klimaüberwachung und -regelung der Seecontainer definiert drei zusammengesetzte Datentypen. Quelltext A.4 zeigt die drei definierten Datentypen *(i)* `Conditions`, *(ii)* `Temperature` und *(iii)* `Humidity`. Datentyp `Conditions` setzt sich aud den beiden anderen Datentypen `Temperature` und `Humidity` zusammen.

```
1      ⋮
2  "types": {
3    "Conditions": {
4      "temperature": { "type": "Temperature" },
5      "humidity": { "type": "Humidity" }
6    },
7    "Temperature": {
8      "value": {
9        "doc": "Temperaure in °C",
10       "type": "i16"
11     },
12     "response": {
13       "doc": "Response of challenge",
14       "type": "u8[]"
15     }
16   },
17   "Humidity": {
18     "value": {
19       "doc": "Relative humidity in %",
20       "type": "u16"
21     },
22     "response": { "type": "u8[]" }
23   }
24 },
25     ⋮
```

Quelltext A.4 – *Beschreibung der Typen des Diensten zur Klimaregelung von Seecontainern*

Bei der Verarbeitung eines JSON-Dokuments ist die Reihenfolge der Verarbeitung der Eigenschaften nicht definiert. Daher ist es, im Gegensatz zu den Definitionen in den meisten Programmiersprachen möglich, die Typen in beliebiger Reihenfolge zu definieren. So wird der Datentyp `Conditions` vor den Typen definiert, aus denen er zusammengesetzt ist. In der Dienstbeschreibung ist bei den Typen jede Reihenfolge gültig. Da in den meisten Programmiersprachen Datentypen vor ihrer Benutzung definieren, stellt der Codegenerator sicher, dass die Typen im generierten Code in der richtigen Reihenfolge definiert werden.

`Temperature` und `Humidity` definieren jeweils zwei Felder mit den Namen `value` und `response`. Datentyp `Temperature` repräsentiert den Rückgabewert eines Temperatursensors und Datentyp `Humidity` einen den Rückgabetyp eines Feuchtigkeitssensors in einem Seecontainer. Für die Felder mit der Bezeichnung `value` gibt es bei beiden Datentypen eine kurze Freitext-Beschreibung mit der `doc`-Eigenschaft. Feld `value` enthält jeweils den aktuell gemessenen Temperatur- beziehungsweise Feuchtigkeitswert. Das Feld mit der Bezeichnung `response` enthält die Response auf eine Challenge, die beim Aufruf der Dienstmethode übergeben wird, da sich die Sensoren in dem Beispiel mittels eines Challenge-Response Verfahrens authentifizieren (vgl. Dienste zur Authentifizierung in Abschnitt 7.6). In dem Beispiel wird zusätzlich angenommen, dass der

```
1       ⋮
2  "methods": {
3    "get_temp": {
4      "doc": "Returns container inner temperature",
5      "params": {
6        "challenge": {
7          "type": "u8[]",
8          "doc": "Challenge for sensor response"
9        }
10     },
11     "ret_type": "Temperature"
12   },
13      ⋮
14 }
```

Quelltext A.5 – *Beschreibung der Methode zur Temperaturermittlung des Dienstes zur Klimaregelung von Seecontainern*

Status der Sensoren ebenfalls Teil der Challenge ist. So teilt der Dienstanbieter dem Dienstnutzer zusätzlich mit, ob der gelieferte Sensorwert gültig ist.

Nach den zusammengesetzten Typen werden die *Dienstmethoden* definiert. Quelltext A.5 zeigt exemplarisch die Beschreibung der Methode `get_temp`. Sie hat einen Parameter `challenge`, der beim Aufruf die Challenge übergeben bekommt, die der Dienstanbieter lösen muss. Der Rückgabewert der Dienstmethode ist vom Typ `Temperature`. Dieser enthält mit in dem Feld `response` die Response der Challenge. Ist die Response richtig berechnet worden, ist der Temperaturwert gültig.

VOLLSTÄNDIGE DIENSTBESCHREIBUNG

Die vollständige Dienstbeschreibung des Beispiels, aus dem vorherigen Abschnitt Abschnitt A.3.2, zeigt Quelltext A.6. Hierbei ist zu beachten, dass die Zeilennummern bei allen Quelltexten fortlaufend mit Eins beginnend vergeben werden. Somit sind Zeilen an den jeweiligen Nummern in den Quelltexten A.3 bis A.5 verschieden zu denen in Quelltext A.6.

Quelltext A.6 – *Vollständiges Beispiel der Dienstbeschreibung*

```
1  {
2    "type": "ncs/desc",
3    "name": "de.fhl.sensors",
4    "version": "0.1",
5    "uri": "ccnx:/de/oceancarrier/position/ncs/sc/container/
        ncs/desc",
6    "doc": "Acquire and controls conditions in a maritime
        container",
7    "types": {
```

```
 8      "Conditions": {
 9        "temperature": { "type": "Temperature" },
10        "humidity": { "type": "Humidity" }
11      },
12      "Temperature": {
13        "value": {
14          "doc": "Temperaure in °C",
15          "type": "i16"
16        },
17        "response": {
18          "doc": "Response of challenge",
19          "type": "u8[]"
20        }
21      },
22      "Humidity": {
23        "value": {
24          "doc": "Relative humidity in %",
25          "type": "u16"
26        },
27        "response": { "type": "u8[]" }
28          }
29    },
30    "methods": {
31      "get_temp": {
32        "doc": "Returns container inner temperature",
33        "params": {
34          "challenge": {
35            "type": "u8[]",
36            "doc": "Challenge for sensor response"
37          }
38        },
39        "ret_type": "Temperature"
40      },
41      "get_hum": {
42        "doc": "Returns container inner humidity",
43          "params": {
44                    "challenge": { "type": "u8[]" }
45          },
46          "ret_type": "Humidity"
47      },
48      "get_cond": {
49        "doc": "Returns container inner conditions",
50          "params": {
51            "challenge": { "type": "u8[]" }
52          },
53          "ret_type": "Conditions"
54      },
55      "set_temp": {
56        "doc": "Set container inner temperature, returns
             status code",
57          "params": {
58            "value": {
59              "type": "i16",
60                "doc": "New temperature value"
```

```
61            },
62            "response": {
63              "type": "u8[]",
64              "doc": "Temporary response value"
65            }
66          },
67          "ret_type": "u8"
68      },
69      "set_airc": {
70        "doc": "Set container air circulation",
71          "params": {
72            "value": {
73              "type": "i16",
74              "doc": "intensity/direction of container air
                    circulation"
75            },
76            "response": { "type": "u8[]"}
77          },
78          "ret_type": "u8"
79      }
80    }
81  }
```

A.3.3 IMPLEMENTIERUNG DER DATENTYPEN UND METHODEN IN DESCRIPTION

Dieser Abschnitt geht auf die Details der Implementierung von Methoden und Datentypen der Description-Komponente ein. Generatoren für die Datentypen werden dabei durch die Klasse Type und Generatoren für Methoden durch die Klasse `Method` repräsentiert.

TYPE

Datentypen werden in der Dienstbeschreibung in der Eigenschaft `types` definiert. Weiterhin hat jedes Feld eines zusammengesetzten Datentyps, jeder Parameter sowie der Rückgabetyp einer Methode einen Datentyp. Aus diesem Grunde bietet sich eine gemeinsame Oberklasse für alle Kategorien von Datentypen an. Die Kategorien von Datentypen in der Dienstbeschreibung für namenszentrische Dienste sind: *(i)* primitiver Datentyp, *(ii)* Array-Datentyp und *(iii)* zusammengesetzter Datentyp. Die gemeinsame Oberklasse hat den Namen Type. Klasse `SimpleType` repräsentiert die primitiven Datentypen, Klasse `ArrayType` repräsentiert die Array-Datentypen und Klasse `ComplexType` repräsentiert die zusammengesetzten Datentypen. Ein Array-Datentyp besteht bei den namenszentrischen Diensten für das Internet der Dinge aus primitiven Datentypen. Aus diesem Grund wird er von den primitiven Datentypen abgeleitet. Der zusammengesetzte Datentyp besteht wiederum aus Datentypen, den Feldern. Für den Zugriff auf die Felder bietet `ComplexType` einen Iterator an.

In Abbildung A.16 sind die Klassen der Datentypen in Description und deren Beziehungen zu anderen Klassen zusammengefasst. Klasse Type ist als Interface ausgeführt. Alle Klassen, die Type implementieren müssen die Methode

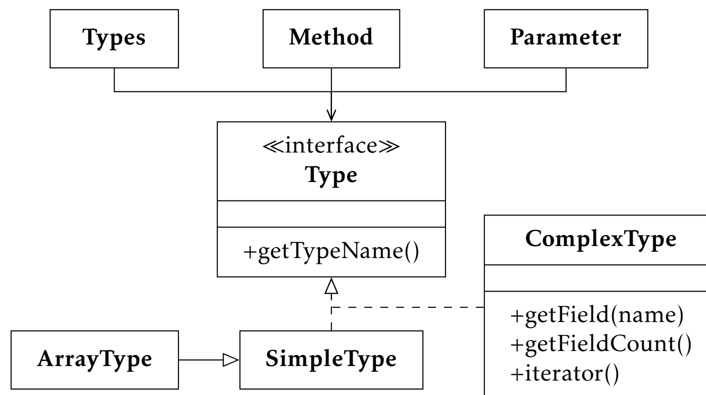


Abbildung A.16 – *Klassen zur Darstellung der Datentypen in Description und deren Beziehungen zu anderen Klassen*

`getTypeName()` implementieren. Über diese Methode erfahren die Codegeneratoren die Bezeichnung des jeweiligen Datentyps. `SimpleType`, `ArrayType` und `ComplexType` implementieren dieses Interface, da jede Klasse eine Kategorie von Datentypen repräsentiert. `ArrayType` erbt von `SimpleType`, da `ArrayType` eine Erweiterung von `SimpleType` ist. Die Klassen `Parameter` und `Method`, die wie die Klasse `Types` die Klasse `Type` verwenden, gehören zur Beschreibung der Dienstmethode und werden im folgenden Abschnitt eingeführt.

METHOD

Methoden werden in der Dienstbeschreibung in der Eigenschaft `methods` definiert. Eine Methode hat eine Parameterliste (die auch leer sein kann) und einen Rückgabewert. Jeder Parameter hat einen Datentyp. Eine Methode wird durch die Klasse `method`, die Parameterliste durch die Klasse `Parameters` und Parameter durch die Klasse `Parameter` repräsentiert. Abbildung A.17 fasst die Klassen zur Darstellung von Methoden in Description zusammen. Klasse `Methods` arbeitet die Methoden der Beschreibung ab, indem für jede Methode ein Objekt vom Typ `Method` erstellt wird. `Method` wiederum nutzt die Klasse `Parameters`, die durch die Parameterliste geht und für jeden Parameter ein Objekt vom Typ `Parameter` erstellt. Methoden der Beschreibung und deren Parameter sind über die jeweiligen Methoden beziehungsweise über die Iteratoren erreichbar.

A.3.4 IMPLEMENTIERUNG DER CODEGENERATOREN

Die Codegeneratoren sind in Java implementiert und in Paketen organisiert. Auch die Description-Komponente ist in der vorliegenden Implementierung als Paket ausgeführt. Alle Codegeneratoren sind im Paket `Tools` zusammengefasst. Es enthält das Interface `Generator` sowie die drei abstrakten Generatorklassen. Alle abstrakten Generatorklassen implementieren die Schnittstelle `Generator`, welche die Methode `generate()` bereitstellt. Methode `generate()` gibt den generierten Quellcode zurück. Generatoren für eine bestimmte Plattform – Java oder C++ für CCN-IoT – implementieren die abstrakten Klassen `ServiceProviderProxyGenerator`, `ServiceProviderStubGenerator` und

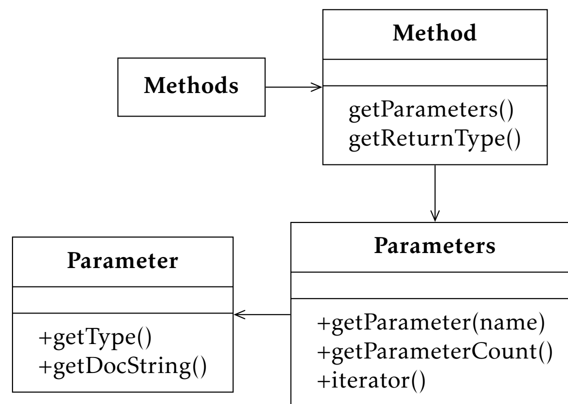


Abbildung A.17 – *Klassen zur Darstellung von Methoden in Description*

`ServiceConsumerProxyGenerator`. Die abstrakten Generatorklassen sind eine Hilfe für den Entwickler, der einen Codegenerator für die jeweilige Plattform implementiert, damit er weiß, welche Klassen zu implementieren sind. Das Description-Paket enthält die Klasse Dokument, die von den abstrakten Generatorklassen eingebunden wird. Abbildung A.18 stellt den oben beschriebenen Aufbau der Codegeneratoren mit dem Interface Generator und den abstrakten Klassen sowie deren Abhängigkeit von Document in einem UML-Klassendiagramm dar.

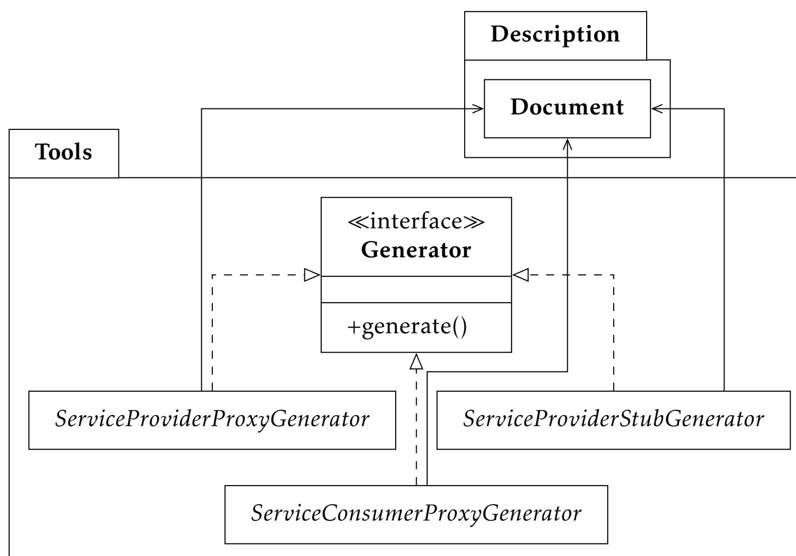


Abbildung A.18 – *Aufbau Codegenerator und Beziehung zu Description*

Im Paket Tools befinden sind auch die konkreten Implementierungen der Codegeneratoren in den Unterpaketen *Java* und *Wiselib*. Die Klassen im Paket Java sind die Klassen für den Java-Codegenerator, diejenigen im Paket Wiselib sind die Klassen für den C++-Codegenerator für CCN-IoT. Jede Klasse im Paket Java beziehungsweise im Paket Wiselib erbt von der gleichnamigen abstrakten Klasse im übergeordneten Paket Generator. Die abstrakten Oberklassen

für die Generatoren enthalten alle wichtigen Variablen, die für die Generator-Implementierungen benötigt werden. Durch die geerbten Oberklassen haben die konkreten Implementierungen der Generatoren Zugriff auf Document im Paket Description. Mit den Informationen der Dienstbeschreibung aus Document generieren die Codegeneratoren den Quellcode. Die Klassen der konkreten Implementierungen in den jeweiligen Paketen und deren Beziehungen zu den abstrakten Generatorklassen sind im UML-Klassendiagramm in Abbildung A.19 zusammengefasst.

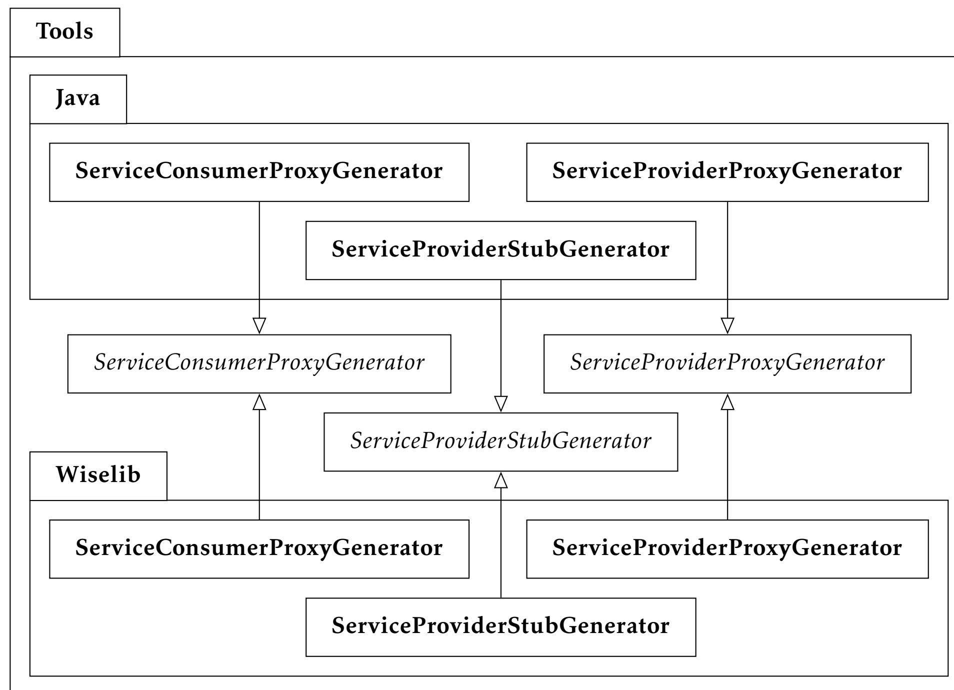


Abbildung A.19 – *Implementierungen der Codegeneratoren für C++ (Wiselib) und Java*

Die Ausgabe der Codegeneratoren sind Quellcodes für Proxies und Stubs (Codegerüste). Quelltext A.7 zeigt das generierte Codegerüst für die Methode `get_temp()` des Dienstanbieters, die in der Dienstbeschreibung in Quelltext A.5 beschrieben wird.

Bei dem Codegerüst implementiert der Entwickler die Logik der Methode (ab Zeile 11). Die Logik der Methode umfasst das Auslesen des Temperatursensors und den Wert in dem `Temperature`-Objekt, das in Zeile 10 definiert wird, in Zeile 12 zurückzugeben.

Der Parameter mit dem Namen `stop_processing` in Zeile 3 ist eine Referenz. Mit dieser Referenz wird die weitere Verarbeitung des Interests gesteuert. Setzt man die Referenz von `stop_processing` auf false, wird der Interest auch anderen Diensten auf dem Knoten zur Verarbeitung angeboten, was standardmäßig nicht der Fall ist. Mit der Referenz auf das `ContentObjectInfo` in Zeile 4 werden Eigenschaften des Content Objects gesetzt, in dem der Rückgabewert transportiert wird. In der vorliegenden Implementierung ist die einzige Eigenschaft des Content Objects seine Gültigkeit, die in CCNx als *Freshness Seconds* bezeichnet

```
1  Temperature
2  get_temp(
3      bool& stop_processing,
4      ContentObjectInfo& coi,
5      const Interest& request,
6      const FixedDataObjectU8& challenge
7  )
8  {
9      // Returns container inner temperature
10     Temperature ret;
11     // TODO: Implement method ...
12     return ret;
13 }
```

Quelltext A.7 – *C++-Codegerüst für die Methode* get_temp *des Dienstanbieters (vgl. Definition in Quelltext A.5)*

wird. Wird die Gültigkeit nicht gesetzt, wird in der vorliegenden Implementierung ein Standardwert angenommen. Als dritter Parameter wird der Interest der Anfrage in Zeile 5 der Methode übergeben. Der Parameter mit dem Namen `challenge` in Zeile 6 aus der Beschreibung ist der letzte Parameter der Methode. Die Challenge dient als Eingabe, um die Response zu berechnen, die in der Antwort mitgeschickt wird. Wird aus der Challenge die Response richtig berechnet, so ist die Antwort authentisch.

In der Praxis werden die ersten drei Parameter vom Entwickler ignoriert. Insbesondere der erste Parameter in Zeile 3 dient in erster Linie zu Debugging-Zwecken. Setzt man `stop_processing` auf false, ist beispielsweise ein Logging-Dienst auf dem Knoten in der Lage, die Interests über die Default-Methode zu empfangen.

Die Methode aus Quelltext A.7 wird von dem dazugehörigen Dienstanbieter-Proxy aufgerufen. Dienstanbieter-Proxy sind für die Verarbeitung von entfernten Aufrufen zuständig. Dabei wird geprüft, ob der empfangene Interest eine Anfrage ist, also eine serialisierten Dienstmethodenaufruf enthält, die von dem Proxy verarbeitet werden kann. Kann der serialisierte Methodenaufruf verarbeitet werden, wird dieser von dem Proxy in einen lokalen Methodenaufruf umgesetzt.

Quelltext A.8 zeigt den Teil für die Verarbeitung von entfernten Aufrufen an `get_temp()` des Dienstanbieter-Proxies. Hierbei handelt es sich um C++-Quellcode für die drahtlosen Sensorknoten. Der Quellcode in Quelltext A.8 wird vollständig generiert; der Entwickler muss hier keinen Code mehr hinzufügen.

In Zeile 4 wird zunächst geprüft, ob der empfangene Interest den Methodennamen `get_temp()`, beziehungsweise deren Hash `0xda34a0c3`, enthält. Die Klassen für Methodennamen werden bei den namenszentrischen Diensten von Namen abgeleitet (vgl. Klassenhierarchie der Buffer-Implementierungen in Abbildung 4.18). Somit werden sie als Felder variabler Länge kodiert (vgl. Kodierung der Typ-Längen-Felder in Tabelle A.1). Das erklärt auch, warum sich vor den

```
1  /*** get_temp */
2  {
3    const uint8_t method[] = {0xc3, 0xda, 0x34, 0xa0, 0xc3};
4    typename Interest::fast_param_iterator mpit = interest.
         get_method( method, sizeof( method ), &has_method );
5
6    if ( has_method )
7    {
8      // challenge
9      FixedDataObjectU8 challenge = FixedDataObjectU8::
           template from_transport_array<FixedDataObjectU8>(
           mpit.get_data(), mpit.get_data_len(), &ok );
10     if ( !ok || !mpit.is_array() )
11       return provider_.param_error( interest, method, sizeof
             ( method ), 0 );
12     mpit.next();
13
14     // Check end
15     if ( !mpit.is_end() )
16       return provider_.param_error( interest, method, sizeof
             ( method ), 1 );
17     bool stop_processing = true;
18
19     ContentObjectInfo co_info;
20     Temperature rv = provider_.get_temp( stop_processing,
           co_info, interest, challenge );
21
22     ContentObject co( interest );
23     ContentObjectIterator co_it = co.get_iterator();
24     co.set_freshness_seconds( co_it, co_info.
           freshness_seconds );
25
26     co.template set_content<int16_t>( co_it, rv.value, 0 );
27     const uint8_t *rva;
28     size_t rva_sz;
29     rva_sz = rv.response.to_transport_array( rva );
30     co.set_content( co_it, rva, rva_sz, 0 );
31
32     base_class::send( co );
33     return stop_processing;
34   }
35 }
```

Quelltext A.8 – *C++-Code des Dienstanbieter-Proxies zur Verarbeitung der Methode get_temp() (vgl. Definition in Quelltext A.5)*

Hash-Bytes in Zeile 4 ein Byte mit dem Wert `0xc3` befindet. In diesem Byte kodieren die drei höherwertigen Bits eine Namenskomponente, die restlichen fünf Bits kodieren die Länge plus eins. Diese Byte leitet, vergleichbar zum Command Marker in CCNx (vgl. CCNx-Kommando in Abbildung 3.5), eine Methode der namenszentrischen Dienste ein.

Enthält der Interest die Methode, erfolgt die weitere Verarbeitung. Der Parameter `challenge`, eine Bytearray, wird in Zeile 8 bis Zeile 12 verarbeitet. Bei der Verarbeitung des Parameters erfolgt eine Typprüfung. Ist die Typprüfung nicht erfolgreich, wird dem Dienstanbieter ein Fehler in Zeile 11 signalisiert. In Zeile 14 erfolgt eine Prüfung auf das Ende der Parameterliste. Da der Entfernte Aufruf der Dienstmethode nur einen Parameter hat, wird dem Dienstanbieter in Zeile 16 ein Fehler mitgeteilt, wenn es noch weitere Parameter gibt. Der Aufruf von `get_temp()` am Dienstanbieter erfolgt in Zeile 20. In Zeile 22 wird das Content Object erstellt, das den Rückgabewert des Methodenaufrufs transportiert. Da es sich bei dem Rückgabewert um einen zusammengesetzten Datentyp handelt, wird jedes Feld des Datentyps einzeln serialisiert. In Zeile 26 wird der Temperaturwert serialisiert, in Zeile 30 wird die Response serialisiert. Ist der Rückgabewert vollständig im Content Object serialisiert, wird das Content Object in Zeile 32 versendet.

Für jede Methode eines Dienstes ist ein Abschnitt wie Quelltext A.8 nötig. Das gezeigte Beispiel ist relativ kurz, da der entfernte Methodenaufruf nur einen Parameter, die Challenge, hat. Für jeden Parameter kommen zwischen vier und fünf Zeilen Code dazu, wobei immer eine Prüfung der Parameter enthalten ist. Auch das Serialisieren des Rückgabewertes braucht eine bis vier Zeilen, abhängig davon, ob es sich um einen primitiven Datentyp oder um ein Array handelt.

Neben den in der Dienstbeschreibung enthaltenen Methoden wird beim Dienstanbieter noch die Default-Methode standardmäßig generiert. Im Codegerüst für den Dienstanbieter ist die Default-Methode eine Methode, die als Parameter nur einen Interest hat. Bleibt das Codegerüst leer, so ist die Default-Methode nicht implementiert. Die Default-Methode wird nicht explizit in der Dienstbeschreibung aufgeführt. Hat ein Dienst nur die Default-Methode, so besteht die Dienstbeschreibung nur aus den Metadaten. Informationen über eine implementierte Default-Methode können in der doc-Eigenschaft des Dienstes angegeben werden. Auf der Dienstnutzerseite gibt eine Callback-Methode, die gewissermaßen das Gegenstück der Default-Methode darstellt. Mit dieser Callback-Methode werden Content Objects empfangen, die von keiner Methode im Dienstnutzer-Proxy verarbeitet werden.

Das obige Beispiel zeigt, dass die Proxy-Implementierung von einfachen Dienstmethoden bereits recht komplex ist. Die Komplexität ist nicht nur auf die Anzahl der Codezeilen zurückzuführen. Auch die Tatsache, dass die Proxies plattformspezifische APIs nutzen, macht die Proxies komplex. Würde der Code für die Proxies manuell erstellt werden, wäre jeder Entwickler angehalten, sich in die jeweilige plattformspezifische API einzuarbeiten.

ABBILDUNGSVERZEICHNIS

1.1 Entwicklungstrends des Internets 4

2.1 Referenzmodelle für Netzwerkprotokolle 12
2.2 Beispiel für ein drahtloses Ad-hoc-Netz 17
2.3 Netzwerkstack für drahtlose Sensorknoten 18
2.4 DODAG Topologie 20
2.5 Dienst und Protokoll im ISO/OSI-Referenzmodell 23
2.6 Anwendungsschicht und Anwender 26
2.7 CCN-Knotenmodell 28
2.8 Aufbau der CCN-Datenstrukturen 29
2.9 Interest-Verarbeitung im Dämon 30
2.10 Content-Object-Verarbeitung im Dämon 31
2.11 Wiederholte Anfrage nach einem Content Object im Netz 32

3.1 Annahmen für ein inhaltszentrisches Internet der Dinge 35
3.2 Aktivitäten bei der Entwicklung von Diensten 40
3.3 SOAP Messaging Pattern 43
3.4 Nachrichtenstrukturen 46
3.5 Beispiel für ein CCNx-Kommando 47
3.6 Präfix und Suffix einer inhaltszentrischen Nachricht 47
3.7 Verarbeitung von kurzlebigen Interests 49
3.8 Abstraktes Konzept von Dienstanbieter und Dienstnutzer 51
3.9 Knotenmodell für namenszentrische Dienste 51
3.10 Verteilte namenszentrische Dienste 52
3.11 Lose Kopplung von namenszentrischen Diensten 53
3.12 Aufrufsyntax einer Dienstmethode 54
3.13 Entfernter Methodenaufruf (knotenzentrisch) 55
3.14 Entfernter Methodenaufruf (namenszentrisch) 56
3.15 Sequenzdiagramm Dienstmethode 57
3.16 Segmentierung von großen Rückgabewerten durch die Proxies . . . 58
3.17 Aufruf von Dienstmethoden am Dienstnutzer 58
3.18 Ablauf zum Aufruf der Default-Methode 59
3.19 Werkzeugunterstützte Codegenerierung 60

4.1 Inhalts-/namenszentrisches Internet 66
4.2 Schnittstellen zwischen Diensten und Protokollen 67
4.3 Dienstmethode als Schnittstelle 68
4.4 Face-Schnittstelle zwischen Dienstinstanz und CCN-Dämon 68
4.5 Schnittstellen zwischen Dienstanbieter, Dienstnutzer und Proxies . 69

4.6 Dienstanbieter und Dienstnutzer in einer Komponente vereint . . . 69
4.7 Beispiel Knotenarchitektur 70
4.8 Wesentliche Klassen von CCN-IoT 72
4.9 Beziehung zwischen Face, Dienstanbieter- und Dienstnutzer-Proxy . 72
4.10 Beziehung zwischen Dienstanbieter, -nutzer und den Proxies 73
4.11 Sequenzdiagramm zur Erzeugung der Softwarekomponenten 75
4.12 Struktur der serialisierten Nachricht aus Quelltext 4.1 77
4.13 Speicherplatzverbrauch Name und Daten getrennt 78
4.14 Speicherplatzverbrauch Name und Daten in einem Array 78
4.15 Aufbau des Buffers 78
4.16 Felder fester Länge 79
4.17 Felder variabler Länge 79
4.18 Klassenhierarchie der Buffer-Implementierungen 81
4.19 Verkürzen eines Buffers 83
4.20 Nachrichtenformat von CCN-IoT 84
4.21 Leistungsbewertung der Vergleichsoperatoren 86
4.22 Interest-Verarbeitung in FIB-Hash 89
4.23 Aufbau von FIB-BF 90
4.24 Entity-Relationship-Diagramm: Präfix und Face 91
4.25 Struktur der FIB von CCN-IoT 91
4.26 Abbildungen von Präfixen zu Faces in der FIB 92
4.27 Größen von FIB mit FIB-BF und FIB-CBF 94
4.28 Kürzeste Präfixlänge für einen identischen Speicherverbrauch . . . 95
4.29 Präfixlänge gegen Größenverhältnis 96

5.1 Anwendung des Schemas im Entwicklungsprozess 102
5.2 Werkzeugunterstützung der namenszentrischen Dienste 110
5.3 Automatisierte Erstellung von Description 111
5.4 Aufbau von Description 113
5.5 Prinzipieller Aufbau eines Werkzeugs zur Codegenerierung 114
5.6 Caption 115
5.7 LLOC-Einsparungen der Dienstbeschreibung 118

6.1 Lebenszyklusmodell für namenszentrische Dienste 124
6.2 Aufteilung des Dienstnamens an einem Beispiel 125
6.3 Beispiel für einen Link-Local Dienstnamen 127
6.4 Name Solicitation Service auf einem Gateway 130
6.5 Nachrichtenaustausch Name Solicitation Service 131
6.6 Technische und organisatorische Bereiche 132
6.7 Naive Zuordnung von Aliasnamen 135
6.8 Komponenten eines Namens als Baumstruktur 136

7.1 Teil der Architektur aus Radio, Simple-Radio und Dämon 140
7.2 Detaillierte Architektur des Simple-Radio Service 141
7.3 Verbreitung von CCNx Nachrichten in drahtlosen Ad-hoc Netzen . 142
7.4 Sequenzdiagramm Backpath-Radio Service 144
7.5 Asymetrischer Link 145
7.6 Beispiel für Mutual Announce and Update 147

7.7 ID Service auf dem Knoten 149
7.8 Gateway Service 151
7.9 Authentifizierung des Dienstanbieters beim Methodenaufruf 152
7.10 Authorisierter Methodenaufruf 153

A.1 Serialisierter Hop-Count 162
A.2 Zugriff auf Felder variabler Länge 163
A.3 Serialisierter Name `/de/fhl/light` 163
A.4 Aufbau eines serialisierten Interests 164
A.5 Aufbau eines serialisierten Content Objects 166
A.6 Aufbau einer serialisierten Methode 167
A.7 Klassenhierarchie der Buffer-Iterator Basisklassen 168
A.8 Schnelle Buffer-Iteratoren 169
A.9 Modifizierende Buffer-Iteratoren 169
A.10 Indexmenge, Elemente und Belegungs-Bitfeld von Set und Vec . . . 171
A.11 Content Store 172
A.12 Pending Interest Table 172
A.13 Forwarding Information Base 173
A.14 FIB-Abbildungscharakteristika im NDN-Testbed 174
A.15 Präfixlängen im NDN-Testbed 176
A.16 Klassen zur Darstellung der Datentypen 183
A.17 Klassen zur Darstellung von Methoden in Description 184
A.18 Aufbau Codegenerator und Beziehung zu Description 184
A.19 Implementierungen der Codegeneratoren 185

TABELLENVERZEICHNIS

2.1 Auswahl von Internetprotokollen nach Schichten 13

3.1 Aspekte knotenzentrischer vs. inhaltszentrischer Ansatz 38

4.1 Laufzeit für das Kopieren von Buffern unterschiedlicher Größe . . . 85
4.2 Ausgewählte Werte aus Abbildung 4.29 96
4.3 Zusammenfassung der Evaluationsergebnisse 96

5.1 Bezeichner und Wertebereiche der Datentypen 108
5.2 Komplexität und Tiefe des generierten Codes 117

6.1 Standardnamen für namenszentrische Dienste 129

A.1 Kodierung der Typ-Längen-Felder 163
A.2 Interest Scope 165
A.3 Typ-Bits bei Integer Content 166

QUELLTEXTVERZEICHNIS

4.1 Strukturdatentyp knotenzentrische Nachricht 77
5.1 JSON-Schema der Dienstbeschreibung 103
5.2 JSON-Schema der zusammengesetzten Datentypen 105
5.3 JSON-Schema der Dienstmethoden 107
5.4 JSON-Schema Abhängigkeiten, Definitionen der Typnamen . . . 108
5.5 Zusammengesetze Datentypen 112
7.1 Beschreibung der Methode `mau()` 146
7.2 Beschreibung von NeighborList, dem Rückgabetyp von `list()` . 148
A.1 Entfernen von Elementen während der Iteration 171
A.2 Vollständiges Schema der Dienstbeschreibung 176
A.3 Beispiel Dienstbeschreibung: Metadaten 178
A.4 Typen Klimaregelung Seecontainer 179
A.5 Temperaturermittlung Klimaregelung Seecontainer 180
A.6 Vollständiges Beispiel der Dienstbeschreibung 180
A.7 Codegerüst `get_temp()` 186
A.8 Verarbeitung `get_temp()` Methode im Dienstanbieter-Proxy . . 187

PUBLIKATIONSVERZEICHNIS

[1] Torsten Teubler, Dennis Pfisterer und Horst Hellbrück. „Memory Efficient Forwarding Information Base for Content-Centric Networking“. In: *International Journal of Computer Networks & Communications (IJCNC)* Bd. 9, Nr. 3 (Mai 2017), S. 67–85. URL: http://aircconline.com/ijcnc/V9N3/9317cnc05.pdf.

[2] Torsten Teubler und Horst Hellbruck. „Name-Centric Services for the Internet of Things“. In: *Networked Systems (NetSys), 2017 International Conference on*. IEEE. 2017, S. 1–2.

[3] Torsten Teubler und Horst Hellbrück. „Architecture and Message Processing for Name-Centric Services in Wireless Sensor Networks“. In: *Advances in Wireless and Optical Communications (RTUWO), 2016*. IEEE. 2016, S. 95–100.

[4] Torsten Teubler und Horst Hellbrück. „Tool Chain for Application Development with Name-Centric Services“. In: *The 12th IEEE International Workshop on Managing Ubiquitous Communications and Services, 2015 (MUCS'15)*. St. Louis, USA, März 2015.

[5] Torsten Teubler, Mohamed Ahmed M. Hail und Horst Hellbrück. „A Solution for the Naming Problem for Name-Centric Services“. In: *12th International Conference on Wired & Wireless Internet Communications (WWIC 2014)*. Paris, France, Mai 2014.

[6] Torsten Teubler, Mohamed Ahmed M. Hail und Horst Hellbrück. „Efficient Data Aggregation with CCNx in Wireless Sensor Networks“. In: *19th EUNICE Workshop on Advances in Communication Networking (EUNICE 2013)*. Chemnitz, Germany, Aug. 2013.

[7] Torsten Teubler, Mohamed A. Hail und Horst Hellbrück. „Transparent Integration of Non-IP WSN into IP Based Networks“. In: Los Alamitos, CA, USA: IEEE Computer Society, 2012, S. 353–358. ISBN: 978-0-7695-4707-7. DOI: 10.1109/DCOSS.2012.10.

[8] Torsten Teubler, Ulrich Walther und Horst Hellbrück. „EZgate - A Flexible Gateway for the Internet of Things“. In: *Proceedings of the Workshops der wissenschaftlichen Konferenz Kommunikation in verteilten Systemen 2011 (WowKiVS 2011)*. Hrsg. von Tiziana Margaria, Julia Padberg und Gabriele Taentzer. Bd. 37. Electronic Communications of the EASST, 2011.

[9] Horst Hellbrück, Torsten Teubler und Stefan Fischer. „Name-Centric Service Architecture for Cyber-Physical Systems (Short Paper)". In: *Service-Oriented Computing and Applications (SOCA), 2013 6th IEEE International Conference on*. 2013.

[10] Daniel Bimschas, Sándor Fekete, Stefan Fischer, Horst Hellbrück, Alexander Kröller, Richard Mietz, Max Pagel, Dennis Pfisterer, Kay Römer und Torsten Teubler. „Real-World G-Lab: Integrating Wireless Sensor Networks with the Future Internet". In: *Testbeds and Research Infrastructures. Development of Networks and Communities*. Springer, 2011, S. 577–579.

[11] Daniel Bimschas, Horst Hellbrück, Richard Mietz, Dennis Pfisterer, Kay Römer und Torsten Teubler. „Middleware for Smart Gateways Connecting Sensornets to the Internet". In: *MidSens'10: The fifth international workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. Bangalore, India, Nov. 2010. URL: http://dl.acm.org/citation.cfm?id=1890787.

[12] Daniel Bimschas, Sándor Fekete, Stefan Fischer, Horst Hellbrück, Alexander Kröller, Richard Mietz, Max Pagel, Dennis Pfisterer, Kay Römer und Torsten Teubler. „Poster Abstract: Real-World G-Lab: Integrating Wireless Sensor Networks with the Future Internet". In: *TridentCom 2010: The 6th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*. Berlin, Germany, Mai 2010. URL: http://link.springer.com/chapter/10.1007/978-3-642-17851-1_46?null.

WEITERE VERÖFFENTLICHUNGEN DES AUTORS

[13] Horst Hellbrück, Torsten Teubler und Gunther Ardelt. „Autonome Unterwasserfahrzeuge und Kommunikationssysteme". In: *MST 2018 - Multisensortechnologie: Low-Cost Sensoren im Verbund*. Bd. 92. Deutsche Hydrographische Gesellschaft, 2018. ISBN: 978-3-95786-170-2. URL: http://geodaesie.info/sr/mst-2018-multisensortechnologie-low-cost-sensoren-im-verbund/7659/1951.

[14] Manfred Constapel, Torsten Teubler und Horst Hellbrück. *A Syntactic Approach to Wreck Pattern Recognition in Sonar Images*. Hrsg. von T. M. Buzug et. al. März 2017.

[15] Torsten Teubler und Horst Hellbrück. „Design of Expert Systems for Autonomous Underwater Vehicle Control". In: *Oceans'16 MTS/IEEE Monterey, CA, USA*. Sep. 2016.

[16] Torsten Teubler, Martin Anschütz und Horst Hellbrück. „Testbed for Development of Networked Autonomous Underwater Vehicles". In: *Oceans'16 MTS/IEEE Shanghai*. Apr. 2016.

[17] Matthieu Sion, Torsten Teubler und Horst Hellbrück. „Embedded Multibeam Sonar Feature Extraction for Online AUV Control". In: *Oceans'16 MTS/IEEE Shanghai*. Apr. 2016.

[18] Torsten Teubler, Liang Shuang und Horst Hellbrück. „Integrating Expert System CLIPS into DUNE for AUV Control“. In: *Oceans’15 MTS/IEEE Genova*. Mai 2015.

[19] Tahir Akram, Tim Esemann, Torsten Teubler und Horst Hellbrück. „A Reusable and Extendable Testbed for Implementation and Evaluation of Cooperative Sensing“. In: *The 8th ACM International Workshop on Performance Monitoring, Measurement and Evaluation of Heterogeneous Wireless and Wired Networks PM2HW2N’13*. Barcelona, Spain, Nov. 2013.

[20] Torsten Teubler und Horst Hellbrück. „Wiseman - A Management and Deployment Approach for WSN Testbed Software“. In: *2013 IEEE INFOCOM Student Poster Session (INFOCOM’2013 Student Posters)*. Turin, Italy, Apr. 2013.

[21] Torsten Teubler, Stefan Krause und Horst Hellbrück. „Verborgenes Rechencluster im neuen Studienarbeitsraum der Elektrotechnik und Informatik“. In: *Impulse* Bd. 16 (Juli 2012), S. 62–63.

[22] Horst Hellbrück, Philipp Krummenauer und Torsten Teubler. „GAAP - Generic Android Application Programming“. In: *Proceedings of WWW/Internet 2011*. Hrsg. von Bebo White, Pedro Isaías und Flávia Maria Santoro. Rio de Janeiro, Brazil, Nov. 2011, S. 581–586. ISBN: 978-989-8533-01-2. URL: `http://www.iadisportal.org/digital-library/gaap-generic-android-application-programming`.

[23] Mohamed A. Hail, Jan Pinkowski, Torsten Teubler, Maick Danckwardt, Dennis Pfisterer und Horst Hellbrück. „RoombaNet - Testbed for Mobile Networks“. In: *Proceedings of the Workshops der wissenschaftlichen Konferenz Kommunikation in verteilten Systemen 2011 (WowKiVS 2011)*. Hrsg. von Tiziana Margaria, Julia Padberg und Gabriele Taentzer. Bd. 37. Electronic Communications of the EASST, 2011.

[24] Torsten Teubler, Jan Pinkowski und Horst Hellbrück. „Cooperative Virtual Memory for Sensor Nodes“. In: *Real-World Wireless Sensor Networks*. Hrsg. von Pedro Marron, Thiemo Voigt, Peter Corke und Luca Mottola. Bd. 6511. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, S. 186–189.

LITERATURVERZEICHNIS

[25] E.C. Ackermann. *C Programming Language Essentials*. EBL-Schweitzer. Research & Education Association, 2015. ISBN: 9780738671956.

[26] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam und Erdal Cayirci. „A Survey on Sensor Networks“. In: *Communications Magazine, IEEE* Bd. 40, Nr. 8 (2002), S. 102–114.

[27] Mohammed M. Alani. „TCP/IP Model“. In: *Guide to OSI and TCP/IP Models*. Cham: Springer International Publishing, 2014, S. 19–50. ISBN: 978-3-319-05152-9. DOI: 10.1007/978-3-319-05152-9_3. URL: http://dx.doi.org/10.1007/978-3-319-05152-9_3.

[28] Aiman J. Albarakati, Junaid Qayyum und Khalid A. Fakeeh. „A Survey on 6LowPAN & its Future Research Challenges“. In: (2014).

[29] Cristina Alcaraz, Pablo Najera, Javier Lopez und Rodrigo Roman. „Wireless Sensor Networks and the Internet of Things: Do We Need a Complete Integration?“ In: *1st International Workshop on the Security of the Internet of Things (SecIoT10)*. 2010.

[30] *ARM buys Sensinode for ‘internet of things’ push*. http://www.computerweekly.com/news/2240204209/ARM-buys-Sensinode-for-internet-of-things-push. Abgerufen am: 25.01.2017.

[31] *ARM mbed*. https://www.mbed.com/en/. Abgerufen am: 25.01.2017.

[32] Kevin Ashton. „That ’Internet of Things’ Thing“. In: *RFiD Journal* Bd. 22, Nr. 7 (2009), S. 97–114.

[33] Luigi Atzori, Antonio Iera und Giacomo Morabito. „The Internet of Things: A Survey“. In: *Computer networks* Bd. 54, Nr. 15 (2010), S. 2787–2805.

[34] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch und Thomas C. Schmidt. „RIOT OS: Towards an OS for the Internet of Things“. In: *Computer Communications Workshops (INFOCOM Workshops), 2013 IEEE Conference on*. IEEE. 2013, S. 79–80.

[35] Emmanuel Baccelli, Christian Mehlis, Oliver Hahm, Thomas C. Schmidt und Matthias Wählisch. „Information Centric Networking in the IoT: Experiments with NDN in the Wild“. In: *arXiv preprint arXiv:1406.6608* (2014).

[36] Naveen Balani und Rajeev Hathi. *Apache CXF Web Service Development: Develop and Deploy SOAP and RESTful Web Services*. Packt Publishing Ltd, 2009.

[37] Tobias Baumgartner, Ioannis Chatzigiannakis, Sándor Fekete, Christos Koninis, Alexander Kröller und Apostolos Pyrgelis. „Wiselib: A Generic Algorithm Library for Heterogeneous Sensor Setworks“. In: *European Conference on Wireless Sensor Networks*. Springer. 2010, S. 162–177.

[38] Michael Bell. „Service-Oriented Modeling Framework“. In: *Service-Oriented Modeling. Wiley, New York* Bd. 366 (2008).

[39] Tom Bellwood, Luc Clément, David Ehnebuske, Andrew Hately, Maryann Hondo, Yin Leng Husband, Karsten Januszewski, Sam Lee, Barbara McKee, Joel Munter u. a. „UDDI Version 3.0“. In: *Published Specification, OASIS* Bd. 5 (2002), S. 16–18.

[40] M. Belshe, R. Peon und M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. RFC Editor, Mai 2015.

[41] Panorios G. Benardos und George Christopher Vosniakos. „Internet of Things and Industrial Applications for Precision Machining“. In: *Solid State Phenomena*. Bd. 261. Trans Tech Publ. 2017, S. 440–447.

[42] Tim Berners-Lee, Roy T. Fielding und Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. RFC Editor, Jan. 2005. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[43] Tim Berners-Lee, Larry Masinter und Mark McCahill. *Uniform Resource Locators (URL)*. RFC 1738. RFC Editor, Dez. 1994. URL: http://www.rfc-editor.org/rfc/rfc1738.txt.

[44] Abhay K. Bhushan und Robert H. Stotz. „Procedures and Standards for Inter-Computer Communications“. In: *Spring Joint Computer Conference, Proceedings of the, April 30–May 2, 1968*. AFIPS ’68 (Spring). Atlantic City, New Jersey: ACM, 1968, S. 95–104. DOI: 10.1145/1468075.1468092. URL: %5Curl%7Bhttp://doi.acm.org/10.1145/1468075.1468092%7D.

[45] Jiang Bian, Kenji Yoshigoe, Amanda Hicks, Jiawei Yuan, Zhe He, Mengjun Xie, Yi Guo, Mattia Prosperi, Ramzi Salloum und François Modave. „Mining Twitter to Assess the Public Perception of the ’Internet of Things’“. In: *PloS one* Bd. 11, Nr. 7 (2016), e0158450.

[46] Philip Bianco, Rick Kotermanski und Paulo F. Merson. „Evaluating a Service-Oriented Architecture“. In: (2007).

[47] Paul V. Biron, Ashok Malhotra, World Wide Web Consortium u. a. *XML Schema Part 2: Datatypes*. 2004.

[48] Burton H. Bloom. „Space/Time Trade-offs in Hash Coding with Allowable Errors“. In: *Communications of the ACM* Bd. 13, Nr. 7 (1970), S. 422–426.

[49] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte und Dave Winer. *Simple Object Access Protocol (SOAP) 1.1*. 2000.

[50] Torsten Braun, Volker Hilt, Markus Hofmann, Ivica Rimac, Moritz Steiner und Matteo Varvello. „Service-Centric Networking“. In: *Communications Workshops (ICC), 2011 IEEE International Conference on*. IEEE. 2011, S. 1–6.

[51] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor, Dez. 2017.

[52] T. Bray, J. Paoli, CM. Sperberg-McQueen, Y. Mailer und F. Yergeau. *Extensible Markup Language (XML) 1.0 5th Edition, W3C Recommendation, November 2008*.

[53] Pedro M. Caldas und Otto Carlos MB. Duarte. „Performance Evaluation of Content Centric Networks“. In: *WNetVirt'2013, Angra dos Reis, RJ, Brazil*. Okt. 2013.

[54] D. Caromel, L. Cardelli und L. Henrio. *A Theory of Distributed Objects: Asynchrony - Mobility - Groups - Components*. Springer Berlin Heidelberg, 2005. ISBN: 9783540272458.

[55] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana u. a. *Web Services Description Language (WSDL) 1.1*. 2001.

[56] Simone Cirani, Luca Davoli, Gianluigi Ferrari, Rémy Léone, Paolo Medagliani, Marco Picone und Luca Veltri. „A Scalable and Self-Configuring Architecture for Service Discovery in the Internet of Things“. In: *IEEE Internet of Things Journal* Bd. 1, Nr. 5 (2014), S. 508–521.

[57] Thomas Clausen, Ulrich Herberg und Matthias Philipp. „A Critical Evaluation of the IPv6 Routing Protocol for Low Power and Lossy Networks (RPL)“. In: *2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE. 2011, S. 365–372.

[58] A. Conta, S. Deering und M. Gupta. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. RFC 4443. RFC Editor, März 2006. URL: http://www.rfc-editor.org/rfc/rfc4443.txt.

[59] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi und Sanjiva Weerawarana. „Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI“. In: *IEEE Internet Computing* Bd. 6, Nr. 2 (2002), S. 86–93.

[60] Li Da Xu, Wu He und Shancang Li. „Internet of Things in Industries: A Survey“. In: *IEEE Transactions on Industrial Informatics* Bd. 10, Nr. 4 (2014), S. 2233–2243.

[61] John D. Day und Hubert Zimmermann. „The OSI Reference Model“. In: *Proceedings of the IEEE* Bd. 71, Nr. 12 (1983), S. 1334–1340.

[62] Stephen E. Deering und Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. http://www.rfc-editor.org/rfc/rfc2460.txt. RFC Editor, Dez. 1998. URL: http://www.rfc-editor.org/rfc/rfc2460.txt.

[63] Whitfield Diffie und Martin Hellman. „New Directions in Cryptography“. In: *IEEE Transactions on Information Theory* Bd. 22, Nr. 6 (1976), S. 644–654.

[64] Olivier Dubuisson. *ASN. 1: Communication Between Heterogeneous Systems*. Morgan Kaufmann, 2001.

[65] R. Dumke. *Software Engineering: Eine Einführung für Informatiker und Ingenieure: Systeme, Erfahrungen, Methoden, Tools*. Vieweg+Teubner Verlag, 2013. ISBN: 9783322969392.

[66] A. Dunkels, B. Gronvall und T. Voigt. „Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors“. In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. Nov. 2004, S. 455–462.

[67] Adam Dunkels. „Contiki: Bringing IP to Sensor Networks“. In: *Ercim News* (2009).

[68] K. Edler, C. Götz, F. Pirchner und D. Obermaier. *MQTT im IoT: Einstieg in die M2M-Kommunikation*. shortcuts. entwickler.Press, 2014. ISBN: 9783868025194.

[69] Engesser, H. and Claus, V. and Schwill, A. *Duden Informatik: ein Sachlexikon für Studium und Praxis*. Duden Series. Dudenverlag, 1993. ISBN: 9783411052325.

[70] David C. Fallside und Priscilla Walmsley. „XML Schema Part 0: Primer Second Edition“. In: *W3C Recommendation* Bd. 16 (2004).

[71] Li Fan, Pei Cao, Jussara Almeida und Andrei Z Broder. „Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol“. In: *IEEE/ACM Transactions on Networking (TON)* Bd. 8, Nr. 3 (2000), S. 281–293.

[72] D. Farinacci, V. Fuller, D. Meyer und D. Lewis. *The Locator/ID Separation Protocol (LISP)*. RFC 6830. http://www.rfc-editor.org/rfc/rfc6830.txt. RFC Editor, Jan. 2013. URL: http://www.rfc-editor.org/rfc/rfc6830.txt.

[73] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach und Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, Juni 1999.

[74] Klaus Finkenzeller und RFID Handbook. „Radio-Frequency Identification Fundamentals and Applications“. In: *Chippenham: John Wiley & Son* (1999).

[75] Francis Galiegue, Kris Zyp u. a. „JSON Schema: Core Definitions and Terminology“. In: *Internet Engineering Task Force (IETF)* (2016).

[76] John Gantz und David Reinsel. „THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East“. In: *IDC iView: IDC Analyze the Future* Bd. 2007, Nr. 2012 (2012), S. 1–16.

[77] Alexander Gluhak, Srdjan Krco, Michele Nati, Dennis Pfisterer, Nathalie Mitton und Tahiry Razafindralambo. „A Survey on Facilities for Experimental Internet of Things Research“. In: *IEEE Communications Magazine* Bd. 49, Nr. 11 (2011).

[78] Groves, Christian and Bormann, Carsten and Jimenez, Jaime. *Constrained RESTful Environments (CoRE) Parameters*. http://www.iana.org/assignments/core-parameters/core-parameters.xhtml. Abgerufen am: 02.02.2017.

[79] Marc J. Hadley. „Web Application Description Danguage (WADL)“. In: *Sun Microsystems Inc., Mountain View, CA* (2006).

[80] J. Hui und P. Thubert. *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*. RFC 6282. http://www.rfc-editor.org/rfc/rfc6282.txt. RFC Editor, Sep. 2011. URL: http://www.rfc-editor.org/rfc/rfc6282.txt.

[81] Chengjia Huo, Ting-Chou Chien und Pai H. Chou. „Middleware for IoT-cloud Integration across Application Domains“. In: *IEEE Design & Test* Bd. 31, Nr. 3 (2014), S. 21–31.

[82] IEEE. „Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)“. In: *IEEE Standard for Local and Metropolitan Area Networks* (2013).

[83] Chalermek Intanagonwiwat, Ramesh Govindan und Deborah Estrin. „Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks“. In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*. ACM. 2000, S. 56–67.

[84] Patricia Jablonski und Daqing Hou. „CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE“. In: *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*. ACM. 2007, S. 16–20.

[85] Van Jacobson, Marc Mosko, D Smetters und Jose Garcia-Luna-Aceves. „Content-Centric Networking“. In: *Whitepaper, Palo Alto Research Center* (2007), S. 2–4.

[86] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs und Rebecca L. Braynard. „Networking Named Content“. In: *Emerging Networking Experiments and Technologies, Proceedings of the 5th International Conference on*. ACM. 2009, S. 1–12.

[87] Wen Jiang und Emmanuel Stefanakis. „What3Words Geocoding Extensions“. In: *Journal of Geovisualization and Spatial Analysis* Bd. 2, Nr. 1 (2018), S. 7.

[88] Nicolai Josuttis. *Das SOA-Manifest: Kontext, Inhalt, Erläuterung*. dpunkt, 2011.

[89] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[90] D. Katz und D. Ward. *Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop)*. RFC 5881. RFC Editor, Juni 2010.

[91] Hamidreza Kermajani und Carles Gomez. „On the Network Convergence Process in RPL over IEEE 802.15. 4 Multihop Networks: Improvement and Trade-Offs“. In: *Sensors* Bd. 14, Nr. 7 (2014), S. 11993–12022.

[92] N. Kushalnagar, G. Montenegro und C. Schumacher. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*. RFC 4919. http://www.rfc-editor.org/rfc/rfc4919.txt. RFC Editor, Aug. 2007. URL: http://www.rfc-editor.org/rfc/rfc4919.txt.

[93] John Larmouth. *ASN. 1 Complete*. Morgan Kaufmann, 2000.

[94] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts und Stephen Wolff. „A Brief History of the Internet“. In: *ACM SIGCOMM Computer Communication Review* Bd. 39, Nr. 5 (2009), S. 22–31.

[95] Sergio Lembo, Jari Kuusisto und Jukka Manner. „In Depth Breakdown of a 6LoWPAN Stack for Sensor Networks“. In: *International Journal of Computer Networks & Communications (IJCNC)* Bd. 2, Nr. 6 (2010).

[96] Christian Lerche, Klaus Hartke und Matthias Kovatsch. „Industry Adoption of the Internet of Things: A Constrained Application Protocol Survey“. In: *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on*. IEEE. 2012, S. 1–6.

[97] Philip Levis, Arsalan Tavakoli und Stephen Dawson-Haggerty. „Overview of Existing Routing Protocols for Low Power and Lossy Networks“. In: *Internet Engineering Task Force, Internet-Draft* (2009).

[98] Zhenmin Li, Shan Lu, Suvda Myagmar und Yuanyuan Zhou. „CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code“. In: *IEEE Transactions on Software Engineering* Bd. 32, Nr. 3 (2006), S. 176–192.

[99] Geraint Luff u. a. „JSON Schema Validation: A Vocabulary for Structural Validation of JSON“. In: *Internet Engineering Task Force (IETF)* (2016).

[100] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, Rebekah Metz und Booz Allen Hamilton. „Reference Model for Service Oriented Architecture 1.0“. In: *OASIS Standard* Bd. 12 (2006).

[101] Somayya Madakam, R. Ramaswamy und Siddharth Tripathi. „Internet of Things (IoT): A Literature Review“. In: *Journal of Computer and Communications* Bd. 3, Nr. 05 (2015), S. 164.

[102] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk und John Anderson. „Wireless Sensor Networks for Habitat Monitoring“. In: *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*. Acm. 2002, S. 88–97.

[103] J. Martocci, P. De Mil, N. Riou und W. Vermeylen. *Building Automation Routing Requirements in Low-Power and Lossy Networks*. RFC 5867. RFC Editor, Juni 2010.

[104] Friedemann Mattern und Christian Flörkemeier. „Vom Internet der Computer zum Internet der Dinge“. In: *Informatik-Spektrum* Bd. 33, Nr. 2 (2010), S. 107–121.

[105] Yannis Mazzer und Bernard Tourancheau. „Comparisons of 6LoWPAN Implementations on Wireless Sensor Networks“. In: *Memory* Bd. 128, Nr. 10k (2009), 512k.

[106] Thomas J. McCabe. „A Complexity Measure“. In: *IEEE Transactions on Software Engineering*, Nr. 4 (1976), S. 308–320.

[107] Steve McConnell. *Code Complete*. Pearson Education, 2004.

[108] Deepankar Medhi. *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann, 2010.

[109] P. Mockapetris. *Domain Names - Concepts and Facilities*. STD 13. http://www.rfc-editor.org/rfc/rfc1034.txt. RFC Editor, Nov. 1987. URL: http://www.rfc-editor.org/rfc/rfc1034.txt.

[110] G. Montenegro, N. Kushalnagar, J. Hui und D. Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. http://www.rfc-editor.org/rfc/rfc4944.txt. RFC Editor, Sep. 2007. URL: http://www.rfc-editor.org/rfc/rfc4944.txt.

[111] Guido Moritz, Frank Golatowski und Dirk Timmermann. „A lightweight SOAP over CoAP Transport Binding for Resource Constraint Networks“. In: *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*. IEEE. 2011, S. 861–866.

[112] Mosco, M. *CCNx 1.0 Protocol Specifications Roadmap*. 2013.

[113] Marc Mosko und Christopher A. Wood. *The CCNx URI Scheme*. Internet-Draft draft-mosko-icnrg-ccnxurischeme-01. Work in Progress. Internet Engineering Task Force, Apr. 2016. 18 S. URL: https://datatracker.ietf.org/doc/html/draft-mosko-icnrg-ccnxurischeme-01.

[114] S. Müller. *Internet of Things (IoT): Ein Wegweiser durch das Internet der Dinge*. Books on Demand, 2016. ISBN: 9783743121584.

[115] Cristina Muñoz, Liang Wang, Eduardo Solana und Jon Crowcroft. „I (FIB) F: Iterated Bloom Filters for Routing in Named Data Networks“. In: *Networked Systems (NetSys), 2017 International Conference on*. IEEE. 2017, S. 1–8.

[116] T. Narten, E. Nordmark, W. Simpson und H. Soliman. *Neighbor Discovery for IP Version 6 (IPv6)*. RFC 4861. http://www.rfc-editor.org/rfc/rfc4861.txt. RFC Editor, Sep. 2007. URL: http://www.rfc-editor.org/rfc/rfc4861.txt.

[117] Peter Naur, John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy und Alan J. Perlis. *Revised Report on the Algorithmic Language Algol 60*. Springer, 1969.

[118] M. Nottingham und E. Hammer-Lahav. *Defining Well-Known Uniform Resource Identifiers (URIs)*. RFC 5785. RFC Editor, Apr. 2010.

[119] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds und Clemente Izurieta. „Comparison of JSON and XML Data Interchange Formats: A Case Study“. In: *Caine* Bd. 2009 (2009), S. 157–162.

[120] D. Obermaier und B. Cabé. *M2M by Eclipse*. shortcuts. entwickler.Press, 2013. ISBN: 9783868024760.

[121] Kazuya Okada, Takeshi Okuda und Suguru Yamaguchi. „Design of Geographically Aggregatable Address and Routing Toward Location Based Multicast“. In: *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*. IEEE. 2013, S. 1307–1312.

[122] Jonas Olsson. „6LoWPAN Demystified“. In: *Texas Instruments* (2014).

[123] A. Osborn. *The MQTT Handbook - Everything You Need to Know about MQTT*. HISTORY INK BOOKS, 2016. ISBN: 9781489135230.

[124] Jianli Pan, Subharthi Paul und Raj Jain. „A Survey of the Research on Future Internet Architectures“. In: *Communications Magazine, IEEE* Bd. 49, Nr. 7 (2011), S. 26–36.

[125] Aishwarya Parasuram, David Culler und Randy Katz. „An Analysis of the RPL Routing Standard for Low Power and Lossy Networks“. In: (2016).

[126] Carlos Pedrinaci und John Domingue. „Web Services are Dead. Long Live Internet Services“. In: *SOA4All White Paper* Bd. 8 (2010).

[127] Charith Perera, Arkady Zaslavsky, Peter Christen und Dimitrios Georgakopoulos. „Sensing as a Service Model for Smart Cities Supported by Internet of Things“. In: *Transactions on Emerging Telecommunications Technologies* Bd. 25, Nr. 1 (2014), S. 81–93.

[128] Phillip James Plauger, Meng Lee, David Musser und Alexander A. Stepanov. *C++ Standard Template Library*. P, 2000.

[129] *PLUS CODES*. https://plus.codes/. Abgerufen am: 02.09.2018.

[130] Gustav Pomberger und Heinz Dobler. *Algorithmen und Datenstrukturen: Eine systematische Einführung in die Programmierung*. P, 2008.

[131] J. Postel. *Internet Control Message Protocol*. STD 5. RFC Editor, Sep. 1981. URL: http://www.rfc-editor.org/rfc/rfc792.txt.

[132] Jon Postel. *Internet Protocol*. STD 5. http://www.rfc-editor.org/rfc/rfc791.txt. RFC Editor, Sep. 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt.

[133] Jon Postel und Joyce Reynolds. „File Transfer Protocol“. In: (1985).

[134] Vidyasagar Potdar, Atif Sharif und Elizabeth Chang. „Wireless Sensor Networks: A Survey“. In: *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on*. IEEE. 2009, S. 636–641.

[135] Zhong Ren, Mohamed A. Hail und Horst Hellbrück. „CCN-WSN - a lightweight, flexible Content-Centric Networking Protocol for Wireless Sensor Networks“. In: *2013 IEEE Eighth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (IEEE ISSNIP 2013)*. Melbourne, Australia, Apr. 2013.

[136] E. Rescorla. *HTTP Over TLS*. RFC 2818. http://www.rfc-editor.org/rfc/rfc2818.txt. RFC Editor, Mai 2000. URL: http://www.rfc-editor.org/rfc/rfc2818.txt.

[137] Jennifer Rexford und Constantine Dovrolis. „Future Internet Architecture: Clean-Slate versus Evolutionary Research“. In: *Communications of the ACM* Bd. 53, Nr. 9 (2010), S. 36–40.

[138] James Roberts. „The Clean-Slate Approach to Future Internet Design: A Survey of Research Initiatives“. In: *Annals of Telecommunications* Bd. 64, Nr. 5 (2009), S. 271–276.

[139] James Rumbaugh, Ivar Jacobson und Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[140] Bilel Saadallah, Abdelkader Lahmadi und Olivier Festor. „CCNx for Contiki: Implementation Details“. Diss. INRIA, 2012.

[141] Rich Salz und Christopher Blunck. *ZSI: The Zolera SOAP Infrastructure*. 2007.

[142] D. Saucez, L. Iannone, O. Bonaventure und D. Farinacci. „Designing a Deployable Internet: The Locator/Identifier Separation Protocol“. In: *IEEE Internet Computing* Bd. 16, Nr. 6 (Nov. 2012), S. 14–21. ISSN: 1089-7801. DOI: 10.1109/MIC.2012.98.

[143] S. Schäffer. *Enterprise Java mit IBM WebSphere: J2EE-Applikationen Effizient Entwickeln*. IBM Software Press. Addison-Wesley, 2002. ISBN: 978-3-827-31898-5.

[144] Manuel Schappacher, Edgar Schmitt, Axel Sikora, Patrick Weber und Artem Yushev. „A Flexible, Modular, Open-Source Implementation of 6LoWPAN“. In: *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2015 IEEE 8th International Conference on*. Bd. 2. IEEE. 2015, S. 838–844.

[145] Scherb, C. and Sifalakis, M. and Tschudin, C. *CCN-lite*. 2013.

[146] D. Schmidt und S.D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Pearson Education, 2002. ISBN: 9780672334283.

[147] Michael Lee Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.

[148] Susmit Shannigrahi, Chengyu Fan und Christos Papadopoulos. „Request Aggregation, Caching, and Forwarding Strategies for Improving Large Climate Data Distribution with NDN: A Case Study“. In: *Information-Centric Networking, Proceedings of the 4th ACM Conference on*. ACM. 2017, S. 54–65.

[149] Shelby, Zach and Bormann, Carsten. *6LoWPAN: The Wireless Embedded Internet*. Bd. 43. John Wiley & Sons, 2011.

[150] Z. Shelby. *Constrained RESTful Environments (CoRE) Link Format*. RFC 6690. RFC Editor, Aug. 2012.

[151] Z Shelby, K Hartke und C Bormann. *The Constrained Application Protocol (CoAP)(RFC 7252)*. 2014.

[152] P.F. Siegert. *Die Geschichte der E-Mail: Erfolg und Krise eines Massenmediums*. Technik - Körper - Gesellschaft. transcript Verlag, 2015. ISBN: 9783839408964.

[153] F. Sprenger und C. Engemann. *Internet der Dinge: Über smarte Objekte, intelligente Umgebungen und die technische Durchdringung der Welt*. Digitale Gesellschaft. transcript Verlag, 2015. ISBN: 9783839430460.

[154] Standard, OASIS. *MQTT version 3.1.1*. 2014.

[155] OASIS Standard. *Devices Profile for Web Services Version 1.1*. 2009.

[156] Stanford-Clark, Andy and Truong, Hong Linh. „MQTT For Sensor Networks (MQTT-SN) Protocol Specification“. In: *International Business Machines (IBM) Corporation Version* Bd. 1 (2013).

[157] ITU Strategy und Policy Unit. „ITU Internet Reports 2005: The Internet of Things“. In: *Geneva: International Telecommunication Union (ITU)* (2005).

[158] J. Strickland. *Verification and Validation for Modeling and Simulation*. LULU Press, 2014. ISBN: 9781312740617.

[159] Sunil Taneja und Ashwani Kush. „A Survey of Routing Protocols in Mobile Ad Hoc Networks“. In: *International Journal of Innovation, Management and Technology* Bd. 1, Nr. 3 (2010), S. 279.

[160] Andrew S. Tanenbaum und David J. Wetherall. *Computer Networks*. Pearson, 2011.

[161] Taube, Christian [Bearb.] *Computer-Lexikon; [10000 Fachbegriffe; Englisch-Wörterbuch]; Microsoft Computer-Lexikon*. Ausg. 2005; [8., aktualisierte Aufl., Sonderausg.] Unterschleißheim: Microsoft Press, 2005. ISBN: 3-86063-076-8.

[162] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn u. a. *XML Schema Part 1: Structures*. 2001.

[163] Thread Group. *Thread Stack Fundamentals*. White Paper 2. Thread Group, Juli 2015.

[164] Robert A. Van Engelen und Kyle A. Gallivan. „The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks“. In: *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*. IEEE. 2002, S. 128–128.

[165] Vasseur, J.P. and Dunkels, A. *Interconnecting Smart Objects with IP: The Next Internet*. Elsevier Science, 2010.

[166] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Sergio Gusmeroli, Harald Sundmaeker, Alessandro Bassi, Ignacio Soler Jubert, Margaretha Mazura, Mark Harrison, Markus Eisenhauer u. a. „Internet of Things Strategic Research Roadmap“. In: *Internet of Things-Global Technological and Societal Trends* Bd. 1 (2011), S. 9–52.

[167] Walther, Ulrich and Fischer, Stefan. „EZnet: A Framework for Rapid Protocol Protyping“. In: *Networks*. World Scientific, 2002, S. 523–534.

[168] Frank J. Wancho. *Digest Message Format*. RFC 1153. RFC Editor, Apr. 1990.

[169] Axel Wegener, Horst Hellbrück, Stefan Fischer, Christiane Schmidt und Sándor Fekete. „AutoCast: An Adaptive Data Dissemination Protocol for Traffic Information Systems“. In: *Vehicular Technology Conference, 2007. VTC-2007 Fall. 2007 IEEE 66th*. IEEE. 2007, S. 1947–1951.

[170] L. Wehmeyer und P. Marwedel. *Eingebettete Systeme*. eXamen.press. Springer Berlin Heidelberg, 2007. ISBN: 9783540340492.

[171] Andreas Weigel, Martin Ringwelski, Volker Turau und Andreas Timm-Giel. „Route-Over Forwarding Techniques in a 6LoWPAN“. In: *Mobile Networks and Management: 5th International Conference, MONAMI 2013, Cork, Ireland, September 23-25, 2013, Revised Selected Papers*. Hrsg. von Dirk Pesch, Andreas Timm-Giel, Ramón Agüero Calvo, Bernd-Ludwig Wenning und Kostas Pentikousis. Cham: Springer International Publishing, 2013, S. 122–135. ISBN: 978-3-319-04277-0. DOI: 10.1007/978-3-319-04277-0_10.

[172] M. Westerlund und S. Cheshire. „IANA Procedure for the Management of the Service Name and Transport Protocol Port Number Registry“. In: *RFC6335* Bd. 201, Nr. 1 (2011).

[173] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur und R. Alexander. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. http://www.rfc-editor.org/rfc/rfc6550.txt. RFC Editor, März 2012.

[174] C. Wood, M. Mosko und E. Uzun. „CCNx Key Exchange Protocol Version 1.0“. In: *Internet Engineering Task Force, Internet-Draft draft-wood-icnrg-ccnxkeyexchange-01* (2016).

[175] Jennifer Yick, Biswanath Mukherjee und Dipak Ghosal. „Wireless Sensor Network Survey“. In: *Computer Networks* Bd. 52, Nr. 12 (2008), S. 2292–2330.

[176] Wei You, Bertrand Mathieu, Patrick Truong, Jean-François Peltier und Gwendal Simon. „DiPIT: A Distributed Bloom-Filter based PIT Table for CCN Nodes“. In: *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*. IEEE. 2012, S. 1–7.

[177] Z-Wave Alliance. *Z-Wave Product's*. 2014.

[178] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D. Thornton, Diana K. Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, Christos Papadopoulos u. a. „Named Data Networking (NDN) Project“. In: *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC* (2010).

[179] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu und Weijun Qin. „IoT Gateway: Bridging Wireless Sensor Networks into Internet of Things“. In: *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*. IEEE. 2010, S. 347–352.

[180] ZigBee Alliance. *ZigBee IP and 920 IP*. 2014. URL: http://www.zigbee.org/zigbee-for-developers/network-specifications/zigbeeip/.

[181] ZigBee Alliance and others. *ZigBee specification*. 2006.

[182] Richard Zurawski. *Industrial Communication Technology Handbook*. CRC Press, 2014.

[183] Kris Zyp u. a. „A JSON Media Type for Describing the Structure and Meaning of JSON Documents“. In: *draft-zyp-json-schema-02 (work in progress)* (2010).