Programmierabstraktionen für eine adaptive nachrichtenbasierte Gruppenkommunikation

Dissertation

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

vorgelegt von
Matthias Prellwitz
geboren am 22. März 1980 in Berlin

Rostock, 28. August 2024

https://doi.org/10.18453/rosdok_id00005140

Gutachter	Prof. Dr.-Ing. habil. Gero Mühl Universität Rostock
	Prof. Dr. rer. nat. Clemens H. Cap Universität Rostock
	Prof. Dr. Stefan Fischer Universität zu Lübeck
Tag der Einreichung	28. August 2024
Tag der Verteidigung	19. November 2024

Zusammenfassung

Das Internet der Dinge (Internet of Things, IoT) bezeichnet ein dynamisches Netzwerk verbundener physischer Objekte, die mit Sensoren und Aktuatoren, Software und weiteren Technologien ausgestattet sind. Diese ermöglichen die Erfassung von Daten der Umgebung sowie die Veränderung von Zuständen. In Verbindung mit Software-Anwendungen entstehen ubiquitäre Umgebungen, welche die Automatisierung, Effizienz oder Assistenz verbessern. Die Anwendungsdomänen erstrecken sich von vernetzten alltäglichen Haushaltsgegenständen bis hin zu hochspezialisierten Industrieanlagen.

Die Komplexität derartiger ubiquitärer Umgebungen stellt Anwendungsentwickler vor Herausforderungen, die zum Zeitpunkt der Anwendungsentwicklung nicht antizipiert werden können: Bei der Implementierung wird seitens der Entwickler der Fokus auf einfache und konsistente Schnittstellen zu den zu bindenden Objekten gelegt. Dynamische Ensembles aus gekoppelten, miteinander kooperierenden Entitäten weisen jedoch veränderliche Konfigurationen auf, was die Entwicklung von Software-Anwendungen vor Herausforderungen stellt. Die Ausprogrammierung potenzieller Bindungs- und Fehlerbehandlungsroutinen erweist sich als anspruchsvolle Herausforderung. Die Vielzahl an Herstellern resultiert in Geräten mit unterschiedlichen Fähigkeiten sowie Schnittstellen, die über verschiedene Protokolle mit individuellen Kommunikationszyklen kommunizieren.

Die vorliegende Arbeit befasst sich mit der Unterstützung für Anwendungsentwickler bei der Umsetzung verteilter Anwendungen. Hierfür werden Programmierabstraktionen vorgestellt, welche die Lücke zwischen Entwurfs- und Laufzeit schließen. Dies ist erforderlich, da die Art und die Anzahl der Geräte, die zur Laufzeit zur Verfügung stehen, bei der Implementierung unbekannt sein können. Die Programmierabstraktion *Dynamische Mengen* erlaubt die Verwendung eine Menge passender Geräte wie ein einzelnes. Diesbezüglich besteht für Anwendungsentwickler die Möglichkeit, zur Entwurfszeit imperativ gegen eine Schnittstelle zu programmieren und deren Methoden aufzurufen, wobei eine einzelne Instanz repräsentiert wird. Die erwähnte Schnittstelle wird mit Metadaten versehen, welche Empfehlungen bezüglich der Ausführung und Aggregation sowie Strategien der Dienstgüte angeben können. Eine Middleware erstellt daraus Datenstrukturen, verwaltet die selektierten Mengenmitglieder, repliziert die Metho-

denaufrufe, behandelt bei der Aggregation auftretende Fehler und berücksichtigt konfigurierte Ausführungsrichtlinien.

Die Schaffung von Transparenz durch einen lokalen Stellvertreter, welcher die Interaktion mit mehreren entfernten Geräten ermöglicht, erlaubt dessen hochfrequenten Aufruf durch die Anwendungslogik. Die Replikation sowie die resultierenden Transportkosten bleiben jedoch verborgen, so dass eine adaptive Kommunikation erforderlich ist, um eine Reduzierung der Netzwerklast zu erreichen. Für die Interaktion über Zustände von Geräten wird zwischen Anfragen von Anwendungen (Pull) und Aktualisierungen von Geräten (Push) unterschieden. Dabei werden sowohl einzelne Adressierungen (Unicast) als auch Gruppenadressierungen (Multicast) berücksichtigt. Da die vollständige Topologie eines Netzwerkes nicht bekannt ist bzw. eine zeitnahe Berechnung der entstehenden Realkosten zu aufwendig wäre, stellt die vorliegende Arbeit verschiedene Heuristiken mit unterschiedlichen Wirkungsweisen vor. Unter Berücksichtigung von Anfrageraten durch Anwendungskomponenten sowie Aktualisierungsraten von Geräteinstanzen erfolgt eine Ermittlung der kostengünstigeren Kommunikationsart unter Zuhilfenahme von Kostenfunktionen. Die Heuristiken wechseln adaptiv zwischen Request/Reply und Publish/Subscribe für einzelne Verbindungen oder eine Gruppe von Empfängern.

Im Rahmen der Interaktion zwischen Geräten und Anwendungen in verteilten Systemen wird eine plattform- und protokollunabhängige Abstraktion zur Bindung vorgestellt, welche Pull- und Push-basierte Interaktionen sowie den adaptiven Wechsel zur Optimierung der Kommunikation ermöglicht. Die Interaktionsmuster werden auf ein nachrichtenbasiertes Adressierungsschema abgebildet, welches auf themenbasierter Publish/Subscribe-Kommunikation basiert. Die Abbildung wird anhand aktueller Protokoll-Implementierungen mit angepassten Nachrichtenstrukturen sowie einer plattformunabhängigen Serialisierung der Nachrichtennutzlast erläutert.

Abstract

The Internet of Things (IoT) is a dynamic network of connected physical objects equipped with sensors and actuators, software and other technologies to collect data from the environment or change states. In conjunction with software applications, ubiquitous environments are created that improve automation, efficiency or assistance. The application domains range from networked everyday household items to highly specialized industrial systems.

The complexity of such ubiquitous environments presents application developers with challenges that are not known at the design time of application development: When implementing, developers prefer simple and consistent interfaces to binding objects. However, dynamic ensembles of coupled, cooperating entities have changing configurations. Programming possible binding and error handling routines is a difficult problem to solve. The large number of manufacturers results in devices with different capabilities and interfaces that communicate via different protocols with individual communication cycles.

The aim of the work is to support application developers in the implementation of distributed applications. For this purpose, programming abstractions are presented that close the gap between design and runtime, since the type and number of devices available at runtime may be unknown during implementation. The programming abstraction *Dynamic Sets* allows a set of suitable devices to be used as a single one. At design time, application developers can imperatively program against an interface and call its methods, which represents a single instance. This interface is annotated with metadata that contains execution recommendations for selection, aggregation and quality of service strategies. A middleware creates data structures from this and manages the selected set members, replicates the method calls, handles errors that occur during aggregation and takes configured execution guidelines into account.

The establishment of transparency through the utilisation of a local proxy facilitates interaction with multiple remote devices, thereby enabling its high-frequency invocation by the application logic. Nevertheless, it is important to note that replication and the subsequent transportation costs are not readily apparent. Consequently, in order to reduce the network load, adaptive communication is required. For interaction of device states, a distinction is made

between requests from applications (pull) and updates from devices (push). Both individual addressing (unicast) and group addressing (multicast) are taken into account. In the absence of complete knowledge of the network topology or where a prompt calculation of the resulting real costs would be too time-consuming, this thesis presents various heuristics with different modes of action. It is possible to determine the most cost-effective type of communication by taking into account request rates by application components and update rates of device instances with the use of cost functions. The heuristics adapt in a manner that is conducive to the seamless transition between request/reply and publish/subscribe for individual connections or a group of recipients.

For the interaction between devices and applications in distributed systems, a platform and protocol-independent abstraction for binding, pull and push-based interactions and adaptive switching to optimize communication are discussed. The interaction patterns are mapped to a message-controlled addressing scheme based on topic-based publish-subscribe communication. Current implementations show the mapping to protocols with different message structures and the platform-independent serialization of the message payload.

Vorwort

Danksagung

Ohne die Unterstützung von einigen Personen wäre die Arbeit nicht möglich gewesen. Zuerst möchte meinem Betreuer Prof. habil. Dr.-Ing. Gero Mühl für die Unterstützung, Motivation und Förderung während Stipendiums und auch in der Zeit zwischen dem Ende der Finanzierung und der Fertigstellung dieser Arbeit herzlich danken. Ein besonderer Dank gilt ebenso Dr.-Ing. Helge Parzyjelga für die konstruktive Zusammenarbeit und den fachlichen Austausch. Ferner danke ich den Gutachtern für das Interesse und die Zeit die vorliegende Arbeit zu sichten und zu bewerten.

Die vorliegende Forschungsarbeit wurde im Rahmen eines Stipendiums des DFG–Graduiertenkollegs MuSAMA (GRK 1424) begonnen und am Lehrstuhl Architektur für Anwendungssysteme (AVA) am Institut für Informatik durchgeführt. Ich danke für Möglichkeit im GRK MuSAMA am fachlichen Austausch mit Professoren und Stipendiaten zum übergeordneten Forschungsthema teilnehmen zu können.

Mein abschließender und tiefempfundener Dank gilt meinen Eltern für Ihre Liebe und Unterstützung sowie Freunden und Kollegen, die an der Fortführung der Arbeit an mich geglaubt sowie mir Ermahnung, Ermutigung und Motivation die Dissertation auch nach Ende des Stipendiums während beruflicher Tätigkeit ermöglicht haben.

Veröffentlichungen

Teile der Arbeit wurden bereits während ihrer Erstellung vorab veröffentlicht. Dafür bilden die Inhalte aus begutachteten Konferenzbeiträgen und ein Journalartikel in eigener Erstautorenschaft die Grundlage, die – so weit nicht anders vermerkt – in Zusammenarbeit mit meinen Betreuern Prof. Dr.-Ing. habil. Gero Mühl und Dr.-Ing. Helge Parzyjegla entstanden sind. Die Publikationen stehen im fachlichen Zusammenhang, da sie dem Datenfluss eines Methodenaufrufes folgen, beginnend von der Anwendungsintegration der Programmierabstraktion über die Verwaltung der Gruppenkommunikation sowie Adaptivitätssteuerung in der Laufzeitumgebung hin zu nachrichtenbasierter Übertragung in verteilten Systemen.

In der ersten Phase entstand die Motivation und Herausforderungen der Programmierabstraktion, welche mit zwei eigenen Veröffentlichungen in Workshops präsentiert wurde [125, 124]. Kapitel 3 basiert auf den gemeinsamen Arbeiten zum Konzept der Programmierabstraktion dynamischer Mengen [126], deren Schnittstellen zusätzlich mit Prof. Dr.-Ing. Dirk Timmermann [129] veröffentlicht wurde. Kapitel 4 hat als Grundlage die fortgeführte Zusammenarbeit mit der Vorstellung und Evaluierung von Heuristiken zur adaptiven Kommunikation, wie sie von der zuvor vorgestellten Programmierabstraktion verwendet werden können [130]. Nach Begutachtung des Konferenzbeitrages wurde ermöglicht zusätzliche Forschungsergebnisse in einem Journalartikel darzustellen [127]. Das letzte Beitragskapitel 5 basiert auf der gemeinsamen Publikation zur Abbildung der vorab motivierten adaptiven Interaktionsmuster mit adaptivem Paradigmenwechsel auf Publish/Subscribe zur entkoppelten Kommunikation in heterogenen Geräteensembles [128].

Inhalte weiterer gemeinsamer Veröffentlichungen von Themen zu Publish/Subscribe [105, 117] wurden nicht in die Arbeit integriert.

Sprache

In der vorliegenden Arbeit wird aus Gründen der besseren Lesbarkeit auf die Nennung der weiblichen, männlichen und diversitären Form von Personenbezeichnungen verzichtet. Das generische Maskulinum richtet sich an alle Leserinnen und Leser und gilt für alle Geschlechter, sofern dies nicht explizit ausgeschlossen wird.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Motivation und Problemstellung 2
1.2 Herausforderungen 4
1.3 Ziele 7
1.4 Beiträge 8
1.5 Struktur 11

2 Grundlagen **13**

2.1 Ubiquitäre Systeme 14
2.1.1 Interaktionsmodelle 16
2.1.2 Kommunikationsstile 18
2.1.3 Datenübertragung 21
2.1.4 Discovery-Mechanismen 23
2.1.5 Multicast 25
2.2 Publish/Subscribe 26
2.2.1 Einordnung 26
2.2.2 Entkopplungen 28
2.2.3 Akteure und Dienste 29
2.2.4 Routing 30
2.2.5 Protokolle 31
2.2.6 Broker 33
2.3 Selbstorganisierende Systeme 35
2.3.1 Self-X Eigenschaften 36
2.3.2 Dienstgüte 37
2.4 Softwareentwicklung 41
2.4.1 Konfiguration 41
2.4.2 Ablaufsteuerung 43
2.4.3 Reflektion 49
2.4.4 Laufzeitumgebung 52
2.4.5 Stellvertreter 54

3 Dynamische Mengen **57**

3.1 Motivation 58
3.2 Konzept 61
3.2.1 Anwendungsintegration 61

3.2.2 Mitgliederverwaltung . . . 63
3.2.3 Methodenaufruf . . . 66
3.2.4 Dienstgüte . . . 66
3.3 Methodenausführung . . . 69
3.3.1 Phasenmodell . . . 69
3.3.2 Aufrufkontext . . . 74
3.3.3 Nachbessernde Rückgabewerte . . . 75
3.4 Programmierschnittstelle . . . 77
3.4.1 Konfiguration . . . 77
3.4.2 Laufzeitumgebung . . . 79
3.4.3 Deklaration und Instanziierung . . . 80
3.4.4 Aggregation . . . 81
3.4.5 Management . . . 82
3.4.6 Erweiterung der Logik . . . 84
3.4.7 Annotation API . . . 88
3.5 Umsetzung . . . 88
3.5.1 Dynamischer Stellvertreter . . . 90
3.5.2 Aufrufstrategie und -planer . . . 91
3.5.3 Aggregation . . . 91
3.6 Verwandte Arbeiten . . . 92
3.7 Diskussion . . . 94

4 Adaptive Unicast- und Multicast-Kommunikation 95
4.1 Motivation . . . 96
4.2 Interaktionsmuster . . . 97
4.2.1 Unicast Pull . . . 98
4.2.2 Unicast Push . . . 98
4.2.3 Multicast Push . . . 99
4.2.4 Multicast Pull . . . 99
4.3 Middleware-Architektur . . . 100
4.4 Heuristiken . . . 101
4.4.1 Verbindungsbasierte Heuristik . . . 101
4.4.2 Geräte-Heuristik . . . 102
4.4.3 Hybride Heuristik . . . 103
4.4.4 Heuristiken mit Multicast Pull . . . 105
4.4.5 Werteglättung . . . 108
4.5 Umsetzung . . . 109
4.5.1 Aufrufe . . . 109
4.5.2 Statistiken . . . 110
4.5.3 Evaluierung . . . 112
4.5.4 Algorithmen . . . 113
4.6 Evaluation . . . 118
4.6.1 Gleiche Anfrageraten . . . 122
4.6.2 Gespreizte Anfrageraten . . . 125
4.6.3 Kostenfunktionen . . . 127
4.6.4 Konfigurationen . . . 129

4.6.5 Mengengrößen . . . 132
4.6.6 Zeitliche Änderungen der Ereignisraten . . . 133
4.7 Verwandte Arbeiten . . . 135
4.8 Diskussion . . . 136

5 Adaptives Publish/Subscribe 137
5.1 Motivation . . . 138
5.2 Anforderungen . . . 139
5.3 Adressierungsschema . . . 140
5.4 Interaktionsmuster . . . 142
5.4.1 Initialisierung . . . 142
5.4.2 Bindung . . . 144
5.4.3 Datenkommunikation . . . 145
5.4.4 Adaptivität . . . 146
5.4.5 Beendigung . . . 147
5.5 Kommunikationsprotokoll . . . 148
5.5.1 Nachrichtentypen . . . 148
5.5.2 Protokollabbildung . . . 151
5.6 Serialisierung . . . 151
5.7 Architektur . . . 153
5.8 Evaluierung . . . 155
5.9 Verwandte Arbeiten . . . 158
5.10 Diskussion . . . 159

6 Zusammenfassung und Ausblick 161
6.1 Diskussion . . . 162
6.2 Ausblick . . . 165

Literaturverzeichnis 167

Abbildungsverzeichnis

1.1 Beitragskapitel der Arbeit 11

2.1 Kommunikationsstile 19
2.2 Ereignissystem für Publish/Subscribe 29
2.3 Publish/Subscribe Protokollstapel 32
2.4 Aufrufe von Frameworks und Programmbibliotheken 47

3.1 Konzept Programmierabstraktion Dynamische Mengen 61
3.2 Phasenmodell für die Methodenausführung in dynamischen Mengen 70
3.3 Konfigurationsoptionen für dynamischer Mengen 78
3.4 Kaskadierende Konfigurationsgenerierung 79
3.5 Schematische Darstellung der Methodenausführung dynamische Mengen 89
3.6 UML-Klassendiagramm, Stellvertreter für dynamische Mengen . 90
3.7 UML-Klassendiagramm, Aufrufstrategie 91

4.1 Middleware-Architektur 100
4.2 Wirkungsweise der Interaktionsmuster 102
4.3 UML-Klassendiagramm, Zählstatistiken 110
4.4 Sequenzdiagramm, Evaluierung der verbindungsbasierten Heuristik auf Geräteseite 113
4.5 Kostengleichheit Pull/Push für Unicast und Multicast-Muster . . 120
4.6 Evaluierung gleicher Ereignisraten der verbindungsbasierten Heuristik 123
4.7 Evaluierung gleicher Ereignisraten der gerätebasierten und der hybriden Heuristiken 124
4.8 Evaluierung der Heuristiken mit Multicast Pull 125
4.9 Evaluierung gespreizter Anfrageereignisraten der verbindungsbasierten Heuristik 126
4.10 Evaluierung gespreizter Anfrageereignisraten der gerätebasierten und hybriden Heuristiken 127
4.11 Evaluierung verschiedener Multicast-Kostenfunktionen der gerätebasierten und hybriden Heuristiken 128
4.12 Evaluierung unterschiedlicher Schwellwerte zur Umschaltung des Kommunikationsparadigmas mit der gerätebasierten Heuristik . . 129

4.13 Evaluierung unterschiedlicher Glättungsfaktoren zur Umschaltung des Kommunikationsparadigmas mit der gerätebasierten Heuristik 130
4.14 Evaluierung unterschiedlicher Schwellwerte zum Austausch von Statistiken mit der gerätebasierten Heuristik 131
4.15 Evaluierung des Umschaltbereiches bei unterschiedlicher Mengengrößen 132
4.16 Evaluierung des Schwellwertes α im zeitlichen Verlauf 133
4.17 Evaluierung des Schwellwertes T_h im zeitlichen Verlauf 134
4.18 Evaluierung des Schwellwertes T_s im zeitlichen Verlauf 134

5.1 Aktivitätsdiagramm mit Interaktionsmustern abgebildet auf Publish/Subscribe-Themen 143
5.2 Klassendiagramm der Nachrichtentypen 150
5.3 Schematische Darstellung der Transportabstraktion 154
5.4 Klassendiagramm zur Protokollbindung 155
5.5 Anpassung des Push/Pull-Interaktionsschemas für unterschiedliche Anfrageraten. 157

Kapitel 1

Einführung

Inhalt

1.1 Motivation und Problemstellung 2
1.2 Herausforderungen 4
1.3 Ziele 7
1.4 Beiträge 8
1.5 Struktur 11

1.1 Motivation und Problemstellung

Die Entwicklung der Computertechnologie und ihrer Anwendungen lässt einen fundamentalen Wandel in der Beziehung zwischen Mensch und Maschine erkennen. Die folgenden Wellen sind jeweils einen Paradigmenwechsel in der Art und Weise gekennzeichnet, wie Computersysteme konzipiert und genutzt werden: Die anfänglichen Großrechner, die Ära der Arbeitsplatzrechner sowie die heutigen allgegenwärtigen Systeme zeigen die Anwendungen weiterentwickelter Technologien zum Lösen komplexerer Problemstellungen, zur Steigerung der Benutzerfreundlichkeit und zur zunehmenden Verschmelzung von digitaler und physischer Welt. [152, 41, 185]

Mainframe Computing: Die Ära der Großrechner umfasst den Zeitraum von den 1950er bis zu den 1970er Jahren. In dieser Phase der Entwicklung der Informationstechnologie waren Mainframes der Typ verfügbarer Computer. Die Rechenleistung war zentralisiert, wobei ein einzelner leistungsfähiger Computer vielen Nutzern diente. Die Nutzung der monolithischen Ressourcen erfolgte durch mehrere Benutzer gleichzeitig, während Aufträge gesammelt und sequenziell abgearbeitet wurden. Die effiziente Auslastung sollte den hohen Anschaffungs- und Betriebskosten Rechnung tragen, weshalb solche Systeme nur für große Organisationen realisierbar waren.

Desktop Computing: In den 1980er- bis 2000er-Jahren etablierten sich der Personal Computer (PC). Diese Phase kann als ein Paradigmenwechsel hin zur Dezentralisierung bezeichnet werden. Die individualisierte Rechenleistung wurde zur Norm, wobei ein Rechner pro Benutzer üblich wurde. Die Einführung grafischer Benutzeroberflächen führte zu einer Erhöhung der Benutzerfreundlichkeit und Etablierung als Heimcomputer für eine breite Bevölkerungsschicht. Client-Server-Architekturen verteilten die Aufgaben zwischen PCs und Servern. Die Entstehung des Internets läutete den Beginn der globalen Vernetzung von Computersystemen ein. Ab den 1990er Jahren brachten die ersten eingebetteten Systeme die Voraussetzung für ubiquitäre Systeme

Ubiquitous Computing: Die Ära der allgegenwärtigen und mobilen Systeme, welche circa ab den 2000er-Jahren einsetzte, stellt einen weiteren Meilenstein dar. In der aktuellen Phase ist die Computertechnologie in den Alltag integriert (ubiquitär, engl. ubiquitous). Mobile Endgeräte wie Smartphones und Tablets haben sich als primäre Hardware etabliert. Cloud Computing verlagert Rechenleistung und Datenspeicherung in dezentrale Netzwerke. Das Internet der Dinge (engl. Internet of Things, IoT) vernetzt Alltagsgegenstände mit Sensoren und Aktuatoren, deren Miniaturisierung, Energieeffizienz und Konnektivität dazu beiträgt. Die Integration von künstlicher Intelligenz und maschinellem Lernen erlaubt die Entwicklung kontextbezogener und adaptiver Systeme. Der Grad an expliziter Interaktion durch Benutzer reduziert sich, da abgeleitete Assistenz von smarten Umgebungen angeboten wird.

Die vorgenannten „wichtigsten Entwicklungen in der Computertechnik“ [185] oder auch Wellen stehen in direktem Zusammenhang mit der Art und Weise

der Nutzung von Computern. Während in der ersten Phase mehrere Nutzer sich einen Großrechner teilten (n:1), ermöglichte der Personal Computer (PC) die Durchdringung des Massenmarktes und die Individualnutzung eines Geräts durch eine Person (1:1). Die aktuelle Allgegenwärtigkeit führt zu einer Umkehrung des Verhältnisses, sodass jede Person von mehreren Geräten umgeben ist, mit denen sie in Interaktion tritt (1:n) bzw. mehrere Personen Zugriff auf die gleichen Ressourcen haben (m:n).

Die Vielfalt vernetzter, assistenzgebender IT-Systeme manifestiert sich in unterschiedlichen Ausprägungen, die sich durch diverse Schwerpunkte auszeichnen und in einer Vielzahl von Anwendungsbereichen zum Einsatz gelangen:

Ubiquitous Computing verfolgt die Vision der allgegenwärtigen und unsichtbaren Technologieintegration [56, 183]. Diese soll für die Benutzer kaum bemerkbar sein und sich nahtlos in deren Alltag integrieren. Ein Beispiel für eine solche Anwendung sind intelligente Heimlösungen, sogenannte „Smart Homes“ [25]. Hierbei werden verschiedene Faktoren wie Beleuchtung, Temperatur und andere Umgebungsvariablen automatisch auf die Bedürfnisse der Benutzer abgestimmt, ohne dass diese aktiv eingreifen müssen.

Das Internet der Dinge stellt ein konkretes Konzept zur Vernetzung und Kommunikation von Geräten dar, welches 1999 erstmals geprägt wurde. Es wurde ein System beschrieben, in dem Objekte in der physischen Welt durch Sensoren mit dem Internet verbunden werden können [139]. Mögliche Anwendungsgebiete umfassen beispielsweise den Einsatz vernetzter Haushaltsgeräte im privaten Heimbereich oder Smart Cities, bei denen Sensoren die Datengrundlage für Verkehrssteuerung oder die Optimierung des Abfallmanagements im öffentlichen Raum liefern. Das Teilgebiet Industrial Internet of Things (IIoT) ist die Anwendung des IoT im industriellen Umfeld [155]. Die Analyse von Daten vernetzter Maschinen und Anlagen erfolgt häufig in Echtzeit, um Muster zu erkennen und daraus Entscheidungen zu treffen. Höhere Zuverlässigkeit und Sicherheitsmechanismen sind im IIoT notwendig, um sensible Geschäftsdaten und kritische Infrastrukturen zu schützen. Die Anwendungsbereiche erstrecken sich über zahlreiche Sektoren, darunter Produktionslinien der Fertigungsindustrie, die Überwachung von Stromnetzen im Energiesektor sowie die Optimierung von Lieferketten.

Allgegenwärtige und pervasive Systeme sind eher benutzer- und kontextorientiert, während das Internet der Dinge stärker die technische Konnektivität und Datenkommunikation fokussiert.

Die vorliegende Arbeit wurde im Graduiertenkolleg „Multimodal Smart Appliance Ensembles for Mobile Applications (MuSAMA)“ begonnen, welche die allgegenwärtige Prägung von IT-Systemen der vorgenannten aktuellen Phase aufgreift. Die Hypothese des Graduiertenkollegs ist, dass zukünftig in alltäglichen Umgebungen dynamische Ensembles intelligenter, vernetzter Geräte existieren, deren Zusammensetzung oft unvorhersehbar und stark dynamisch ist. Die Mitglieder müssen ohne menschliche Konfiguration spontan zusammenarbeiten, um das Ziel der Unterstützung der Benutzer zu erreichen. Die Forschungsaktivitäten

im Rahmen von MuSAMA fokussieren sich auf die Entwicklung von Modellen und Algorithmen, deren Ziel es ist, die Assistenzleistung von intelligenten Umgebungen zu optimieren, ohne dabei auf globales Wissen zurückzugreifen [174].

Das *Smart Appliance Lab* steht an der Universität Rostock als vernetzte Laborumgebung zur Verfügung [176]. Als Anwendungsdomäne dient hier ein intelligenter Besprechungsraum mit der Möglichkeit visueller Präsentationen. Der Raum ist mit verschiedenen Sensoren ausgestattet, welche Aktivitäten von Nutzern erfassen. Dazu zählen beispielsweise berührungsempfindliche Bodenkacheln, tragbare Lokalisierungs-Tags, die drahtlos mit der Infrastruktur verbunden sind, sowie bildgebende Verfahren. Die Fusion sowie die Verarbeitung der Rohdaten in Anwendungslogiken dienen der Ableitung der Intention von Benutzern und der Anpassung des Raumes für passende Szenarien entsprechend dem aktuellen Kontext. Aktuatoren ermöglichen die Helligkeitsanpassung durch Steuerung der Deckenleuchten und der Veränderung von Fensterjalousien, welche auch zusätzliche Projektionsflächen schaffen, sowie die verteilte Platzierung von Präsentationsinhalten durch Projektoren.

Die fachliche Struktur des Graduiertenkollegs unterteilt sich in funktionale Ebenen, auf denen jeweils die dezentrale Kooperation erforscht wird. Das Forschungsprogramm ist in vier Schwerpunkte unterteilt: i) Kontexterkennung und -analyse, ii) Multimodale Interaktion und Visualisierung, iii) Intentionserkennung und Strategieentwicklung und iv) Datenhaltung, Ressourcen- und Infrastrukturmanagement [175]. Die vorliegende Arbeit ist dem vierten Schwerpunkt zuzuordnen, in welchem Themen für ensembleweite Basisdienste zur Realisierung von Kooperationsstrategien erforscht werden. Im Rahmen dessen wird insbesondere das Unterthema ressourcenadaptive, infrastrukturfreie Kommunikationsmechanismen für dynamische Ensembles behandelt.

1.2 Herausforderungen

Die Entwicklung von Anwendungen für verteilte Systeme ist mit einer Vielzahl von Herausforderungen verbunden, die sowohl aus der inhärenten Natur verteilter Systeme als auch aus den spezifischen Eigenschaften moderner Netzwerktechnologien resultieren. Die mittlerweile acht Irrtümer der verteilten Datenverarbeitung beschreiben häufige, falsche Annahmen, die Entwickler über verteilte Systeme treffen [140]:

1. *Das Netzwerk ist zuverlässig*: Netzwerke können dennoch ausfallen, es kann zu Paketverlusten, Verzögerungen oder Trennungen kommen.

2. *Keine Latenz im Netzwerk*: Vielmehr treten Verzögerungen auf, die durch die Distanz, die Übertragungsmedien und den Netzwerkverkehr bedingt sind.

3. *Die Bandbreite ist unendlich*: Die Bandbreite ist jedoch begrenzt, was zur Überlastung und einer suboptimalen Performance führen.

4. *Das Netzwerk ist sicher*: Zum Schutz vor Angriffen oder Abhörmaßnahmen sind Sicherheitsmaßnahmen wie Verschlüsselung und Authentifizierung von Bedeutung, um die Sicherheit der Daten zu gewährleisten.

5. *Die Topologie ändert sich nicht*: Die Netzwerktopologie ändert sich allerdings beispielsweise durch den Ausfall oder dem Hinzufügen neuer Knoten sowie der Änderung von Pfaden.

6. *Es gibt nur einen Administrator*: Verteilte Systeme sind häufig in mehrere administrative Domänen unterteilt, die unterschiedliche Richtlinien, Sicherheitsanforderungen und Konfigurationsmanagement erfahren.

7. *Die Kosten des Transports sind null*: Es fallen in der Praxis Kosten an, beispielsweise in Form von Latenz, Bandbreite oder Energieverbrauch, die in die Systemarchitektur und das Design einbezogen werden müssen.

8. *Das Netzwerk ist homogen*: Netzwerke weisen auf Anwendungsebene keine Homogenität auf. Dies führt zu Herausforderungen bei der Integration und Wartung beispielsweise proprietärer Protokolle. Für die Gewährleistung der Interoperabilität sind daher standardisierte, offene Technologien von entscheidender Bedeutung.

Die Berücksichtigung von Unsicherheiten in ubiquitären oder auch verteilten Systemen stellt eine wesentliche Herausforderung an die Anwendungsentwicklung dar. In diesem Kontext sind insbesondere Netzwerktechnologien, Nebenläufigkeit, Fehlertoleranz und Sicherheit von Relevanz. Die Bewältigung dieser Herausforderungen erfordert die Entwicklung und den Einsatz spezifischer Architekturen, Protokolle und Algorithmen.

Objektorientierte und imperative Programmierung zählen zu den zwei in der Industrie weit verbreiteten Paradigmen zur Umsetzung von Softwareanwendungen. Einerseits unterstützt das objektorientierte Design Entwickler von Anwendungen dabei, aussagekräftige Geschäftsobjekte zu identifizieren, Datenfelder und Methoden für die (Wieder-) Verwendung, die mit diesen Daten arbeiten, zu kapseln und zu bündeln sowie Details der Objektimplementierung vor anderen Entitäten zu verbergen. Andererseits erlaubt der imperative Programmierstil Entwicklern die erforderlichen Schritte zur Erzielung eines bestimmten Ergebnisses aufzuschreiben. Dies kann beispielsweise die Berechnung des Rückgabewertes eines Methodenaufrufs innerhalb der Implementierung eines Objekts umfassen.

Für den Einsatz in verteilten Anwendungen erweisen sich jedoch sowohl das objektorientierte als auch das imperative Programmierparadigma als unzureichend: Mit zunehmender Komplexität eines Systems wird es schwieriger, wenn nicht unmöglich, alle Konfigurationsaspekte zu berücksichtigen, jeden möglichen

Ausführungsfall zu programmieren und potenzielle Fehler angemessen zu behandeln. Des Weiteren verfügen weder objektorientierte noch imperative Programmiersprachen über native Konzepte und Mittel, die Anwendungen die Konfigurationsfreiheit bieten, sich selbst zu organisieren, ihre Ausführung zu optimieren und gegebenenfalls aus früheren Läufen zu lernen sowie diese Prozesse durch die Angabe von Hinweisen zu steuern, anstatt Befehle zu erteilen.

Um die Entwicklung von verteilten Anwendungen zu vereinfachen, gehen ergänzende Programmierabstraktionen die folgenden Herausforderungen an, die diese Probleme jeweils detaillierter widerspiegeln.

Entwicklungszeit versus Laufzeit. Bei der Programmierung einer Anwendung bevorzugen Entwickler in der Regel einfache und konsistente Systemschnittstellen, welche dennoch aussagekräftig genug sind, um die Aufgabe der Anwendung zu erfüllen.

Zur Entwurfszeit erwarten sie Mittel, um Systementitäten einfach zu adressieren und an Objektreferenzen zu binden, damit sie gegen klare Schnittstellen programmieren können, die in der Anwendungsprogrammierschnittstelle (API) des Systems dokumentiert sind. Kurz gesagt, Entwickler streben danach, sich auf ihre Anwendungslogik zu konzentrieren, anstatt sich mit den Besonderheiten und Abhängigkeiten des Systems und seiner Umgebung zu befassen. Systeme, die aus Ensembles lose gekoppelter kooperierender Einheiten bestehen, sind jedoch voller Geräteeigenschaften und von Natur aus empfindlich gegenüber ihrer Umgebung. In Anbetracht der Vielzahl an Herstellern ist dies beispielsweise nicht überraschend. Geräte weisen unterschiedliche Funktionen und Schnittstellen auf und unterstützen diverse Protokolle und Protokollversionen, die potenziell auf verschiedenen Kommunikationsstilen basieren. Die Auswahl einer ausgewogenen Abstraktionsebene ist von entscheidender Bedeutung. Einerseits soll die Komplexität verborgen bleiben, andererseits sind detailliertere Informationen erforderlich, um ein Kontextbewusstsein zu etablieren.

Makroebene versus Mikroebene. Im Allgemeinen verfügen Entwickler auf Makroebene über eine sehr präzise Vorstellung davon, welche Funktionalitäten ihre Anwendung aufweisen muss (*was*). Ihre Aufgabe besteht darin, die vom System auszuführenden Programmieranweisungen und Algorithmen auf Mikroebene zu definieren, auf welche Weise (*wie*) das Ziel der Anwendung erreicht werden soll. Bei verteilten Anwendungen in dynamischen Umgebungen ist die Diskrepanz zwischen Makro- und Mikroebene jedoch besonders groß.

Die dynamische Umgebung sowie die unterschiedlichen Konfigurationen, welche sich aus der Selbstorganisation und Selbstoptimierung ergeben, führen dazu, dass ein Entwickler nicht einmal sicher sein kann, welche Geräte zur Laufzeit verfügbar sind und welche Funktionen zur Ausführung der Anwendung zur Verfügung stehen. Es besteht die Möglichkeit, dass ein Gerät, dessen Fähigkeit von der Anwendung benötigt wird, zum gegenwärtigen Zeitpunkt nicht verfügbar ist,

jedoch durch eine Reihe kooperierender Geräte ersetzt werden könnte. Alternativ können andere Anwendungen, die die erforderliche Ressource belegen, auf andere Geräte migriert werden.

Um die Lücke zwischen Makro- und Mikroebene zu schließen, sollen Entwickler und System Hand in Hand arbeiten, was durch angemessene Programmierabstraktionen unterstützt wird.

Selbstorganisation. Beim Schreiben von Anwendungen möchten Entwickler integrierte selbstorganisierende und selbstoptimierende Strategien von IT–Systemen nutzen. Verschiedene Aspekte selbstorganisierender Systeme werden zu Self-X Eigenschaften zusammengefasst (vgl. Abschnitt 2.3.1). Hier sind bequeme Möglichkeiten zur Angabe gültiger Konfigurationen sowie Optimierungsziele offene Forschungsfragen. Darüber hinaus möchten Anwendungskomponenten in bestimmten Fällen die Organisations- und Optimierungsprozesse des Systems aktiv beeinflussen, da sie nicht zulassen möchten, dass das System alle Entscheidungen übernimmt, sondern stellen stattdessen Präferenzen und Einschränkungen bereit. Die Funktionsfähigkeit dieses Ansatzes setzt voraus, dass die Anwendungskomponenten Mechanismen anbieten, welche Systemeigenschaften im Hinblick auf Konfigurations- und Anpassungszwecke überprüfen lassen. Ein selbstorganisierendes System kann beispielsweise verschiedene Konfigurationsoptionen testen, um die optimale Lösung zu ermitteln, bevor die Komponente tatsächlich bereitgestellt wird. Zudem ist eine weitere Erforschung des Designs der Komponenten und ihrer Schnittstellen erforderlich, welche für diesen Ansatz erforderlich sind.

1.3 Ziele

Die bisherigen Lösungsansätze für die Anwendungsentwicklung in verteilten Systemen erweisen sich als unzureichend, insbesondere im Hinblick auf die Unterstützung von Programmkonstrukten in der Anwendungsdomäne der ubiquitären Systeme bzw. dem Internet der Dinge. Die fortschreitende Vernetzung sowie der technische Fortschritt resultieren in der Entstehung zunehmend komplexere Systeme, die heterogene und dynamische Geräteensembles formen. Während der Implementierung können Bindungs- und Interaktionsroutinen mit entfernten Kommunikationspartnern, wenn überhaupt, durch einen hohen Programmier- und Konfigurationsaufwand umgesetzt werden. Die Ausprogrammierung von Interaktionssequenzen unter Berücksichtigung protokollspezifischer Besonderheiten erlaubt nur wenig Anpassungsfreiheit zur Laufzeit.

Die vorliegende Arbeit verfolgt das Ziel, Methoden zu entwickeln, die Anwendungsentwickler während der Entwurfs- und Implementierungsphase entlasten und die Komplexität zu reduzieren, die mit der Bindung und Interaktion von Kommunikationspartnern in verteilten Systemen einhergeht. Die Auslagerung generischer, sich wiederholender Routinen ermöglicht es Entwicklern, sich auf

die wesentlichen Aspekte eines Problems zu konzentrieren. Dies erleichtert das Verständnis und die Entwicklung von Systemen, insbesondere in großen oder verteilten Softwareprojekten. Die Schaffung von Abstraktionen wie Funktionen, Klassen oder Module erlaubt es Entwicklern, allgemeine Lösungen für spezifische Probleme zu entwerfen, die in verschiedenen Kontexten wiederverwendet und modular eingesetzt werden können. Dies resultiert in einer Reduktion des Zeitbedarfs sowie des Aufwands bei der Entwicklung weiterer Software. Durch Datenkapselung und die Verwendung von Schnittstellen soll verhindert werden, dass Änderungen an einem Teil des Programms unbeabsichtigte Auswirkungen auf andere Teile haben. Ziel ist die Wartbarkeit und Stabilität von Software über deren Lebenszyklus zu erhöhen.

In diesem Zusammenhang ist eine Betrachtung der Konfigurationsmöglichkeiten und -rollen erforderlich, um festzustellen, wie dem Ausführungskontext Richtlinien zur Verfügung gestellt werden können, um diesbezüglich Entscheidungen zu treffen, welche die aktuelle Systemumgebung berücksichtigen. Die Orchestrierung spezialisierter Anwendungen durch Prozeduren in generischen Laufzeitumgebungen verfolgt das Ziel, die Verantwortlichkeiten aufzuteilen. Diesbezüglich ist die Definition und Anwendung von Schnittstellen erstrebenswert zu Elimination von Abhängigkeiten des eigenen Anwendungscodes an eine bestimmte Ausführungsumgebung sowie zur Reduzierung von Wechselhürden bei Migration einer Anwendung auf andere Laufzeitimplementierung, welche mit gleichen Schnittstellen Dienste anbietet.

Ein weiteres Ziel ist die Reduzierung der Netzlast in verteilten Systemen auf nur notwendige und kostengünstige Interaktionen. Die Gegenüberstellung unterschiedlicher Interaktionsmuster sowie deren gezieltem Einsatz zielt darauf ab, die Ressourcennutzung zu optimieren, überflüssige Übertragungen zu eliminieren und Überlastung von Diensten zu reduzieren. Die Anwendbarkeit einer Adaptivitätssteuerung soll durch die Laufzeitumgebung entschieden werden, ohne dass die Anwendungsentwicklung hierauf Einfluss nehmen muss.

Ferner ist die Protokolloffenheit zum Datenaustausch in verteilten Systemen zu bewahren. Es gilt die Integrationsfähigkeit für neue und weiterentwickelte Transportprotokolle in Bestandssysteme zu ermöglichen. Die Einbindung von Protokollen mit ungleichen Interaktionsmustern, Übertragungsstrukturen und Konfigurationsoptionen soll auf höherer Aufrufebene und anwendungsneutral realisiert werden.

1.4 Beiträge

Die Betrachtung der aktuellen Möglichkeiten bei der Anwendungsentwicklung verteilter Systeme hat gezeigt, dass vorhandene Sprachkonstrukte nicht ausreichen, um gewünschte Interaktionen mit Kommunikationspartner während der Programmentwicklung auszugestalten.

Die Beiträge der Arbeit verbessern die Entwicklung verteilter Anwendungen, insbesondere in ubiquitären Umgebungen mit dynamischen und heterogenen Geräteensembles. Dies wird durch den Einsatz unterschiedlicher Programmierabstraktionen erreicht, welche die Komplexität verbergen. In Bezug auf den Datenfluss, der sich von der Anwendungsintegration bis zum Methodenaufruf bei Kommunikationspartnern erstreckt, bieten die Beiträge die folgenden Transparenzgrade:

(i) mit der Anwendungsintegration wird die Anzahl der zu bindenden Kommunikationsteilnehmer eines Typs verborgen,

(ii) die Wahl des günstigeren Kommunikationsparadigmas wird durch Heuristiken abgeschätzt,

(iii) Interaktionssequenzen werden auf Nachrichtenprotokolle für Publish/Subscribe-Kommunikation abgebildet.

Programmierabstraktion der Dynamische Mengen. Die Gruppierung mehrerer Objekte gleichen Typs lässt sich auf verschiedene Motive zurückführen: Eine bewusste Erhöhung der Anzahl an Objekten führt zu statistisch stabilen Resultaten, während die Selektion spezifischer Objekte Interaktionen nach vordefinierten Kriterien ermöglicht. In ubiquitären Umgebungen mit einer dynamischen Gerätezusammenstellung ist eine explizite Bindung während der Anwendungsentwicklung nicht realisierbar. Hauptbeitrag der vorliegenden Arbeit liegt in der Vorstellung des Konzepts zur Programmierabstraktion *Dynamische Mengen*. Dieses Konzept erlaubt die Bündelung mehrerer entfernter Objekte hinter einem Stellvertreter in einer Anwendung. In Abhängigkeit des Verwendungszwecks kann die Transparenz einer Gruppe, die sich dahinter befindet, hergestellt oder aufgelöst werden. Im Rahmen der Anwendungsintegration erfolgt die Definition von Metadaten, welche die Auswahl der Mitglieder einer Menge sowie das Ausführungsverhalten beeinflussen. Das Konzept umfasst mehrere Bearbeitungsphasen, die mit der Aufrufreplikation beginnen und mit der Aggregation der Rückgabewerte in einer Laufzeitumgebung enden. Dazwischen liegen die Interaktionen im Netzwerk. Eine Fallstudie in der Programmiersprache Java zeigt, mit welchen Möglichkeiten Metadaten deklariert werden können. Diese werden von einer Laufzeitumgebung entweder aus dem Programmcode ausgelesen oder von Konfigurationsdateien hinzugezogen.

Heuristiken für adaptive Kommunikation. Die Höhe der Transportkosten, die in einem Netzwerk durch eine Interaktion verursacht werden, ist von verschiedenen Faktoren abhängig. Dabei ist von Relevanz, ob die Kommunikation anfrage- oder aktualisierungsbasiert erfolgt (Push/Pull) sowie ob eine Einzel- oder Gruppenadressierung stattfindet (Unicast/Multicast). Das adaptive Umschalten zwischen verschiedenen Kommunikationsparadigmen kann zu einer Reduktion der Netzwerklast beitragen und einzelne Dienste vor einer zu starken Beanspruchung

schützen. Im Folgenden werden verschiedene Heuristiken mit unterschiedlichen Wirkungsweisen präsentiert, welche die komplexe und nicht verfügbare Netzwerktopologie abstrahieren. Auf Basis von Interaktionen und spezifischer Kostenfunktionen approximieren diese die jeweiligen Transportkosten.

Simulative Evaluation. Die Auswertung der hergeleiteten Pull- und Pushkosten erfolgt in periodischen Abständen an den Klienten, wobei gegebenenfalls eine Umschaltung vorgenommen wird. Im Rahmen der Evaluierung des vorgestellten Konzepts erfolgt eine Überprüfung der prognostizierten Verbesserungen der Gesamtnachrichtenanzahl mittels diskreter Ereignissimulation. Die Modellierung von Anfrage- und Aktualisierungsraten sowie Konfigurationsparametern dient der Anpassung der Reaktivität und Resilienz gegenüber Frequenzänderungen. Dadurch wird ein Vergleich der adaptiven Messungen untereinander sowie von Push- und Pull-Kommunikation ermöglicht.

Abbildung von Interaktionsmuster auf Publish/Subscribe. Der Einsatz nachrichtenbasierter Protokolle trägt durch mehrere Entkopplungen zwischen Sender und Empfänger zu einer Skalierung von Diensten sowie eine effiziente Verarbeitung in Anwendungen bei. In dem Kontext dynamischer und heterogener Systemumgebungen hat sich das Publish/Subscribe-Muster als vorteilhaft erwiesen. Die zuvor präsentierten Interaktionsmuster werden auf ein themenbasiertes Adressierungsschema übertragen, um eine möglichst umfassende Abdeckung durch Dienste und Protokolle zu erreichen.

Abstraktion von Protokoll und Serialisierung. Im Rahmen der vorliegenden Arbeit wird eine Transportschnittstelle vorgestellt, die das Nachrichtenprotokoll sowie die Serialisierung der Nutzlast abstrahiert. Die Vielzahl an nachrichtenbasierten Transportprotokollen, die sich hinsichtlich ihrer Paradigmen, Aufrufsemantiken und Dienstgüteeinstellungen unterscheiden, bedingt bei der Integration in bestehende Systeme entsprechende Anpassungen. Die vorliegende Arbeit präsentiert eine Abstraktion, welche auf höherer Ebene grundlegende Methodenaufrufe bereitstellt. Die Behandlung protokollspezifischer Aspekte erfolgt unter Berücksichtigung konfigurationsspezifischer Parameter durch die Laufzeitumgebung.

Experimentelle Evaluation. Die Simulationsergebnisse zur adaptiven Umschaltung werden nach der Abbildung auf Publish/Subscribe validiert. Ein Experiment mit Klienten und Nachrichtenvermittlern demonstriert die Funktionsweise der Abbildung von Interaktionsmustern auf ausgewählte nachrichtenbasierte Protokolle und präsentiert die zuvor motivierte Adaptivität durch die Anwendung der Heuristiken.

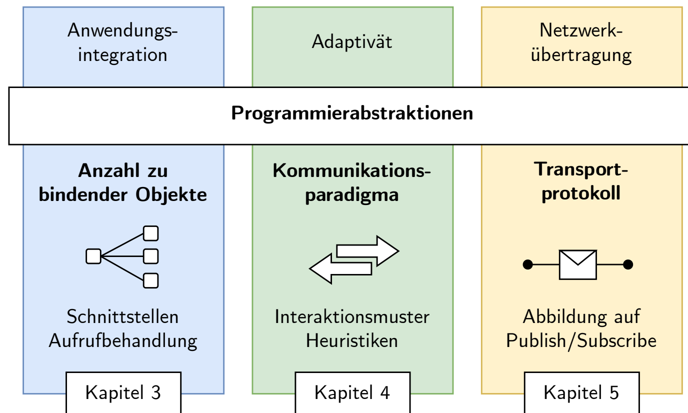


Abbildung 1.1: Beitragskapitel der Arbeit

1.5 Struktur

Der Aufbau der vorliegenden Arbeit ist wie folgt strukturiert: Das anschließende Kapitel 2 führt in die wichtigsten Grundlagen und Hintergründe ein, die in späteren Kapiteln angewendet werden. Kapitel 3 bis 5 beinhalten die Beiträge der Arbeit. Die jeweiligen Ziele und Beiträge sind in Abbildung 1.1 dargestellt. Kapitel 3 präsentiert die Programmierabstraktion der dynamischen Mengen zur Gruppenkommunikation und zeigt Schnittstellen zur Anwendungsintegration sowie Datenflüsse auf. Zur Reduzierung redundanter Netzlast führt Kapitel 4 Heuristiken ein, die ein adaptives Kommunikationsverhalten ermöglichen. In Kapitel 5 werden die Interaktionsmuster herausgearbeitet und auf Publish/Subscribe abgebildet. Ein themenbasiertes Adressierungsschema und Nachrichtendefinitionen setzen die Anforderungen einer adaptiven Kommunikation um. Das Kapitel 6 schließt die Arbeit mit einer Zusammenfassung ab und gibt mit offenen Forschungsfragen einen Ausblick auf mögliche, aufbauende Arbeiten.

Kapitel 2

Grundlagen

Inhalt

2.1 Ubiquitäre Systeme **14**
2.1.1 Interaktionsmodelle 16
2.1.2 Kommunikationsstile 18
2.1.3 Datenübertragung 21
2.1.4 Discovery-Mechanismen 23
2.1.5 Multicast 25
2.2 Publish/Subscribe **26**
2.2.1 Einordnung 26
2.2.2 Entkopplungen 28
2.2.3 Akteure und Dienste 29
2.2.4 Routing 30
2.2.5 Protokolle 31
2.2.6 Broker 33
2.3 Selbstorganisierende Systeme **35**
2.3.1 Self-X Eigenschaften 36
2.3.2 Dienstgüte 37
2.4 Softwareentwicklung **41**
2.4.1 Konfiguration 41
2.4.2 Ablaufsteuerung 43
2.4.3 Reflektion 49
2.4.4 Laufzeitumgebung 52
2.4.5 Stellvertreter 54

2.1 Ubiquitäre Systeme

Ubiquitäre Systeme, engl. Ubiquitous Computing, bezeichnen eine Vision, in der Datenverarbeitungen durch Computer nahtlos in den Alltag integriert und in der Lage sind, auf unauffällige Weise mit der physischen Umgebung und den Benutzern zu interagieren. Dadurch soll ein Mehrwert geschaffen werden. Diese Systeme zielen darauf ab, Computer so allgegenwärtig zu machen, dass sie praktisch überall und jederzeit verfügbar sind, ohne dass sich Benutzer ihrer Anwesenheit bewusst sind [184].

Die Allgegenwärtigkeit erfordert eine Vielzahl von Geräten, die mit lokalen und globalen Diensten verbunden sind, um Benutzern bei ihren Aktivitäten zu unterstützen. Eine Vielzahl von Sensoren nimmt Daten über den Zustand der physischen Umgebung auf, während Aktuatoren auf Empfehlungen des allgegenwärtigen Systems reagieren und entsprechende Aktionen umsetzen. Die Interaktion mit Benutzern soll auf natürliche und unaufdringliche Weise erfolgen, beispielsweise durch Sprachsteuerung, Gesten oder ansprechende Benachrichtigungen.

Ubiquitäre Systeme implizieren eine tiefgreifende Transformation in der Art und Weise, wie wir mit Technologie interagieren. Sie eröffnen das Potenzial für verbesserte Effizienz, Komfort, Sicherheit und Lebensqualität. Gleichwohl sind mit der Realisierung ubiquitärer Systeme Herausforderungen in Bezug auf Datenschutz, Sicherheit, Interoperabilität und Benutzerakzeptanz verbunden, die es sorgfältig zu berücksichtigen gilt. Die allgegenwärtige Datenerfassung und -verarbeitung kann zu Bedenken hinsichtlich der Privatsphäre und der potenziellen Überwachung führen. Die Vorteile ubiquitärer Technologien sollen genutzt werden, ohne dabei grundlegende Rechte und Freiheiten zu gefährden. Des Weiteren muss die Technologie so konzipiert werden, dass sie inklusiv ist und nicht zu neuen Formen der Diskriminierung oder sozialen Spaltung führt.

Für allgegenwärtige Systeme existieren englischsprachige Begriffe, die sich in ihrem Wirken überlappen: Die Forschungsfelder Ubiqutious Computing und Ambient Intelligence fokussieren sich primär auf die Entwicklung von Technologien, die eine nahtlose Integration bestehender Systeme ermöglichen. Dadurch soll das Benutzererlebnis verbessert und die Ausführung von Aufgaben erleichtert werden. Im Gegensatz dazu zielt Persuasive Computing darauf ab, das Verhalten und die Entscheidungsfindung von Benutzern zu beeinflussen oder zu lenken, um spezifische Ziele zu erreichen.

Charakteristika

Durch volatile Ausführungsumgebungen erfolgt eine spontane Kommunikation zwischen neuen Gerätekombinationen, die wenig oder gar keine Vorkenntnisse voneinander haben und gegebenenfalls auf keinen gemeinsamen Dritten zur Vertrauensbildung zurückgreifen können. Die nachfolgenden Paradigmen allgegenwärtige Systeme führen zu Herausforderungen [68, 86, 182].

Dezentralisierung. Während in der Vergangenheit leistungsstarke Monolithen die zentralistische Verarbeitung über einfache Terminals bereitstellten, führte die Einführung des Personal Computers (PC) zur Etablierung der Client-Server-Architektur. Virtualisierungs- und Orchestrierungstechnologien ermöglichen die Umsetzung von Softwarediensten, welche lediglich Kernfunktionalitäten umsetzen und über etablierte Schnittstellen Daten mit anderen Diensten austauschen, die Mehrwerte schaffen. Im Rahmen des Pervasive Computing erfolgt eine Verteilung der Verantwortlichkeiten auf eine Vielzahl kleiner Geräte, welche spezifische Aufgaben übernehmen. Die autonome Natur der Dienste führt zu einer heterogenen Systemumgebung, in der sie ein dynamisches Beziehungsnetzwerk bilden.

Die Dezentralisierung wirft neue Fragestellungen auf, insbesondere im Hinblick auf die Synchronisierung von Anwendungen und Informationen auf mobilen Geräten sowie die Konsistenz von Datenbanken mit unterschiedlichen Fähigkeiten und Speicherkapazitäten. Pervasive Geräte und Anwendungen sind häufig in Service-Infrastrukturen wie Mobilfunknetze eingebettet, was Dienstanbietern die Verwaltung und Aktualisierung von Software auf Kundengeräten ermöglicht. Diese Umgebung erfordert hoch skalierbare und flexible Serversoftware, um mit Millionen von allgegenwärtigen Geräten weltweit umgehen zu können.

Diversifikation. Die eingesetzte Technologie wird in Abhängigkeit des jeweiligen Nutzungszwecks ausgewählt, wobei eine optimale Funktionalität gewährleistet werden soll. Die Anwendung bildet den zentralen Aspekt, während der Inhalt eine untergeordnete Rolle spielt. Anstelle von universellen Geräten, die verschiedene Anforderungen erfüllen sollen, werden spezialisierte tragbare Sensoren, Aktoren und Hardware entwickelt. Diese sind auf die Bedürfnisse bestimmter Benutzergruppen und Anforderungen von Anwendungsfällen optimiert und bieten maßgeschneiderte Funktionen für spezifische Aufgaben. Anwendungen im Kontext des Pervasive Computing sind nahtlose Integrationen von Hard- und Software, die für spezifische Situationen optimiert sind. Ein Verbraucher könnte mehrere Geräte parallel nutzen, je nachdem, welchen Zweck er verfolgt. Die einzelnen Geräte sind so aufeinander abgestimmt, dass sie sich gegenseitig ergänzen und optimale Funktionen für einen bestimmten Anwendungskontext bieten. Anpassbare Schnittstellen und kontextbezogene Dienste unterstützen die Benutzer in einer personalisierten Erfahrung, indem sie reagieren, wo und wie sie verwendet werden. Dies kann beispielsweise der Standort, die Zeit oder die Benutzeraktivität sein. Dadurch können Dienste ihr Verhalten dynamisch anpassen und relevante Informationen und Funktionen basierend auf der aktuellen Situation bereitstellen.

Konnektivität. Die nahtlose Integration und Verbindung von Geräten, Systemen und Diensten erlaubt einen allgegenwärtigen Zugriff auf Informationen und Ressourcen. Die Bildung von Ad-hoc-Netzwerken sowie der Beitritt zu bestehenden Netzwerken erfolgt dynamisch, wobei robuste und flexible Kommunika-

tionskanäle eine wesentliche Voraussetzung dafür darstellen, dass Geräte über verschiedene Umgebungen hinweg miteinander interagieren, Daten austauschen und in Echtzeit zusammenarbeiten. Die Nutzung kontextbezogener Informationen – wie Standort, Nähe und Benutzerpräferenzen – gestattet die Verarbeitung abgeleiteter, aggregierter oder fusionierter Informationen zur Generierung von Diensten, die an bestimmte Situationen angepasst sind.

Einfachheit. Die Gestaltung von Systemen und intuitiver Schnittstellen, welche eine einfache Bedienbarkeit aufweisen und nur einen minimalen Aufwand seitens des Benutzers für die Interaktion erfordern, führt zu einer Reduktion der kognitiven Belastung. Die fortschreitende Entwicklung von Hardware-Designs mit verfügbaren Kommunikationsschnittstellen erlaubt eine Verlagerung der Rechenleistung auf die Endnutzergeräte, wodurch ein Kontextbewusstsein erzeugt werden kann. Die Integration von Technologie in die Umgebungen der Benutzer erfolgt auf unaufdringliche Weise, sodass sie sich nahtlos in ihre täglichen Routinen einfügt, ohne Störungen zu verursachen. Technologie soll auch für Benutzer mit Behinderungen zugänglich sein, indem integrative Schnittstellen entworfen und alternative Interaktionsmodi bereitgestellt werden.

Die Basis für allgegenwärtige Systeme bilden Technologien wie Sensornetzwerke, drahtlose Kommunikation (beispielsweise WLAN und Bluetooth), Cloud-Computing, IoT-Geräte sowie intuitive Benutzerschnittstellen (u.a. Touchscreens und Sprachassistenten). Die Verbindung von Hard- und Software zu einem Netzwerk resultiert in einem Ensemble kooperierender Komponenten, welches die Herausforderungen verteilter Systeme teilt. Die Komponenten können Hardwaregeräte, wie Computer, Server oder Sensoren, sowie Softwarekomponenten, wie Anwendungen, Dienste oder Datenbanken, umfassen. Charakteristisch ist dabei, dass die Komponenten räumlich voneinander getrennt sind und dennoch zusammenarbeiten, um gemeinsame Aufgaben zu erfüllen. Verteilte Systeme ermöglichen die Skalierbarkeit, Flexibilität und Redundanz von Computersystemen sowie die Integration verschiedener Technologien und Plattformen.

2.1.1 Interaktionsmodelle

In verteilten Systemen kommt den Interaktionsmodellen eine zentrale Bedeutung zu, da sie die Grundlage für die Kommunikation und Zusammenarbeit des Systems bilden. Diese Modelle legen die Art und Weise der Kommunikation, des Datenaustausches sowie der gemeinsamen Aufgabenbewältigung der verschiedenen Komponenten fest. Sie umfassen sowohl die Sequenzen der Kommunikation als auch die Mechanismen, die verwendet werden, um Nachrichten zu senden, zu empfangen und zu verarbeiten. Dabei können Interaktionsmodelle in verteilten Systemen vielfältig sein und reichen von einfachen Punkt-zu-Punkt-Kommunikationen bis hin zu komplexen verteilten Architekturen mit vielen Teilnehmern und Interaktionsmustern. Tabelle 2.1 zeigt die Kooperationsmodelle mit Kombination aus unterschiedlichen Initiatoren und Adressierungen

		Initiator	
		Verbraucher	Anbieter
Adressierung	Direkt	*Anfrage/Antwort*	*Rückruf*
	Indirekt	*Anonyme Anfrage/Antwort*	*Ereignis-basiert*

Tabelle 2.1: Taxonomie von Kooperationsmodellen, [104, Fig 2.2]

auf, die nachfolgend skizziert werden. Modelle bei denen ein Verbraucher die Daten beim Anbieter anfragt werden als *Pull* zusammengefasst, da die Daten von der Quelle „gezogen“ werden. Andererseits sind Modelle, bei denen ein Anbieter seine Daten an Verbraucher verteilt unter *Push* vereint, da Daten von der Quelle „gedrückt“ werden.

Anfrage/Antwort. (engl. Request/Reply oder Request/Response) Die vorliegende Anfrage sowie die folgende Antwort sind ein exemplarisches Beispiel für einen klar strukturierten Austausch zwischen einem Sender, der eine Anfrage sendet, und einem Empfänger, der darauf antwortet. Der Sender übermittelt dem Empfänger eine Anfrage, auf die der Empfänger mit einer Antwort reagiert. Der Empfänger nimmt die Anfrage entgegen, verarbeitet sie und retourniert eine Antwortnachricht an den Sender. Die Client/Server-Architektur arbeitet nach diesem Muster. Ein prominentes Beispiel ist das Aufrufen von Webseiten, wo vereinfacht der Webbrowser des Clients (Verbraucher) eine HTTP-Anfrage an den Server (Dienstanbieter) sendet, der mit einer Statusmeldung und im Erfolgsfall mit dem HTML-Code zur Darstellung der angefragten Ressource Webseite antwortet.

Anonyme Anfrage/Antwort. Bei einer anonymen Anfrage/Antwort ist der Sender für den Empfänger entweder nicht bekannt oder von untergeordneter Bedeutung. Die Verarbeitung der Anfragen erfolgt durch den Empfänger in einer für alle Sender ähnlichen Weise, wobei die Antworten zurückgesendet werden, ohne dass dabei eine Identifizierung des spezifischen Senders oder eine Berücksichtigung seiner Identität erfolgt.

Rückruf. Mit dem Rückrufparadigma, engl. Callback, erfolgt die Kommunikation zwischen Sender und Empfänger asynchron. Der Sender übermittelt dabei eine Anfrage an den Empfänger, ohne eine unmittelbare Antwort abzuwarten, und folgt lokal dem eigenen Programmfluss. Im Anschluss an die Verarbeitung der Anfrage durch den Empfänger wird eine Funktion oder Methode aufgerufen, welche dem Sender gehört und die Rückgabe des Ergebnisses (Callback) bewirkt. Dies erlaubt eine effiziente Nutzung von Ressourcen sowie die Unterstützung von nicht-blockierenden Operationen.

Ereignis-basiert. Die asynchrone Kommunikation erfolgt durch das Senden und Empfangen von Ereignissen oder Benachrichtigungen. Ein Ereignis wird durch einen Sender ausgelöst, welches durch einen oder mehrere Empfänger empfangen und verarbeitet werden kann. Diese Art der Kommunikation eignet sich insbesondere für situationsbezogene und reaktive Systeme, bei denen Komponenten auf Änderungen oder Ereignisse reagieren müssen, ohne explizite Anfragen zu senden.

Adaptivität

Die in den Interaktionsmodellen vorgestellten Modi agieren entweder nach dem Anfrage- (Pull) oder Aktualisierungsmuster (Push). Die effiziente Auswahl einer Funktionsweise sowie die Einschätzung ihrer Vor- und Nachteile setzt voraus, dass ausreichend Informationen über die Interaktionen und Genauigkeit der Kostenabschätzungen der betreffenden Klienten im System vorhanden sind. Dazu zählen beispielsweise Angaben zur Häufigkeit von Zustandsänderungen und zur Frequenz des Zugriffs durch Klienten.

Bei einer hohen Frequenz von Werteänderungen, jedoch geringer Relevanz für den Klienten, ist der Einsatz der Pull-Technik zur Vermeidung unnötigen Datenverkehrs empfehlenswert. Demgegenüber ist der Einsatz der Push-Technik zu empfehlen, wenn sich ein Wert selten ändert, jedoch häufig vom Client benötigt wird.

Prognostische Aussagen hinsichtlich der Frequenz von Datenänderungen sowie des Zugriffs der Klienten auf diese sind mit Unsicherheiten behaftet, da sich die relevanten Anforderungen in einem dynamischen Kontext verändern können. Eine adaptive Lösung, welche die Vorteile beider Techniken vereint, stellt die Kombination von Push und Pull dar. In der Literatur werden verschiedene Ansätze zur Kombination von Push- und Pull-basierten Techniken wie Push-and-Pull (PaP) und Push-or-Pull (PoP) diskutiert [34, 16]. Bei der Push-or-Pull-Technik werden entweder die Push- oder Pull-Technik verwendet, je nach Bedarf. Diese Technik erlaubt eine adaptive Auswahl der Übertragungsmethode, die sich an den Anforderungen jedes einzelnen Klienten orientiert. In der Entwicklung einer Middleware findet oft eine angepasste Variante dieser Techniken Anwendung, um die Übertragung von Daten effizient zu gestalten. Bei der Push-and-Pull-Methode werden beide Techniken kombiniert, wobei der Klient Daten abruft und der Server zusätzlich Aktualisierungen über Push sendet. Das Verhältnis von Push und Pull kann mit verschiedenen Parametern eingestellt werden, um eine optimale Datenübertragung zu gewährleisten, wobei nur Push oder Pull oder gemischte Kommunikation zum Einsatz kommt.

2.1.2 Kommunikationsstile

Innerhalb von Systemen, Anwendungen oder Prozessen besteht die Möglichkeit, Daten mittels diverser Kommunikationsmethoden und -mechanismen aus-

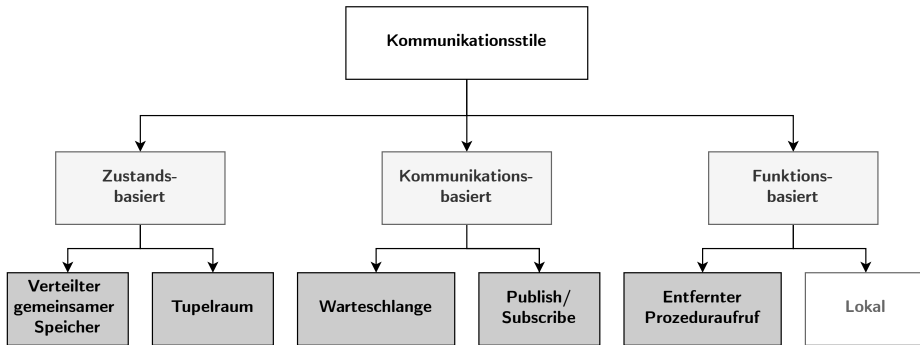


Abbildung 2.1: Kommunikationsstile, angelehnt an [169, Fig. 1.8],[89]

zutauschen. Im Folgenden erfolgt eine Skizzierung einiger der gängigsten Arten der Kommunikation. Abbildung 2.1 zeigt eine Darstellung zur möglichen Klassifizierung von Kommunikationsstilen in verteilten Systemen. Der Auszug zeigt Vertreter der zustandsbasierten Kommunikation durch Zugriff auf geteilte Speicherräume, der die nachrichtenbasierte Kommunikation zur asynchronen Kommunikation und funktionsbasierte Kommunikation zur Annäherung an Aufrufe an lokalen Vertretern.

Funktionsbasierte Kommunikation

Die funktionsbasierte Kommunikation bezeichnet die Interaktion zwischen Programmen oder Prozessen, welche mittels Methoden- oder Funktionsaufrufen über Schnittstellen oder APIs erfolgt. Diese Methode findet häufig Anwendung in der Kommunikation zwischen verschiedenen Modulen oder Komponenten innerhalb eines Softwaresystems.

Sockets. Sockets stellen Kommunikationsendpunkte dar, welche durch das Betriebssystem bereitgestellt werden. Als Transportschnittstelle nehmen sie Daten von Anwendungen in einfachen Blöcken auf und übertragen diese entweder lokal an andere Dienste (Interprozesskommunikation) oder in verteilten Systemen mittels eines Transport- und Vermittlungsprotokolls über TCP/IP an den Empfänger. Die Verantwortung für die Kodierung und Dekodierung der übertragenen Daten liegt bei Socket-Übertragungen auf Sender- und Empfängerseite. Ferner ist die gegebenenfalls notwendige Konvertierung zu behandeln, beispielsweise bei divergierenden Zahlenrepräsentationen oder Compiler-Repräsentationen von Strukturen auf dem Sende- und Zielsystem [146].

Remote Procedure Calls. Entfernte Methodenaufrufe, engl. Remote Procedure Calls (RPC), erlauben einem Programm die Ausführung einer Prozedur oder

Funktion über ein Netzwerk auf einem entfernten System, als würde sie lokal ausgeführt werden. Ein RPC bezeichnet die synchrone Übergabe des Kontrollflusses zwischen zwei Prozessen mit unterschiedlichen Adressräumen auf der Ebene der Programmiersprache [159, 167].

Die Voraussetzung für eine Kommunikation über RPC ist die einheitliche Beschreibung der Schnittstelle des Servers in einer gemeinsamen Notation. In den meisten Fällen erfolgt dies über die sogenannte *Interface Definition Language (IDL)* (vgl. Abschnitt 2.1.3). Ein Compiler generiert für den anfragenden Client und den empfangenden Server die Stubs. Diese dienen der Kapselung derjenigen Funktionalitäten, die zur Aufnahme bzw. Weitergabe an den lokalen Prozess sowie zur Konvertierung der Ein- bzw. Ausgabeparameter in ein Übertragungsformat notwendig sind.

Im Anschluss übergibt ein lokaler Prozess die Aufrufparameter der entfernten Methode stellvertretend an den Client-Stub. Daraufhin wartet der lokale Prozess blockierend auf das Ergebnis. Im Anschluss werden die Aufrufdaten durch den Client-Stub serialisiert und über das Laufzeitsystem vermittelt. In der Folge wird durch das Laufzeitsystem eine Transportverbindung aufgebaut und die Daten an den Server-Stub, teilweise auch Skeleton genannt, auf dem entfernten Server übermittelt. Im Anschluss erfolgt die Deserialisierung der empfangenen Aufrufdaten durch den Server-Stub, welcher die beabsichtigte lokale Prozedur ausführt. Nach der Abarbeitung der Prozedur wird eine erneute Serialisierung vorgenommen und der Transport der Ergebnisdaten in umgekehrter Reihenfolge initiiert. Der vom Client-Stub empfangene und deserialisierte Ergebniswert wird schließlich an den lokalen Prozeduraufruf zurückgegeben, welcher im Programmfluss fortfährt.

Zustandsbasierte Kommunikation

Die zustandsbasierte Kommunikation beschreibt den Datenaustausch zwischen verschiedenen Akteuren über Objekte, welche in gemeinsam zugänglichen Speicherbereichen verwaltet und konsistent gehalten werden.

Verteilter gemeinsamer Speicher. Das Konzept des Distributed Shared Memory (DSM) ermöglicht es Prozessen, die auf verschiedenen Maschinen laufen, auf einen einheitlichen Speicherbereich zuzugreifen, als befänden sie sich auf einem einzigen Computer. Durch die Verteilung des Speichers auf verschiedenen Knoten wird die Bandbreite erhöht und die Reaktionszeit auf lokalen Speicher reduziert. DSM-Systeme gewährleisten die Konsistenz und Kohärenz der Daten über das Netzwerk hinweg, wodurch die Programmierung verteilter Anwendungen vereinfacht wird. Die Entwickler müssen sich nicht mehr mit dem Nachrichtenaustausch befassen, was die Komplexität der Programmierung reduziert [69, 131].

Tupelraum. Ein Tupelraum bzw. Tupelspeicher (engl. Tuple Spaces) stellt eine zentrale Datenstruktur dar, welche die Speicherung von Daten in Tupeln Daten organisiert. Die Anzahl der Felder eines Tupels ist nicht festgelegt und kann beliebige Werte annehmen. In ihnen werden Daten gespeichert, die zwischen verschiedenen Prozessen ausgetauscht werden sollen. Die Tupel können in den Tupelraum eingefügt, abgerufen, gelöscht und modifiziert werden. Dabei erfolgt der Zugriff in der Regel atomar, um Konsistenz und Zuverlässigkeit zu gewährleisten. Tupelräume finden in einer Vielzahl von Bereichen Anwendung, darunter in verteilten Systemen, Peer-to-Peer-Netzwerken, Grid-Computing und Multiagentensystemen.

JavaSpaces ist eine Implementierung von Tupelräumen, die mit in der Programmiersprache Java realisiert wurde. Sie setzt die Speicherung sowie den Austausch von Objekten zwischen verschiedenen Java-Prozessen um, welche sich in einer verteilten Umgebung befinden [55]. Linda stellt eine generische Implementierung von Tupelräumen dar und bietet eine abstrakte Schnittstelle für die Manipulation von Tupeln [60].

Nachrichtenbasierte Kommunikation

Nachrichtenbasierte Kommunikation erfolgt in der Regel asynchron über einen Kanal zwischen verschiedenen Komponenten oder Systemen. Diese Art der Kommunikation findet häufig in verteilten Systemen Anwendung, um lose gekoppelte und skalierbare Architekturen zu unterstützen.

Message Queuing. Die zu übertragenden Nachrichten werden in Warteschlangen gespeichert, bis sie von einem Empfänger abgeholt und verarbeitet werden. Sender und Empfänger müssen nicht gleichzeitig aktiv sein. Zudem besteht die Möglichkeit, Nachrichten in dem Dienst, welcher die Warteschlangen vorhält, zu persistieren, um Datenverlust bei Systemausfällen zu verhindern (Fehlertoleranz). Sind mehrere Abonnenten für eine Warteschlange vorhanden, kann die Last gleichmäßig zwischen ihnen verteilt werden.

Publish/Subscribe. Im Gegensatz zur Warteschlange, werden beim Publish/Subscribe die Nachrichten an alle Abonnenten verteilt, wodurch eine parallele Verarbeitung der gleichen Nachricht durch mehrere Empfänger ermöglicht wird. Eine detaillierte Erläuterung von Publish/Subscribe erfolgt im nachfolgenden Abschnitt 2.2.

2.1.3 Datenübertragung

Die adäquate Formatierung und Strukturierung von Daten in verteilten Systemen ist von entscheidender Bedeutung, um eine effiziente Übertragung, Verarbeitung und Interpretation der Daten zwischen den verschiedenen Komponenten

des Systems zu gewährleisten. Die Organisation der Daten in einer bestimmten Struktur definiert ihre Bedeutung sowie ihre Beziehungen zu anderen Daten. Einfache bzw. primitive Datentypen, wie Ganzzahlen, Gleitkommazahlen oder Zeichenfolgen, werden häufig verwendet, um grundlegende Informationen zu speichern. Die Gruppierung mehrerer Werte in einer Datenstruktur durch zusammengesetzte Datentypen wie Arrays, Listen, Tupel oder Dictionaries erlaubt einen erleichterten Zugriff auf die enthaltenen Daten. Die Verwendung komplexer Datentypen, wie Objekte, Klassen, Strukturen und dergleichen, erlaubt eine strukturierte und organisierte Darstellung vielschichtiger Daten.

Serialisierung bezieht sich auf die Umwandlung von Datenstrukturen in eine Bytefolge, die gespeichert, übertragen und später wieder rekonstruiert werden soll. Der Prozess der Serialisierung ist nicht auf die Kommunikation zwischen verteilten Systemen beschränkt, sondern kann auch für die Speicherung von Datenstrukturen auf Festplatte oder für die Übertragung über ein Netzwerk verwendet werden. Beispiele sind Datenbankpersistenz (BLOB: Binary Large Object), Zwischenspeicherung von Objekten oder von Sitzungsdaten, Zustandserhaltung in verteilten Systemen, Übertragung von Daten über das Netzwerk (bspw. JSON- oder XML).

Marshalling bezeichnet den Prozess der Konvertierung von Datenstrukturen aus dem Speicherbereich eines Programms in ein für die Übertragung über ein Netzwerk oder zwischen verschiedenen Programmen geeignetes Format. Typischerweise findet Marshalling in verteilten Systemen Anwendung, um Objekte oder Daten zwischen verschiedenen Prozessen oder Rechnern auszutauschen. Dazu zählen beispielsweise die Netzwerkübertragung von Daten in einem verteilten System, die Interprozesskommunikation (IPC), der entfernte Methodenaufruf (RPC) sowie die Kommunikation zwischen Webservices. Marshalling beinhaltet die Verwaltung von Referenzen zu Parametern und Schnittstellen, wodurch die Integrität und Struktur des Objekts während der Übertragung gewährleistet wird [31].

Interface Definition Language

Eine Interface Definition Language (IDL) stellt eine formale Sprache dar, welche zur Definition der Schnittstelle zwischen verschiedenen Softwaresystemen verwendet wird. Dabei ist die Schnittstelle unabhängig von der zugrunde liegenden Plattform oder Programmiersprache (Plattformunabhängigkeit). Sie erlaubt die Spezifikation der Kommunikation zwischen verschiedenen Komponenten oder Systemen.

Die Beschreibung der Schnittstellen erfolgt in abstrakter Form, sodass eine Bindung an eine bestimmte Programmiersprache nicht erforderlich ist. Dies erleichtert die Integration von Komponenten, die in verschiedenen Programmiersprachen verfasst sind (Sprachneutralität). IDL erlaubt die explizite Angabe von Datenstrukturen und Typen, die von den Schnittstellen verwendet werden. Dadurch können potenzielle Typfehler vermieden werden (Typsicherheit).

Auf Basis der IDL-Beschreibung erfolgt eine automatische Codegenerierung für verschiedene Programmiersprachen, wodurch die Umsetzung der Schnittstellen vereinfacht wird. IDLs kommen häufig in verteilten Systemen zum Einsatz. Anwendungsfälle sind unter anderem entferne Methodenaufrufe (Remote Procedure Call), Beschreibung von Methoden und Datentypen für Objektanforderungsbroker (ORBs), Endpunkte und Verhaltensweisen von Webdiensten (APIs), Daten(de)serialisierung [187].

Die folgenden Beispiele zeigen Implementierungen auf, die IDL-Funktionen nutzen:

- *CORBA IDL (Common Object Request Broker Architecture Interface Definition Language)* stellt einen Standard für die Entwicklung von verteilten Anwendungen dar. Die CORBA IDL findet Anwendung bei der Definition von Schnittstellen für Objekte in CORBA-Anwendungen [109, 151].
- *SOAP (Simple Object Access Protocol)* ist ein Protokoll zur Kommunikation zwischen verteilten Systemen über das Internet [67]. Eine SOAP IDL, die Web Services Description Language (WSDL) [138], findet Einsatz bei der Definition von Nachrichtenstrukturen und -typen, welche in SOAP-basierten Webdiensten zum Einsatz gelangen. Eine Spezifikation lässt CORBA IDL-Konstrukte auf WSDL abbilden [30].
- *Apache Thrift* stellt ein Framework zur Entwicklung skalierbarer und effizienter verteilter Systeme bereit. Es verwendet eine eigene IDL, um die Schnittstellen zwischen verschiedenen Diensten zu definieren und automatisch Code für verschiedene Programmiersprachen zu generieren [156].
- *Protocol Buffers* wurde von Google entwickelt und sind eine Methode zur Serialisierung strukturierter Daten. Die Definition der Struktur von Nachrichten, welche zwischen Anwendungen ausgetauscht werden, erfolgt mittels einer eigenen IDL [63].
- *ASN.1* stellt eine Art IDL dar, welche sich primär auf die Definition der Struktur und des Formats von Daten konzentriert, die zwischen verschiedenen Systemen transferiert werden. ASN.1 stellt eine formale Syntax zur Definition von Datenstrukturen bereit, die sowohl einfache Datentypen wie Ganzzahlen und Zeichenfolgen als auch komplexe Datentypen wie Sequenzen, Sets und Wahlmöglichkeiten umfasst. Zudem ist ASN.1 ein wesentlicher Bestandteil vieler Kommunikationsprotokolle und Standards, darunter SNMP (Simple Network Management Protocol), LDAP (Lightweight Directory Access Protocol) und X.509-Zertifikate [108].

2.1.4 Discovery-Mechanismen

Diverse Entdeckungsmechanismen (engl. discovery) existieren in der Informatik, welche dazu dienen, Ressourcen, Dienste oder Geräte in einem Netzwerk zu identifizieren und darauf zuzugreifen.

Automatische Konfiguration. *Zero Configuration Networking (Zeroconf)* ist eine Sammlung von Technologien, welche es ermöglichen, dass Geräte und Dienste in einem Netzwerk automatisch und ohne manuelle Konfiguration entdeckt und genutzt werden können. Zu den wesentlichen Komponenten von Zeroconf gehören [162]: Link-Local Addressing lässt Geräten automatisch IP-Adressen in einem lokalen Netzwerk zuweisen, ohne einen DHCP-Server (Dynamic Host Configuration Protocol) zu benötigen. Multicast DNS (mDNS) stellt eine Erweiterung des DNS-Protokoll dar, welche Namensauflösungen in einem lokalen Netzwerk durch Multicast-Broadcasts umsetzt. Dadurch können Geräte ihre Anwesenheit und Dienste ankündigen. DNS Service Discovery (DNS-SD) ergänzt mDNS, indem es standardisierte Mechanismen bereitstellt, welche die Suche und das Finden von Diensten im Netzwerk ermöglichen. Zeroconf wird häufig in kleinen Netzwerken wie Heimnetzwerken und kleinen Büroumgebungen verwendet, da es die Einrichtung und Verwaltung von Netzwerken erheblich vereinfacht. Apples Bonjour[1] Dienst stellt ein bekanntes Beispiel für die Implementierung von ZeroConf dar.

Broadcast. Bei einem Broadcast sendet ein Teilnehmer Datenpakete an alle verfügbaren Punkte in einem Netzwerk, wobei eine explizite Adressierung nicht erfolgt. Ein IP-Broadcast nutzt definierte IP-Adressen, um alle Teilnehmer in einem Netzwerksegment zu erreichen. Auf Protokollebene gibt es definierte Adressierungen, welche alle erreichbaren Teilnehmer erreichen. Das Publish/Subscribe-Kommunikationsmuster bietet je nach verwendetem Protokoll Mechanismen zur Dienstentdeckung, wie bspw. CoAP (Constrained Application Protocol) in IoT-Netzwerken oder MQTT (Message Queuing Telemetry Transport) über zentrale Broker.

Registratur. Als weitere Option steht den Teilnehmern ein Registrierungsdienst zur Verfügung, über den sie sich an- und abmelden. Die Registratur hält die aktuellen verfügbaren Teilnehmer vor und bietet die Referenz zur direkten Interaktion mit dem Anfragenden. In Abhängigkeit von der Implementierung der Registratur sowie der verwendeten Protokolle werden zusätzliche Metadaten zu den Einträgen vorgehalten, wodurch eine Abfrage zur Vorauswahl bereits an der Registratur angeboten wird.

Beispiele für zentralisierte Discovery-Mechanismen sind Directory Services wie LDAP oder UDDI (Universal Description, Discovery, and Integration) oder Service Registries wie *Apache ZooKeeper* [80] für die Verwaltung von Konfigurationsdaten und die Koordination von verteilten Komponenten. Für dezentralisierte Umsetzungen ist zum einen Peer-to-Peer (P2P) Discovery ein Vertreter, welches das Finden und die direkte Interaktion zwischen Teilnehmern in P2P-Netzwerken ohne zentrale Steuerung umsetzt. Eine weitere Möglichkeit stellen

[1] `https://developer.apple.com/bonjour/`

sogenannte Gossip-Protokolle dar. Hierbei teilt jeder Knoten periodisch Informationen über verfügbare Dienste mit zufälligen Nachbarn, was zu einer schnellen Verbreitung der Information im gesamten Netzwerk führt.

2.1.5 Multicast

Multicast ist eine Adressierungsart, welches in Computernetzwerken zu Einsatz kommt, um Daten von einem Sender gleichzeitig an mehrere Empfänger zu übertragen. Im Gegensatz zu Multicast adressiert Unicast nur einen Empfänger. Eine effiziente Datenverteilung an mehrere Empfänger wird erreicht, indem unabhängig von der Anzahl der Empfänger eine einzige Kopie der Daten über das Netzwerk gesendet wird. Dies resultiert in einer Einsparung von Netzwerkbandbreite und reduziert die Belastung sowohl des Absenders als auch der Netzwerkinfrastruktur. Bei der Multicast-Kommunikation wird eine Multicast-Gruppe erstellt, die aus einem einzelnen Sender und mehreren Empfängern besteht. Der Absender übermittelt Datenpakete an die Multicast-Gruppenadresse, welche von Netzwerkroutern repliziert an allen Mitgliedern der Gruppe zugestellt werden.

Die Umsetzung von IP-Multicast erfolgt auf der Vermittlungsschicht des OSI-Modells (Schicht 3) und erfordert die Unterstützung von Netzwerkgeräten wie Router. Für die Adressierung von Multicast-Gruppen werden spezielle Adressen verwendet. Klienten, die am Empfang von Multicast-Verkehr interessiert sind, treten der entsprechenden Multicast-Gruppe bei, indem sie deren Gruppenadresse abonnieren.

Multicast-Routingprotokolle verteilen den Verkehr effizient über Netzwerkrouter. Protokolle wie *Protocol Independent Multicast (PIM)* und *Internet Group Management Protocol (IGMP)* gewährleisten, dass Multicast-Pakete lediglich an Router und Schnittstellen weitergeleitet werden, an denen interessierte Empfänger vorhanden sind.

Das *Application Layer Multicast (ALM)* arbeitet auf Anwendungsebene und ermöglicht mehreren Teilnehmern den Empfang eines einzigen Datenstroms von einer einzigen Quelle, ohne dass dafür explizite Änderungen des Routings oder in der Topologie erforderlich sind. Die Endsysteme übernehmen die Aufgaben des Multicast-Routings, indem sie ein Overlay-Netzwerk bilden, das die Datenverteilung organisiert [189]. Ein Overlay-Netz ist eine logische Abbildung zu einer existierenden (physischen) Netzinfrastruktur, die einen eigenen Adressraum sowie Routingverfahren anwenden kann.

Die Übertragung von Daten über Multicast reduziert den Bedarf an Netzwerkbandbreite und wirkt Netzwerküberlastungen entgegen, indem Daten ausgewählt nur an interessierte Empfänger weitergeleitet werden. Dies steht im Gegensatz zur Unicast-Kommunikation, bei der separate Kopien der Daten an jeden einzelnen Empfänger gesendet werden.

Die Zusagen von zuverlässiger Gruppenkommunikation beinhalten, dass alle Empfänger die Daten vollständig und fehlerfrei erhalten. Die garantierte Zustel-

lung wird durch eine Bestätigung der Empfänger hinsichtlich der (nicht-)erhaltenen Pakete (ACK oder NAK) quittiert. Fehlererkennungs- und -korrekturmechanismen identifizieren verlorene oder beschädigte Datenpakete und korrigieren sie. Die Pakete werden erneut übertragen, wobei eine Sequenzierung sicherstellt, dass die Datenpakete in der korrekten Reihenfolge ankommen [167, 59, 88].

Die Skalierbarkeit von Multicast ist insbesondere bei einer großen Anzahl von Empfängern von Vorteil, da der mit der Datenübertragung verbundene Overhead nicht linear mit der Anzahl der Empfänger zunimmt. In Szenarien, in denen eine Datenverteilung an mehrere Empfänger gleichzeitig erforderlich ist, bietet sich der Einsatz von Multicast an, z.B. Live-Audio- und Video-Streaming, Software-Updates, Echtzeit-Finanzdatenverteilung, Online-Spiele sowie Peer-to-Peer (P2P) und Content-Delivery-Netzwerke (CDN).

2.2 Publish/Subscribe

Das Entwurfsmuster Publish/Subscribe findet in verschiedenen Anwendungsgebieten wie der Softwarearchitektur, Nachrichtenverarbeitungssystemen und verteilten Systemen breite Anwendung. Es erlaubt die asynchrone Kommunikation über Nachrichten zwischen mehreren Sendern und Empfängern. Das Ziel ist die Schaffung einer flexiblen und entkoppelten Kommunikation zwischen Komponenten oder Systemen.

2.2.1 Einordnung

Es existieren diverse Begrifflichkeiten im Umfeld von Publish/Subscribe, die auf ähnlichen Konzepten basieren, jedoch unterschiedliche Einsatzszenarien und Schwerpunkte aufweisen.

Ereignisgesteuerte Architektur. Die ereignisgesteuerte Architektur, engl. Event-Driven Architecture (EDA), stellt ein architektonisches Muster dar, welches auf dem Konzept der asynchronen Ereignisverarbeitung basiert. In einer ereignisgesteuerten Architektur werden Ereignisse produziert, verarbeitet und konsumiert, um den Zustand des Systems zu ändern oder darauf zu reagieren. Publish/Subscribe stellt ein Schlüsselelement von EDA dar und wird häufig verwendet, um Ereignisse zwischen den Komponenten des Systems zu verteilen.

Beobachter-Entwurfsmuster. Das Beobachter-Entwurfsmuster, engl. Observer, ist ein Verhaltensmuster, das zur Implementierung von Abhängigkeiten zwischen Objekten verwendet wird. Eine Veränderung des Zustandes eines Objektes führt zu einer automatischen Benachrichtigung und Aktualisierung aller abhängigen

Objekte [58]. Innerhalb dieses Entwurfsmusters existieren zwei wesentliche Rollen: Zum einen ist das *Subjekt* ist dasjenige Objekt, welches den Zustand modifiziert und die Informationen über diese Änderungen an die beobachtenden Objekte weitergibt. Das Subjekt beinhaltet eine Liste von Beobachtern und bietet Methoden zum Hinzufügen, Entfernen und Benachrichtigen von Beobachtern. Zum anderen werden *beobachtende Objekte* über die Änderungen im Zustand des Subjekts informiert. Dazu registrieren sie sich beim Subjekt, um Benachrichtigungen über Zustandsänderungen zu erhalten. Anschließend implementieren sie eine Update-Methode, die vom Subjekt aufgerufen wird, um die Beobachter über die Aktualisierung zu informieren. Das Entwurfsmuster ermöglicht eine lose Kopplung zwischen Subjekten und Beobachtern, sodass eine unabhängige Existenz beider gewährleistet ist. Es erleichtert die Wartung und Erweiterung des Systems, da neue Beobachter hinzugefügt oder bestehende entfernt werden können, ohne das Subjekt zu verändern. das Muster findet häufig Anwendung in Situationen, in denen Änderungen in einem Objekt den Zustand anderer Objekte beeinflussen können. Beispiele hierfür sind grafische Benutzeroberflächen, Ereignisverarbeitungssysteme oder die Implementierung des Model-View-Controller-Musters.

Programmiersprachen stellen Konstrukte bereit, die das Beobachter-Muster bzw. Publish/Subscribe unterstützen. In der Programmiersprache Java war bis Version 8 der Standard Edition die Klasse `java.util.Observable` für das Subjekt die Schnittstelle `Observer` zur eigenen Implementierung des Beobachter-Musters angeboten. Mit der Einführung der Flow API in Version 9 wurden die zuvor genannten funktionalen Schnittstellen `java.util.concurrent.Flow.Publisher` und `Subscriber` bereitgestellt [112, 91].

Reaktive Programming. Die reaktive Programmierung ist ein Paradigma, das sich auf die Verarbeitung von asynchronen Datenströmen und Ereignissen fokussiert. Im Rahmen der reaktiven Programmierung erfolgt die Behandlung von Änderungen im System durch das Zusammenspiel von Ereignisquellen (Observables) und Ereignisempfängern (Observers) behandelt. Publish/Subscribe wird oft als Mechanismus für die Handhabung von Ereignissen in reaktiven Systemen verwendet. Als vergleichbare Konzepte und Ziele werden Elastizität und Resilienz verfolgt, wobei unterschiedliche Ansätze und Schwerpunkte für die Implementierung von Kommunikationsmechanismen in Softwareanwendungen und Systemen angeboten werden [18, 87].

In der Programmiersprache Java besteht ab Version 8 der Standard Edition die Möglichkeit Implementierungen des asynchronen Aufrufes über die Klasse `java.util.concurrent.Executor` zu realisieren [112]. Des Weiteren stellen Programmbibliotheken zur asynchronen Programmierung wie *ReactiveX* [136] Funktionalitäten zur Beobachtung von Sequenzen bereit und bieten sprachspezifische Erweiterungen der Laufzeitumgebung, für Java über *RxJava (Reactive Extensions for the JVM).*

Warteschlange. Bei diesem Muster erfolgt der Datenaustausch zwischen Sender und Empfänger über Nachrichten. Die Nachrichten werden in die Datenstruktur einer Warteschlange gespeichert und anschließend nach dem First-in-First-Out-Prinzip (FIFO) abgearbeitet [45]. Es gibt verschiedene Arten von Message-Queueing-Systemen in Bezug auf den Konsum der Nachrichten: Beim Point-to-Point-Queueing konsumiert nur einer (aus mehreren möglichen) mit der Warteschlange verbunden Empfänger eine Nachricht, vgl. Lastverteilung zum einmaligen Senden einer E-Mail über mehrere Dienste. Beim Publish/Subscribe-Queueing, wo das Muster verwendet, werden Nachrichten an mehrere Abonnenten verteilt.

Nachrichtenvermittlung. Message Brokering bezieht sich auf die Verwendung eines oder mehrerer miteinander verbundener Vermittler, welche die Weiterleitung von Nachrichten zwischen Produzenten und Konsumenten übernehmen. Ein Broker ist in der Lage Nachrichten zu filtern, zu transformieren und zu weiterzuleiten, um sicherzustellen, dass sie an die richtigen Empfänger gelangen. Publish/Subscribe kann als eine Form des Message Brokering betrachtet werden, bei der Broker Nachrichten an registrierte Abonnenten verteilen.

2.2.2 Entkopplungen

In Publish/Subscribe-Systemen existieren verschiedene Entkopplungsdimensionen, welche die Interaktion zwischen den Veröffentlichenden (Publisher) und den Abonnenten (Subscriber) regulieren. Diese Dimensionen ermöglichen eine flexible Kommunikation zwischen den Komponenten des Systems, indem sie die Abhängigkeiten verringern und die Skalierbarkeit verbessern [45]:

- Räumlich: Publisher und Subscriber müssen nicht in derselben Domäne existieren. Die Teilnehmer können Nachrichten veröffentlichen sowie empfangen, ohne die Identität oder den Standort des Gegenübers zu kennen.
- Zeitlich: Sender und Empfänger einer Nachricht müssen nicht gleichzeitig aktiv sein. Die Veröffentlichung von Nachrichten durch Publisher kann zu beliebigen Zeitpunkten erfolgen, während die Empfangsmöglichkeit durch Subscriber an die Verfügbarkeit der jeweiligen Nachrichten geknüpft ist. Dadurch können die Komponenten des Systems asynchron arbeiten und unabhängig voneinander agieren.
- Zustand: Jede Nachricht wird als eigenständige Einheit behandelt, sodass es weder persistente Verbindungen noch Sitzungen zwischen den Komponenten gibt, über welche ein gemeinsamer Zustand geteilt würde.
- Synchronisation: Der jeweils lokale Programmfluss beim Publisher beim Senden sowie beim Subscriber während des Empfangs einer Nachricht wird nicht blockiert. Die Erzeugung und Verarbeitung von Ereignissen erfolgt außerhalb des Hauptflusses der Programmlogik und ist daher nicht synchron.

- Inhaltlich: Sender und Empfänger müssen sich nicht auf die genaue Struktur oder den Inhalt der Nachrichten einigen. Publisher können Nachrichten mit beliebigen Inhalten veröffentlichen, während Subscriber selektiv diejenigen Nachrichten empfangen können, die für sie von Relevanz sind. Dabei ist es nicht erforderlich, dass sie die interne Struktur der Nachrichten kennen.

2.2.3 Akteure und Dienste

In Publish/Subscribe-Systemen existieren es mehrere Definitionen und Komponenten, die eine tragende Rolle bei der Nachrichtenvermittlung einnehmen. Abbildung 2.2 visualisiert die nachfolgenden vorgestellten Begriffe in einer vereinfachten Darstellung eines Ereignissystems mit den Akteuren und Aktionen [104, 169, 45]. Mehrere Verleger veröffentlichen Ereignisse an einen Nachrichtendienst. Konsumenten, die sich für die Nachrichten interessieren, abonnieren mit einem Filterausdruck. Ein Nachrichtendienst korreliert eingehende Ereignisse mit Abonnements und benachrichtigt die entsprechenden Konsumenten mit dem Ereignis.

Ein *Publisher* (Verleger) ist eine Entität, die Nachrichten erzeugt und in das Pub/Sub-System zur Veröffentlichung einspeist. Ein Publisher sendet Nachrichten zu bestimmten Themen oder Kanälen, ohne sich um die Empfänger der Nachrichten sowie das Routing kümmern zu müssen.

Ein *Subscriber* (Abonnent) stellt eine Entität dar, welche die Rolle des Empfängers von Nachrichten in einem Publish/Subscribe-System einnimmt. Subscriber interessieren sich für bestimmte Themen oder Nachrichtentypen und empfangen nur Nachrichten, die zu diesen Themen passen.

Ein *Event* (Ereignis) ist eine Anzeige im System und wird von Publishern veröffentlicht. Subskribenten haben die Möglichkeit sich für spezifische Ereignisse registrieren, um entsprechende Benachrichtigungen erhalten, wenn diese Ereignisse auftreten.

Eine *Message* (Nachricht) ist das eigentliche Datenpaket, das zwischen Publishern und Subskribenten ausgetauscht wird. Sie enthält die Informationen über ein Ereignis, das übertragen werden soll und weist verschiedene Formate wie Text, JSON, XML usw. auf.

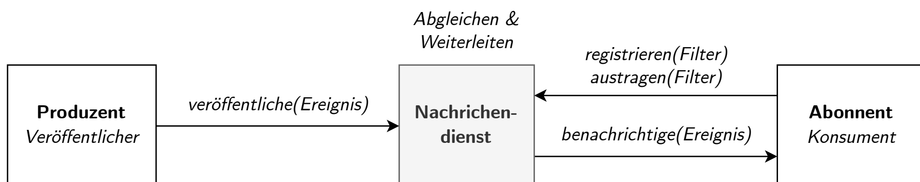


Abbildung 2.2: Ereignissystem für Publish/Subscribe, angelehnt an [45, 104]

Unter dem Begriff *Subscription* (Abonnement) werden die Registrierungen von Subskribenten für bestimmte Ereignisse zusammengefasst. Dazu konfiguriert ein Subscriber das Thema, den Kanal oder einen Filterausdruck über Nachrichten, die empfangen werden möchten. Message-Broker propagieren das Abonnement zur Aktualisierung der Weiterleitungsregeln von den Ereignisquellen und leiten entsprechend dieser Regeln Nachrichten an die Ereignisempfänger weiter.

Unter einem *Topic* (Thema) oder Channel (Kanal) werden logische Kategorien oder Bezeichnungen verstanden, die Nachrichten gruppieren. Publisher senden Nachrichten an bestimmte Themen oder Kanäle, und Subscriber können sich für bestimmte Themen oder Kanäle registrieren, um Nachrichten zu empfangen, die diesem Thema zugeordnet sind.

Der *Message Broker* (Nachrichtenvermittler) stellt eine Umsetzung des Nachrichtendienstes in Publish/Subscribe-Systemen dar, wobei er für die Verteilung von Nachrichten zwischen Veröffentlichern und Abonnenten verantwortlich ist. Ein Broker nimmt Nachrichten von Publishern entgegen und leitet sie basierend auf Weiterleitungsregeln oder -kriterien (Routing) an die entsprechenden Subscriber weiter. Zwischen Sender und Empfänger können mehrere Broker existieren, die in unterschiedlichen Netzwerktopologien miteinander verbunden sind.

2.2.4 Routing

Routing in Publish/Subscribe-Systemen bezeichnet den Prozess der Nachrichtenlieferung oder -verteilung. Es ermöglicht die effiziente Übertragung von Nachrichten von Verlegern über den Nachrichtendienst hin zu den entsprechenden Abonnenten, basierend auf deren Interessen oder Präferenzen. Durch die vorausgegangenen Subskriptionen von Abonnenten und deren Verbreitung hin zu Publishern ist die gegenläufige Nachrichtenleitung durch die Topologie des Nachrichtendienstes zu den beabsichtigten Empfängern gegeben. Die Adressierung ermöglicht einen Abgleich der Nachrichten- oder Gruppierungscharakteristik mit den Weiterleitungstabellen und Binnendiensten, wie beispielsweise Nachrichtenvermittlern.

Kanäle stellen die einfachste Form der Adressierung dar. Dabei geben Produzenten und Abonnenten den Namen eines Kanals explizit an [104].

Beim subjektbasierten Routing erfolgt die Nachrichtenübermittlung an ein spezifisches Thema, engl. topic-based routing. Das Adressierungsschema folgt einer Baumstruktur mit ansteigender Granularität, wobei die einzelnen Ebenen durch ein definiertes Sonderzeichen getrennt werden. Abonnenten können sich für bestimmte (Unter-)Themen registrieren und erhalten dann nur Nachrichten, die diesem Thema entsprechen. Die Methode beschränkt die Ausdrucksfähigkeit auf die Sicht von einem Pfad im Themenbaum. Alternative Perspektiven mit geänderten Teilbäume resultieren in zusätzliche Nachrichten und exponentiellem Anstieg der Datenstruktur

Beim inhaltsbasierten Routing werden Nachrichten auf Basis ihres Inhalts bzw. deren Attributen weitergeleitet, engl. content-based routing. Das System analysiert den Inhalt der Nachricht und leitet sie an die entsprechenden Abonnenten weiter, sofern sie gegen den Filterausdruck einer Subskription erfüllen. Das inhaltsbasierte Routing ermöglicht eine präzisere Weiterleitung von Nachrichten, da sie nicht nur auf Themen, sondern auch auf spezifischen Inhalten basiert. Die Verwendung komplexer Filterausdrücke über die Attribute einer Nachricht kann jedoch zu einer höheren Last für auswertende Dienste führen [96, 45].

Soll die Weiterleitung auf Basis des Types einer Nachricht stattfinden, ist das typenbasierte Routing eine weitere Möglichkeit zur Nachrichtenverteilung in Publish/Subscribe [46].

Weiterführend kann Routing in Publish/Subscribe-Systemen Dienstgüteaspekte (Quality of Service) und Selbstorganisation berücksichtigen. Dies umfasst beispielsweise die Gewährleistung der Zustellung von Nachrichten in der richtigen Reihenfolge, die Vermeidung von Nachrichtenverlusten sowie die Minimierung von Latenzzeiten (vgl. [11, 181, 74]).

2.2.5 Protokolle

Transportprotokolle nehmen eine entscheidende Rolle bei der Kommunikation zwischen verschiedenen Systemen, Anwendungen und Geräten in verteilten Umgebungen ein. In Publish/Subscribe-Systemen sind sie darauf ausgelegt Daten zwischen Softwarekomponenten über Nachrichten asynchron vom Sender zu einem oder mehreren Empfängern zu übertragen. Dabei können die Nachrichten verschiedene Arten von Daten enthalten, wie zum Beispiel Text, Binärdaten oder strukturierte Informationen.

In zahlreichen Protokollen dieses Paradigmas sind Mechanismen zur Gewährleistung der Nachrichtenzustellung und -integrität implementiert, um sicherzustellen, dass Nachrichten nicht verloren gehen oder beschädigt werden. Dies umfasst häufig Funktionen wie Bestätigungen, Wiederholungsmechanismen und Fehlererkennung. Abbildung 2.3 zeigt ausgewählte Schichten des OSI-Modells[2] Standards/Protokolle als Vertreter. Publish/Subscribe-Protokolle sind auf der siebenten Schicht implementiert und setzen auf Anwendungsebene vorgenannte Funktionalitäten um. Das Internet der Dinge (IoT), verteilte Systeme, Echtzeitkommunikation, Unternehmensintegration und Finanzdienstleistungen sind prominente Anwendungsdomänen nachrichtenbasierter Transportprotokolle. Folgend werden ausgewählte nachrichtenbasierte Transportprotokolle vorgestellt, die derzeit einen fortgeschrittenen Reifegrad bzw. eine hohe Marktdurchdringung aufweisen und für Publish/Subscribe genutzt werden können [145, 106, 107]:

[2] Das *Open Systems Interconnection* Referenzmodell stellt eine siebenstufige Schichtenarchitektur für Netzwerkprotokolle dar

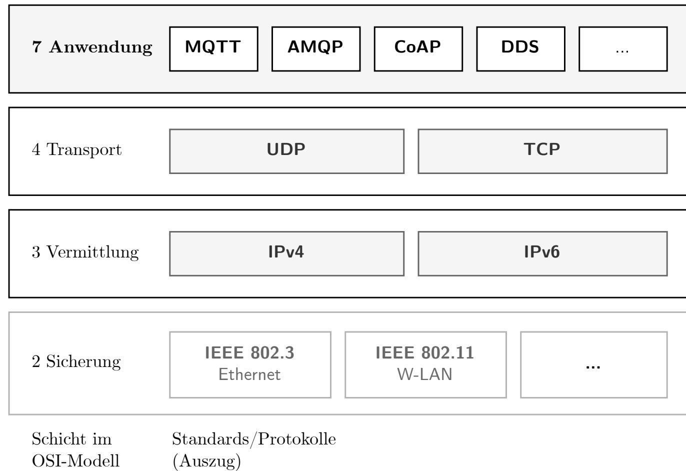


Abbildung 2.3: Publish/Subscribe Protokollstapel, angelehnt an [106, 107]

Java Message Service (JMS) [78], Nachfolger Jakarta Messaging [39]: Der JMS-Standard ist integraler Bestandteil der Enterprise Edition (EE) und bildet die Grundlage für die Implementierung von Nachrichtenverarbeitungssystemen in Java. JMS definiert eine Reihe von Schnittstellen und Protokollen, welche es Java-Anwendungen ermöglichen, Nachrichten auszutauschen und zu verarbeiten. Des Weiteren umfasst es diverse Nachrichtenmuster, wie Point-to-Point und Publish/Subscribe, sowie Funktionalitäten wie Transaktionen, Zuverlässigkeit und Sicherheit.

Message Queuing Telemetry Transport (MQTT): Ein leichtgewichtiges, auf TCP/IP basierendes Protokoll, das für die zuverlässige Übertragung von Nachrichten zwischen Geräten in IoT- und M2M-Anwendungen entwickelt wurde. MQTT verwendet das Publish/Subscribe-Modell und bietet eine effiziente Kommunikation mit geringem Overhead [9, 8].

Advanced Message Queuing Protocol (AMQP): Ein offener Standard für die Nachrichtenübermittlung, welcher auf einer vereinheitlichten Architektur für den Austausch von Nachrichten zwischen Anwendungen basiert. AMQP nutzt ein binäres Netzwerkprotokoll, das den zuverlässigen Austausch zwischen Geschäftsanwendungen gestattet [61].

Simple Text Oriented Messaging Protocol (STOMP): Das textbasierte Protokoll für die Nachrichtenübertragung über verschiedene Netzwerke setzt Aufrufe vergleichbar zu HTTP um. Gegenüber anderen Publish/Subscribe unterstützenden Protokollen wie AMQP und MQTT weist es einen geringen Overhead auf. Die Verfügbarkeit in Nachrichtenbrokern und Programmbibliotheken zur Anbindung an Klienten ist umfassend [164].

Data Distribution Service (DDS): Ein Protokoll, das für den zuverlässigen Austausch von Echtzeitdaten zwischen verteilten Systemen entwickelt wurde. DDS

stellt eine leistungsstarke Middleware für die Kommunikation in komplexen, zeitkritischen Anwendungen bereit [115]. Es existieren sowohl proprietäre als auch quelloffene Implementierungen von DDS in verschiedenen Programmiersprachen, darunter OpenDDS [111], Eclipse Cyclone DDS [40] und RTI Connext [137].

WebSockets: Obwohl WebSockets [49] kein spezifisches Nachrichtenprotokoll darstellen, ermöglichen sie die bidirektionale Kommunikation zwischen Client und Server über eine persistente Verbindung. WebSockets werden häufig für Echtzeitkommunikation in Webanwendungen verwendet und können auch als Transportmittel für die Übertragung von Nachrichten in einem Publish/Subscribe-Modell eingesetzt werden [37]. Beispielsweise bietet die Python-Programmierbibliothek *FastAPI Websocket Pub/Sub* Publish/Subscribe-Kommunikation über WebSockets [51], die JavaScript Laufzeitumgebung *Bun* stellt serverseitig Schnittstellen für Pub/Sub bereit [26] und die Cloud-Computing-Plattform Microsoft Azure bietet mit der WebSocket API und dem Azure Web PubSub Service SDK Pub/Sub-Funktionalität zur Verfügung [99].

Extensible Messaging and Presence Protocol (XMPP): Ein offenes Protokoll für die Nachrichtenübermittlung, das ursprünglich für Instant Messaging entwickelt wurde, aber auch für Echtzeitkommunikation in anderen Anwendungen zum Einsatz kommen kann. XMPP unterstützt verschiedene Messaging-Muster, darunter Publish/Subscribe über eine Erweiterug [37, 142, 101].

Constrained Application Protocol (CoAP): Ein speziell für das Internet der Dinge (IoT) entwickeltes Protokoll, das auf REST-Prinzipien basiert. CoAP wurde für die Kommunikation zwischen IoT-Geräten mit begrenzten Ressourcen optimiert und bietet effiziente Nachrichtenübertragung über UDP oder TCP/IP. Es unterstützt Observe-Mechanismen für die Kommunikation Publish/Subscribe [150].

OpenWire: Ein binäres Protokoll stellt die Standardlösung des Apache ActiveMQ-Nachrichtenbrokers dar [171]. OpenWire bietet eine kompakte Übertragung von Nachrichten zwischen Klienten und dem Nachrichten-Broker und unterstützt verschiedene Kommunikationsmuster wie Point-to-Point und Publish/Subscribe.

Die genannten Protokolle weisen unterschiedliche Funktions- und Eigenschaftsprofile auf, sodass sie den Anforderungen diverser Anwendungen und Einsatzszenarien gerecht werden können. Die Wahl des adäquaten Protokolls hängt von diversen Faktoren wie Leistung, Zuverlässigkeit, Skalierbarkeit und den spezifischen Anforderungen der Anwendung ab (vgl. [188, 106, 153]).

2.2.6 Broker

Mit dem Aufkommen verteilter Systeme begannen erste Forschung an Nachrichtenvermittlern, engl. Message Broker, aber mit den Entwicklungen von Cloud Computing, IoT (Internet der Dinge) und Microservices hat die Bedeutung von

Messaging-Lösungen stark zugenommen. Während frühe Implementierungen von Message Brokern oftmals monolithisch und schwerfällig waren, zeichnen sich moderne Lösungen in der Regel durch eine hohe Skalierbarkeit und Verfügbarkeit aus. Des Weiteren bieten sie Funktionalitäten wie Nachrichtenwarteschlangen, themenbasiertes Routing, Persistenz, Sicherheit und Monitoring. Die Entwicklung von Message Brokern im Bereich Publish/Subscribe hat in den letzten Jahren ein starkes Wachstum erlebt. Ein Überblick über die Entwicklungen und Auszug von derzeit prominenten Implementierungen [144, 94]:

Der Java Message Service (JMS) [78] stellt eine API zur Verfügung, die eine Standardmethode zum Erstellen von Anwendungen für die asynchrone Kommunikation zwischen verteilten Client-Programmen bietet. Es definiert eine gemeinsame Menge von Schnittstellen und Konzepten für Messaging-Systeme und ermöglicht die Entwicklung von plattformunabhängigen Messaging-Anwendungen. JMS wird von verschiedenen Message Broker Implementierungen unterstützt, darunter Apache ActiveMQ, GlassFish, IBM MQ und Oracle WebLogic JMS.

RabbitMQ [21] ist ein Open-Source-Message-Broker, der auf dem Advanced Message Queuing Protocol (AMQP) basiert. Er unterstützt verschiedene Nachrichtenmuster, einschließlich Publish/Subscribe, Nachrichtenwarteschlangen und direkten Austausch.

Apache ActiveMQ [170, 158] ist ein leistungsstarker und flexibler quelloffener Message-Broker, welcher verschiedene Protokolle wie AMQP, MQTT, OpenWire und STOMP unterstützt. Zu den angebotenen Funktionen zählen unter anderem Nachrichtenwarteschlangen, themenbasiertes Routing sowie integrierte Sicherheitsmechanismen.

NATS [28] ist ein leichtgewichtiger und schneller Nachrichtenvermittler, der für Cloud-nativen Anwendungen optimiert ist. Es bietet einfache Konfiguration, hohe Leistung und Skalierbarkeit sowie Unterstützung für die Protokolle TCP, MQTT und WebSockets.

Amazon Simple Notification Service (SNS) [3] ist ein von Amazon Web Services (AWS) verwalteter Nachrichtendienst, der Publish/Subscribe-Messaging unterstützt. Der Dienst ermöglicht Entwicklern die Erstellung von Themen, zu denen Publisher Nachrichten senden können, sowie von Abonnenten, die Nachrichten von diesen Themen empfangen können. SNS unterstützt verschiedene Protokolle für die Kommunikation mit Klienten, darunter HTTP/HTTPS, *Amazon SQS (Simple Queue Service)* und eigenen Datenverarbeitungsdienst *AWS Lambda*.

Google Cloud Pub/Sub [64] ist ein verwalteter Nachrichtendienst in der Google Cloud Plattform, der für hohe Skalierbarkeit, Zuverlässigkeit und globale Verfügbarkeit optimiert ist. Es unterstützt verschiedene Nachrichtenmuster, darunter Publish/Subscribe. Zu den angebotenen Funktionen zählen Push- und Pull-Benachrichtigungen, Latenzsteuerung sowie integrierte Sicherheit. Die Kommunikation zwischen Klienten und Dienst erfolgt über ein proprietäres Protokoll. Ferner bietet es auch Unterstützung für das gRPC-Protokoll für die bidirektionale Kommunikation zwischen Clients und dem Pub/Sub-Service.

Im Forschungsbereich existieren verschiedene Anwendungen zu verteilter Publish/Subscribe-Middleware, die Routingalgorithmen und andere Metriken erforschen. Dazu zählen unter anderem PADRES [73], Hermes [122] und Rebeca [116].

2.3 Selbstorganisierende Systeme

Selbstorganisierende Systeme sind von der natürlichen Welt inspiriert, wo komplexe Verhaltensweisen aus der Interaktion einfacher Komponenten ohne zentrale Kontrolle entstehen. Die Motivation, selbstorganisierende Systeme zu entwickeln, liegt in ihrer Fähigkeit, sich an veränderte Umgebungen anzupassen, Robustheit zu erreichen und neue, nicht explizit programmierte Eigenschaften zu zeigen. In verschiedenen wissenschaftlichen Disziplinen wie Informatik, Biologie, Soziologie und Wirtschaftswissenschaften bietet der Ansatz der Selbstorganisation vielversprechende Möglichkeiten, um komplexe Probleme dezentral und effizient anzugehen [119].

Organic Computing (OC) als Teilmenge von selbstorganisierenden Systemen konzentriert sich insbesondere auf den Entwurf und die Konstruktion von Computersystemen, die von biologischen Organismen inspirierte Eigenschaften hinsichtlich der Komplexität, Anpassung und Widerstandsfähigkeit aufweisen. OC-Computersystemen beobachten die Umgebung und erhalten Rückmeldung, um ihr Verhalten und ihre Konfiguration als Reaktion auf interne oder externe Änderungen anzupassen. Die vom Organic Computing inspirierten Eigenschaften zur Selbstorganisation finden in zahlreichen Teilgebieten der Informatik Anwendung, u.a. [35]:

Selbstheilende Netzwerke: In der Telekommunikation können die Prinzipien des Organic Computing angewendet werden, um selbstheilende Netzwerke zu schaffen, die in der Lage sind, Fehler oder Ausfälle automatisch zu erkennen und zu beheben. Durch den Einsatz von Techniken wie verteilter Überwachung, Fehlervorhersage und dynamischer Rekonfiguration können diese Netzwerke ohne manuelle Eingriffe ein hohes Maß an Zuverlässigkeit und Verfügbarkeit aufrechterhalten.

Robotik: Organic Computing-Konzepte sind maßgeblich an der Entwicklung autonomer Roboter beteiligt, die in unvorhersehbaren Umgebungen wie Katastrophengebieten oder auf Planetenoberflächen navigieren können. Diese Roboter nutzen adaptive Steuerungsalgorithmen, Sensorfusionstechniken und Lernmechanismen, um ihre Umgebung wahrzunehmen, Aktionen zu planen und auf unerwartete Hindernisse oder Geländeveränderungen zu reagieren. Im Rahmen der Schwarmrobotik erfolgt eine Koordination einer Vielzahl einfacher Roboter, sodass Aufgaben gemeinsam und ohne zentrale Steuerung ausgeführt werden können. Jeder Roboter arbeitet auf der Grundlage lokaler Interaktionen mit seinen Nachbarn, was zu neuen Verhaltensweisen wie kollektiver Bewegung, Erkundung und Aufgabenverteilung führt.

Intelligente Netze: Peer-to-Peer-Netzwerke (P2P) basieren auf selbstorganisierenden Prinzipien, um Ressourcen zu verteilen und die Kommunikation zwischen Peers dezentral zu verwalten. Ein Beispiel für ein solches Netzwerk ist das Filesharing mit BitTorrent, in dem Nutzer zusammenarbeiten, um Dateien zu verteilen und herunterzuladen, ohne dass zentrale Server erforderlich sind. In Energiemanagementsystemen können Prinzipien des Organic Computing zum Aufbau intelligenter Netze genutzt werden, welche die Energieverteilung und den Energieverbrauch basierend auf Echtzeit-Nachfrage- und Angebotsbedingungen durch verteilte Steuerungsalgorithmen optimieren. Die Nutzung selbstorganisierender Prinzipien wie adaptiver Algorithmen, prädikativer Analysen und dezentraler Kontrollmechanismen dient dem Lastenausgleich, Verschwendungsminimierung sowie der Stabilitätssicherung des Stromnetzes (Smart Grid).

Biologisch inspirierte Algorithmen: Organic Computing umfasst auch die Entwicklung biologisch inspirierter Algorithmen zur Optimierung, Planung und Entscheidungsfindung. Beispiele hierfür sind genetische Algorithmen, neuronale Netze und Techniken der Schwarmintelligenz, die das Verhalten biologischer Systeme nachahmen, um komplexe Rechenprobleme effizient zu lösen. Algorithmen wie Ant Colony Optimization (ACO) sind ebenfalls von der Natur inspiriert und werden zur Lösung von Optimierungsproblemen wie dem Problem des Handlungsreisenden eingesetzt werden.

Ein tangierendes Forschungsgebiet ist Autonomic Computing, das darauf abzielt, selbstverwaltende Computersysteme zu entwickeln, wobei das menschliche Nervensystem als Inspiration dient. Zur Umsetzung der Selbstverwaltung wurde das Referenzmodell MAPE-K eingeführt, dessen Kontrollzyklus wiederholt durchlaufen wird. In der Beobachtungsphase (Monitor) erfolgt die Erfassung des Zustandes der Umgebung. Die Analysephase umfasst die Auswertung der gesammelten Daten sowie die Identifizierung von Mustern (Analyze). Daraufhin werden Strategien entwickelt (Plan), die zur Ausführung gebracht werden (Execute). Eine Wissensbasis (Knowledge) wird mit aktuellen Erfahrungen aktualisiert und steht in den nachfolgenden Durchläufen zur Verfügung [24, 4, 84].

2.3.1 Self-X Eigenschaften

In selbstorganisierenden Systemen spielen Self-X-Eigenschaften eine entscheidende Rolle. Diese beziehen sich auf die Fähigkeit eines Systems, sich selbst zu organisieren, zu regulieren, anzupassen und zu verbessern, ohne dass dabei eine direkte externe Kontrolle oder Intervention erforderlich ist. Die folgenden Fähigkeiten kennzeichnen unter anderem Self-X-Systeme [36, 84, 103, 173]:

Self-Organization (Selbstorganisation) lässt spontan komplexe Strukturen oder Verhaltensweisen entwickeln, indem sie lokale Interaktionen zwischen ihren Bestandteilen nutzen, ohne dass eine zentrale Steuerung vorhanden ist. Ein bekanntes Beispiel ist die Schwarmintelligenz von Insekten oder die Bildung von Zellen in biologischen Geweben.

Self-Adaptation (Selbstanpassung) befähigt Systeme sich an veränderte Umgebungen oder Anforderungen anzupassen, indem sie ihre Struktur oder Verhaltensweise dynamisch ändern. Dies kann durch Mechanismen wie Feedbackschleifen, Evolution oder Lernprozesse geschehen. Beispiele sind Kontrollschleifen wie MAPE-K oder ein neuronales Netzwerk, das sich durch wiederholtes Training an neue Daten anpasst.

Self-Configuration (Selbstkonfiguration) ist das Ergebnis eines Systems aus der Analyse seiner Umgebung und seines eigenen Zustands in Verbindung mit der Fähigkeit, auf der Grundlage vordefinierter Ziele und Richtlinien selbständig Änderungen an seiner eigenen Konfiguration vorzunehmen. Beispiele hierfür sind Netzwerk-Router, die ihre Routing-Tabellen automatisch aktualisieren, wenn neue Geräte hinzugefügt oder entfernt werden, oder ein Cloud-System, das automatisch zusätzliche Ressourcen bereitstellt, wenn die Nachfrage steigt.

Self-Optimization (Selbstoptimierung) ist die Eigenschaft, dass selbstorganisierende Systeme dazu neigen, ihre Leistung oder Effizienz im Laufe der Zeit zu verbessern, indem sie interne Ressourcen effektiver nutzen oder ihre Struktur anpassen. Evolutionäre Algorithmen sind ein Beispiel, die durch natürliche Selektion verbesserte Lösungen finden.

Self-Repair (Selbstreparatur) lässt Systeme Fehler oder Schäden ohne externe Intervention erkennen und reparieren. Dies kann durch redundante Mechanismen, Selbstüberwachung oder Regenerationsprozesse gewährleistet werden. Ein Vergleich aus der Biologie ist das menschliche Immunsystem, das körpereigene Zellen erkennt und bekämpft, die als fremd oder schädlich erkannt werden.

Self-Healing (Selbstheilung) strebt nach schwerwiegenderen Schäden oder Störungen eine selbstständige Wiederherstellung an. Dies kann durch die Aktivierung von Ruhezuständen, die Umleitung von Ressourcen oder die Mobilisierung von externen Hilfsmitteln erreicht werden. Ein Beispiel ist ein Computernetzwerk, das automatisch auf den Ausfall einer Hardwarekomponente reagiert und den Betrieb aufrechterhält.

2.3.2 Dienstgüte

Die Dienstgüte, engl. Quality of Service (QoS), in verteilten Systemen umfasst eine Vielzahl von Aspekten, die das vom System bereitgestellte Serviceniveau definieren. Dienstgüte wird als die Wirkung der Leistungsfähigkeit eines Dienstes, Netzwerkes und der Verwaltung beschrieben, welche den Grad der Zufriedenheit eines Benutzers desselben Dienstes bestimmt [38].

Die Betrachtung von Dienstgüte-Parametern folgt dabei einem Kreislauf ausgehend vom Benutzer, der Dienstgüte anfordert mit Angaben über einen Leistungsaspekt, die vom Dienstleister erwartet werden. Bei mehreren Aspekten können die einzelnen Anforderungen unterschiedlich gewichtet werden, damit der Dienstleister nach Übersetzung in Parameter und Metriken eine Priorisierung umsetzen

kann. Dafür offeriert ein Diensteanbieter geplante QoS-Aspekte mit Zielwerten oder -spannen an, die den Gütegrad definieren. Der tatsächlich erreichten Grad bei der Ausführung spiegelt die erreichte Dienstgüte des Dienstanbieters wider. Dabei kann die Leistung je Aspekt innerhalb oder auch außerhalb der zuvor angebotenen Werte liegen. Der Benutzer nimmt die rückgemeldete Dienstgüte wahr, indem bei quantitativen Aspekten die Erwartungen aus angeforderten und angebotenen Dienstgüteaspekten mit den erzielten gegenübergestellt werden sowie bei qualitativen Aspekten die Erwartungen aus dem Anwendungsfall resultieren.

Dienstgüte umfasst alle Systemressourcen, die oft in aktive Ressourcen wie Prozessoren, Busse oder Netzwerkadapter, passive Ressourcen wie Speicher und Zwischensysteme wie Router oder Switches unterteilt werden. Der Begriff Dienstgüte umfasst auch die Leistungsfähigkeiten an Klienten, dem Zugang zu Netzwerken sowie deren Durchsatz und Weiterleitungen zwischen Netzwerkbetreibern. Im Kontext der Kommunikationstechnologie sind vier Dienstgüteparameter üblicherweise bestimmend [163, 147]: i) Übertragungskapazität bezeichnet den Durchsatz, der durch die benötigte Datenrate in Abhängigkeit von der Größe der Datenpakete bestimmt wird, ii) Übertragungsverzögerung, unterschieden in lokale durch Verarbeitung an einer Ressource und globale in der Ende-zu-Ende Übertragung, iii) Verzögerungsschwankung (Jitter), die maximale Varianz beim Eintreffen von Daten am Ziel, und iv) Übertragungszuverlässigkeit, die Verlustrate welche in Fehlererkennungs- und -reaktionsmechanismen einfließt.

Objektive (quantitative) Parameter sind durch Beobachtung oder mit Hilfe von Instrumenten messbare Aspekte. Bandbreite, Übertragungsverzögerung, Jitter und Verlustrate sind quantitative Parameter, die ein Dienst von der Netzwerkumgebung anfordert, um in erfolgreicher Kombination aller Parameter die Dienstgüte erbringen zu können [168]. Subjektive (qualitative) Parameter werden durch menschliche Beurteilung klassifiziert, bspw. Bewertung anhand einer vorgegebenen Skala. Diese Aspekte lassen sich in verschiedene Arten von QoS-Attributen einteilen [12, 65, 81, 172]:

- *Zuverlässigkeit* bezieht sich auf die Fähigkeit des Systems, Daten präzise und konsistent, ohne Verlust oder Beschädigung, bereitzustellen. Es umfasst Funktionen wie Fehlertoleranz, Fehlerbehandlung und Datenintegrität.
- Die *Verfügbarkeit* misst den Zeitanteil, in dem ein System betriebsbereit und für Benutzer zugänglich ist. Durch die hohe Verfügbarkeit wird sichergestellt, dass die Dienste auch bei Ausfällen oder Wartungsarbeiten stets verfügbar sind.
- *Leistung* bezieht sich auf die Effizienz und Reaktionsfähigkeit des Systems bei der Bearbeitung von Anfragen und der Bereitstellung von Ergebnissen innerhalb akzeptabler Zeitrahmen. Es umfasst Metriken wie Latenz, Durchsatz und Antwortzeit.

- Unter *Skalierbarkeit* versteht man die Fähigkeit des Systems, steigende Arbeitslasten oder wachsende Benutzerzahlen ohne nennenswerte Leistungseinbußen zu bewältigen. Dabei geht es um Aspekte wie horizontale Skalierbarkeit (Hinzufügen weiterer Ressourcen) und vertikale Skalierbarkeit (Erhöhung der Ressourcenkapazität).
- *Konsistenz* stellt sicher, dass alle Knoten in einem verteilten System die gleiche Datenansicht bieten, selbst wenn gleichzeitige Updates oder Netzwerkpartitionen vorhanden sind. Es umfasst Funktionen wie starke Konsistenz, letztendliche Konsistenz und kausale Konsistenz.
- *Sicherheit* definiert Maßnahmen zum Schutz von Daten, Ressourcen und Kommunikationskanälen vor unbefugtem Zugriff, Manipulation oder Störung und beinhaltet Funktionen wie Authentifizierung, Autorisierung, Verschlüsselung und Audit-Trails.
- Durch *Priorisierung* können bestimmten Arten von Datenverkehr oder Anforderungen Vorrang vor anderen eingeräumt werden, wodurch sichergestellt wird, dass kritische Aufgaben umgehend erledigt werden. Es umfasst Funktionen wie Dienstgüte-Richtlinien, Traffic Shaping (Verkehrsformung) und -Klassifizierung.

Aufrufsemantiken

In verteilten Systemen bezieht sich die Aufrufsemantik auf die Art und Weise, wie die Kommunikation zwischen verschiedenen Komponenten oder Knoten in einem verteilten System erfolgt. Diese Kommunikation wird häufig durch Remote Procedure Calls (RPC), Message Passing oder ähnliche Mechanismen erleichtert. Die Aufrufsemantik definiert, wie Anfragen gestellt werden, wie sie verarbeitet werden und wie Antworten in der verteilten Umgebung gehandhabt werden. In verteilten Systemen werden mehrere gängige Aufrufsemantiken verwendet [167]:

Synchrone Kommunikation. Bei der synchronen Kommunikation wartet der Anrufer darauf, dass der Angerufene die Anfrage verarbeitet und eine Antwort zurückgibt, bevor er fortfährt. Dieser Ansatz ist unkompliziert, kann jedoch zu einer Blockierung des Programmflusses führen, wenn der Angerufene lange braucht, um zu antworten, oder wenn er nicht antwortet.

Asynchrone Kommunikation. Die asynchrone Kommunikation erlaubt es dem Aufrufer, seine Ausführung nach Absenden der Anfrage fortzusetzen, ohne auf die Antwort des Angerufenen warten zu müssen, d.h. der Programmfluss ist nicht-blockierend. Der Angerufene verarbeitet die Anfrage unabhängig und kann später antworten, typischerweise durch Aufrufen eines Rückrufmechanismus (engl. Callback) oder durch irgendeine Form der Benachrichtigung.

Zuverlässigkeit und Fehlertoleranz. Zuverlässigkeit und Fehlertoleranz in der Aufrufsemantik beziehen sich auf die Fähigkeit eines Systems, auftretende Fehler zu verarbeiten und die Korrektheit der Kommunikation trotz potenzieller Fehler oder Störungen in der verteilten Umgebung sicherzustellen. Die folgenden Fehlersemantiken klassifizieren die Zuverlässigkeit. Dabei wird auf die RPC-Fehlersemantiken eingegangen, welche in der Literatur häufig als Referenz herangezogen werden [33, 95, 31]:

- *At-most-once:* Bei einem einseitigen Anruf, auch bekannt als „Fire-and-Forget", sendet der Anrufer eine Anfrage an den Angerufenen, wobei er jedoch keine Antwort erwartet. Dieser Ansatz ist für Szenarien nützlich, in denen der Anrufer das Ergebnis des Vorgangs nicht kennen muss.
- *At-least-once:* Dieser Ansatz stellt sicher, dass eine Anfrage mindestens einmal vom Angerufenen verarbeitet wird, auch wenn dies bedeutet, dass die Anfrage aufgrund von Fehlern oder Netzwerkproblemen mehrmals wiederholt werden muss (garantierte Lieferung, engl. Guaranteed Delivery). Dies sorgt für ein gewisses Maß an Zuverlässigkeit, da garantiert wird, dass keine Anfrage verloren geht, obwohl dies zu einer doppelten Verarbeitung führen kann.
- *Exactly-once:* Die Semantik des Systems gewährleistet, dass eine Anfrage durch den angerufenen Teilnehmer lediglich einmal verarbeitet wird, selbst bei Störungen im Netzwerk. Um eine genau einmalige Semantik zu erreichen, sind häufig komplexere Implementierungen wie verteilte Transaktionen oder Nachrichtendeduplizierung erforderlich. Dies führt zu einer höheren Zuverlässigkeit, da eine doppelte Verarbeitung ausgeschlossen ist, bedingt jedoch eine zusätzliche Last in der Verwaltung.
- *Idempotenter Aufruf:* Idempotente Aufrufe sind solche, die mehrfach wiederholt werden können, ohne dass sich das Ergebnis über die ursprüngliche Anwendung hinaus verändert, wie beispielsweise lesende Zugriffe auf Objektattribute.
  Diese Eigenschaft ist für die Gewährleistung der Fehlertoleranz von entscheidender Bedeutung, da sie im Fehlerfall eine sichere Wiederholung von Vorgängen ermöglicht, ohne dass es zu unbeabsichtigten Nebenwirkungen kommt. Idempotente Aufrufe sind insbesondere in Szenarien von Nutzen, in denen Wiederholungsversuche aufgrund von Netzwerkinstabilität oder vorübergehenden Fehlern häufig auftreten.

Zuverlässigkeits- und Fehlertoleranzmechanismen sind von entscheidender Bedeutung, um die Robustheit und Verfügbarkeit verteilter Systeme sicherzustellen, insbesondere angesichts von Ausfällen, Netzwerkpartitionen und anderen unvorhersehbaren Ereignissen.

Der Einsatz geeigneter Aufrufsemantiken ermöglicht die Aufrechterhaltung der Funktionalität und Konsistenz verteilter Systeme selbst in anspruchsvollen Umgebungen. Die Auswahl der Aufrufsemantik ist von verschiedenen Faktoren abhängig. Dazu zählen die Anforderungen der Anwendung, den Zuverlässigkeits-

und Leistungsmerkmalen des zugrunde liegenden Netzwerks sowie dem gewünschten Grad an Konsistenz und Fehlertoleranz. Verschiedene verteilte Systeme können unterschiedliche Kombinationen von Aufrufsemantiken verwenden, um ihren spezifischen Anforderungen gerecht zu werden.

2.4 Softwareentwicklung

Anwendungsentwicklung, auch bekannt als Softwareentwicklung, engl. Software Engineering, bezieht sich auf den Prozess der Erstellung, Implementierung und Wartung von Softwareanwendungen, um bestimmte Aufgaben zu automatisieren, Probleme zu lösen oder Benutzeranforderungen zu erfüllen. Es handelt sich um einen strukturierten Ansatz zur Entwicklung von Softwarelösungen, der eine Reihe von bewährten Methoden, Werkzeuge und Techniken umfasst [72].

2.4.1 Konfiguration

Konfigurationen erlauben die Anpassung bereits existierender Software, welche Entwickler in neuen Anwendungsentwicklungen einsetzen. Diesbezüglich seien beispielsweise Software-Bibliotheken, wie Protokollierungen (Logging) oder Datenbankinteraktionen, oder Dienste, wie Webserver oder Dienstüberwachungen genannt. Je generischer die Software, desto umfangreicher und ausgereifter können die Konfigurationsmöglichkeiten ausfallen, um diese in diversen Anwendungskontexten angepasst einzusetzen. Die Erstellung von Konfigurationen erfolgt unter Berücksichtigung verschiedener Sichtbarkeiten in unterschiedlichen Phasen der Softwareentwicklung, wobei eine kaskadierende Wirkung zu beobachten ist. In der Konsequenz entstehen zu einem späteren Zeitpunkt oder für eine lokale Anwendung konzipierte Konfigurationen. Sie überschreiben zuvor gesetzte oder solche mit größerer Reichweite.

Bekannt ist dies beispielsweise für Webanwendungen, welche einen Web- oder Anwendungsserver als Laufzeitumgebung haben: Dieser wird mit einer Standardkonfiguration installiert, welche vom Administrator konfiguriert und damit überschrieben werden kann. Für jede bereitzustellende Anwendung können lokale Konfigurationen die dienstglobalen der Instanz überschreiben. Methoden für Sitzungen (Sessions), wie Cookie-Speicherung beim Klienten oder URL[3]-Umschreibungen mit einem eindeutigen Identifikationsparameter für die Zuordnung auf Server-Seite, ermöglichen die Konfigurationsübernahme über mehrere Anfrage/Antwort-Interaktionen zwischen einem Klienten und dem Server unter Verwendung des sonst zustandslosen *Hypertext Transfer Protocol (HTTP)*. Hierbei lassen sich Konfigurationsaspekte auf die jeweilige Sitzung oder gar auf einen einzelnen Aufruf anpassen, welche die vorherigen überschreiben mögen.

[3] Uniform Ressource Locator

Phasen der Softwareentwicklung

Die Softwareentwicklung ist durch einen iterativen Prozess geprägt, der wiederholte Phasen der Anforderungsanalyse, Entwurf, Implementierung, Testen, Bereitstellung, Betrieb und Wartung durchläuft [85, 7]. In verschiedenen Entwicklungsphasen bestehen Konfigurationsmöglichkeiten der Software, die im späteren Betrieb Anwendung finden:

Entwurfs- und Implementierungszeit. Entwurfszeit, engl. Design Time, bezeichnet die Phase im Software-Entwicklungsprozess, in der die Architektur, Struktur und Funktionalität der Software definiert wird. Während dieser Phase werden Anforderungen analysiert, Entwurfsentscheidungen getroffen und Designmodelle erstellt. Entwickler erstellen Entwurfsdokumentationen, UML-Diagramme und Prototypen, um das Design der Software zu kommunizieren und zu dokumentieren. Während der Entwurfsphase können Konfigurationsentscheidungen getroffen werden, um die Struktur und das Verhalten der Software festzulegen. Dies kann beinhalten, wie verschiedene Module oder Komponenten der Software konfiguriert werden, um bestimmte Funktionen zu aktivieren oder anzupassen. In der Implementierungsphase besteht die Möglichkeit, Konfigurationsdateien oder -einstellungen zu erstellen und zu bearbeiten, um das Verhalten der Software anzupassen. Dies kann z.B. die Konfiguration von Datenbankverbindungen, API-Endpunkten oder anderen externen Ressourcen umfassen, die von der Software verwendet werden.

Kompilierzeit. Kompilierzeit, engl. Compile Time, ist der Zeitpunkt, an dem der Quellcode in Maschinencode übersetzt wird, den der Computer ausführen kann. In dieser Phase werden Programmiersprachen und Compiler verwendet, um den Quellcode zu übersetzen und ausführbare Dateien oder Bibliotheken zu erstellen. Kompilierzeitfehler können in dieser Phase entdeckt werden und müssen behoben werden, bevor die Software bereitgestellt wird.

Bereitstellungszeit. Die Bereitstellungszeit, engl. Deployment Time, ist der Zeitraum, in der die Software auf einem Zielrechner oder einer Zielumgebung installiert und konfiguriert wird. Dies umfasst unter anderem die Einrichtung von Servern, die Konfiguration von Datenbanken, das Kopieren von Dateien sowie die Konfiguration von Netzwerkeinstellungen. In dieser Phase erfolgt zudem eine Prüfung der Funktionalität der Software, um deren ordnungsgemäßen Betrieb und die Einsatzbereitschaft für die Benutzer zu gewährleisten.

Laufzeit. Die Laufzeit, engl. Runtime, ist die Phase, in der die Software tatsächlich ausgeführt und von Benutzern verwendet wird, d.h. Menschen oder andere Softwaresysteme interagieren mit der Software, führen Aktionen aus, geben

Eingaben ein und erhalten Ausgaben. In dieser Phase können auch Leistungsüberwachung, Fehlerbehebung und Wartung durchgeführt werden, um sicherzustellen, dass die Software ordnungsgemäß funktioniert und effizient läuft. Ziel ist es, den Benutzern eine effektive und zuverlässige Softwarelösung bereitzustellen, die ihren Anforderungen entspricht und einen Mehrwert bietet. Während des Betriebs und der Wartung können Konfigurationen angepasst und aktualisiert werden, um auf sich ändernde Anforderungen, Fehler oder Leistungsprobleme zu reagieren. Dies kann die Aktualisierung von Konfigurationsdateien, die Anpassung von Einstellungen oder das Hinzufügen neuer Konfigurationsoptionen umfassen.

2.4.2 Ablaufsteuerung

Die Ablaufsteuerung – oder auch Programmfluss – beschreibt die Reihenfolge, in der Anweisungen innerhalb eines Computerprogramms ausgeführt werden. Sie legt fest, wie das Programm Daten verarbeitet, Entscheidungen trifft und mit externen Ereignissen interagiert.

Die „Trennung von Anliegen“, engl. Separation of Concerns (SoC), ist ein Gestaltungsprinzip Software in gut definierte Komponenten aufzuteilen, wobei jede auf ein spezifisches Anliegen konzentriert sein soll. Durch die Trennung von Funktionalität in separate Bereiche wird die Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit der Software verbessert. Das ModelViewControlEntwurfsmuster ist ein Beispiel für eine Trennung mit Aufteilung von Datenzugriff (Model), Geschäftslogik (Control) und Benutzeroberfläche (View).

Das Programmierparadigma „Umkehr der Ausführung“, engl. Inversion of Control (IoC), kann die Umsetzung von Separation of Concerns unterstützen, indem sie die Kontrolle über die Interaktionen zwischen den verschiedenen Komponenten einer Anwendung zentralisiert und dadurch die Trennung weiter fördert.

Das „Hollywood-Prinzip“ besagt *„Don't call us, we'll call you“ (Ruf uns nicht an, wir rufen dich an)*. Es stammt aus der Zeit des klassischen Hollywoods, in dem Schauspieler häufig von Filmstudios rekrutiert wurden. Wenn ein Schauspieler in einem Film mitwirken wollte, musste er nach Registrierung, bspw. durch direktes Vorsprechen oder über einen Katalogeintrag in einer Agentur, nicht (wiederholt) beim Studio über eine mögliche Besetzung nachfragen, sondern das Studio würde den Schauspieler kontaktieren werden, wenn es ihn für eine Rolle auswählte.

In der Softwareentwicklung bedeutet das Hollywood-Prinzip im Wesentlichen, dass hochrangige oder übergeordnete, generische Komponenten (bspw. Frameworks) den Kontrollfluss übernehmen und niederrangige, spezialisierte Komponenten (Anwendungen) aufrufen [97]. Einen übergeordneten Prozess orchestriert, d.h. koordiniert und verwaltet mehrere Computersysteme, Anwendungen oder Dienste. Objekte erstellen und verwalten ihre Abhängigkeiten nicht selbst, sondern werden bei Bedarf von einem externen System injiziert. Vorgenannte Frameworks rufen benutzerdefinierte Callbacks oder Hooks auf, die von Entwicklern

implementiert werden, anstatt dass Entwickler direkt Funktionen im Framework aufrufen. Bei der ereignisgesteuerten Programmierung reagieren Komponenten auf Ereignisse, indem sie sich für Benachrichtigungen registrieren und aufgerufen werden, wenn das Ereignis eintritt, anstatt aktiv nach Ereignissen zu suchen. Das Hollywood-Prinzip fördert eine lose Kopplung zwischen den Komponenten, was zu flexibleren, wartbaren und erweiterbaren Systemen führt.

Methoden der Entkoppelung

Die Trennung von Programmcode zur Konfiguration und Nutzung ist ein wichtiger Aspekt in der Softwareentwicklung, um wartbaren Programmcode zu erstellen und diesen leicht modifizieren zu können. Auch wenn der Programmcode in logische Einheiten gruppiert ist, so kommt es beim traditionellen objektorientierten Ansatz an Stellen zu Abhängigkeiten zwischen zwei Objekten. Die Auslagerung der Erstellung der Abhängigkeiten ist ein Schritt in die richtige Richtung. Die Abhängigkeiten für ein Objekt können über unterschiedliche Methoden bereitgestellt werden:

Manuelle Dependency Injection. Bei der Injektion über den Konstruktor einer Klasse der objektorientierten Programmierung werden die Abhängigkeiten als Parameter übergeben und Feldern zur späteren Nutzung in der Instanz zugewiesen. Eine weitere Möglichkeit ist die Bereitstellung über Set-Methoden. Diese sind eine Konvention in der objektorientierten Programmierung, um den Zustand von gekapselten Feldern in einem Objekt zu setzen. Sie werden auf Basis des Feldnamens und dem Präfix *set* gebildet, [10]. Auch wenn das Testen einer Klasse durch Injektion besser ermöglicht ist, bleibt der Programmcode schlecht wartbar: Wird die Abhängigkeit an verschiedenen Stellen in der Anwendung genutzt, muss das Binden zur Abhängigkeit mehrfach durchgeführt werden.

Fabrikmethode. Die abstrakte Fabrikmethode ist ein Entwurfsmuster, bei der die Erstellung von Abhängigkeiten an eine dritte Klasse ausgelagert wird, die sogenannte Fabrik [58]. Der Objektgraph wird nicht per Hand erstellt, sondern innerhalb einer anderen Klasse. Auch wenn die Fabrikmethode explizit angibt, welcher Art produziert wird, ist der verwendenden Geschäftslogik keine Kenntnis über die Interna gegeben. Der Programmcode wurde durch die Indirektion einer Factory-Klasse in die Erstellung und Nutzung von Services aufgeteilt. Dennoch bleiben Herausforderungen beim Testen des Programmcodes: Über eine zusätzliche statische Methode können Mock-Objekte für Softwaretests gesetzt werden, die eine Instanziierung der Fabrik-Klasse berücksichtigen bzw. eine bereits existierende Instanz zurückgibt.

Service Locator Pattern. Eine Registratur, engl. Registry, dient als Vermittler zwischen einer Dienstanfrage und der verfügbaren Umsetzungsmöglichkeit, vgl.

analoges Verzeichnis zu Dienstanbietern (ugs. Gelbe Seiten), welches die Suche nach Personen oder Unternehmen bietet, die Dienste anbieten. Eine Registratur übernimmt die übergreifende Informationsverteilung und wird daher in der Praxis häufig als Singleton implementiert und als anwendungsglobales Objekt gehalten, dies impliziert, dass es maximal eine Instanz geben darf [52].

Die initiale Konfiguration wird in der Regel in einer oder mehreren Dateien gespeichert, die beim Start der Anwendung in den Registrierungskontext eingelesen wird. In dieser Datei wird die Bindung des Schlüsselwertes zu einer Umsetzung definiert. Dies kann beispielsweise eine Zeichenkette sein, die auf eine Implementierungsklasse verweist, die anschließend instanziiert werden muss. Im Java Enterprise Kontext (Java EE) werden Service Locator für JNDI[4] eingesetzt, um Ressourcen zu registrieren.

Ein Nachteil von Service Locators besteht in den damit versteckten Abhängigkeiten. Aufgrund von Tippfehlern bei der Übergabe des Schlüsselwertes oder fehlender Konfiguration und Einträge in der Registratur sind Tests und Refaktorisierungen schwer umzusetzen, wobei Fehler erst zur Laufzeit auftreten. Zudem ist der Aufruf zu einem Lokalisierungsdienst in der Klasse der Anwendungslogik programmiert.

Strategy Pattern. Das Strategie-Muster, engl. Strategy Pattern, erlaubt die Definition und den Austausch verschiedener Implementierungen, beispielsweise eines Algorithmus, ohne dass eine Modifikation des Aufrufers der Algorithmen erforderlich ist. Es fördert eine flexible und modulare Architektur, indem es den Code so strukturiert wird, dass sich Algorithmen unabhängig von den Klassen, die sie verwenden, austauschen lassen. Eine Kontext-Klasse verwendet die Strategien und enthält eine Referenz auf ein Strategie-Objekt, dessen Methoden sie aufruft, um den Algorithmus auszuführen. Die Strategie stellt eine Schnittstelle oder abstrakte Klasse mit Methodendeklarationen dar, die alle konkreten Strategien durch Implementierung oder Vererbung umsetzen müssen. Der Einsatz des Strategie-Entwurfsmusters lässt bspw. Algorithmen von der Kontrolllogik entkoppeln [57].

Dependency Injection. Bei Anwendung von „Einbringen von Abhängigkeiten“, engl. Dependency Injection (DI), erhalten Komponenten ihre Abhängigkeiten durch ihre Konstruktoren, Methoden oder Felder gesetzt, anstatt sie selbst zu holen oder zu instanziieren, vgl. Hollywood-Prinzip. Die Anwendungslogik wird von der Infrastruktur bzw. der Implementierung getrennt, kann aber auf die Abhängigkeiten vertrauen, wie Implementierungen im Produktiveinsatz oder Mockobjekten für Tests. Von einem Injektor werden die Abhängigkeiten bei Bereitstellung bzw. Start der Anwendung in einem Container bereitgestellt, d.h. Objektimplementierungen instanziiert oder Variablen Werte zugewiesen [178].

[4] Java Naming and Directory Interface

Methoden die strukturelle Trennung von erzeugendem und ausführendem Programmcode sowie die Konfiguration der Bindung betrachtet, fokussiert der Begriff Inversion of Control den Programmfluss einer Anwendung. Im Gegensatz zur klassischen Sichtweise, bei der die Komponenten der Geschäftslogik die Aktionen steuern, wird bei der Inversion of Control-Methode die Laufzeitumgebung als zentraler Akteur betrachtet, der die Orchestrierung der Komponenten übernimmt. Um die Umsetzung dieser Methode zu vereinfachen, wurden zahlreiche Frameworks entwickelt, die generische Funktionalitäten aus der Anwendungslogik bereitstellen und damit die Implementierung und Administration von Anwendungen für Entwickler erleichtern. Neben der losen Kopplung mit Trennung von Implementierung und Ausführung fördert die Modularisierung von Programmcode: Wiederverwendbarer Programmcode mit generalisierten Funktionalitäten als Standardbibliothek in Framework ermöglicht einfachere Administration durch zentrale Anpassungen oder Aktualisierungen, während Anwendung in verschiedenen Umgebungen eines gleichen Standards lauffähig ist, die unterschiedliche Anbieter implementiert haben, was einem *Vendor Lock-in* vorbeugt. Innerhalb der Thematik der Dependency Injection findet zudem der Begriff der *Inversion of Control (IoC)* Verwendung. Dabei stellt die Dependency Injection einen spezifischen Ausdruck der Inversion of Control dar [52].

Frameworks

Frameworks stellen vorgefertigte Strukturen dar, die als Grundlage für die Entwicklung von Softwareanwendungen dienen. Sie stellen eine Sammlung von Bibliotheken, Werkzeugen, Standards und Beispielen bereit, die Entwicklern helfen, effizienter zu arbeiten und qualitative Anwendungen zu erstellen. Sie bieten Vorteile, auf die Entwickler aufbauen können, ohne von Grund auf neu beginnen zu müssen: Frameworks bieten eine vordefinierte Struktur, welche die Entwicklung von Softwareanwendungen erleichtert. Dies umfasst oft eine Hierarchie von Klassen und Modulen sowie Regeln und Konventionen zur Organisation des Codes. Wiederverwendbare Komponenten, die häufig benötigte Funktionen bereitstellen, wie zum Beispiel Datenbankzugriff, Benutzerauthentifizierung oder Routing in Webanwendungen, können von Entwicklern genutzt, anpasst und erweitert, um die gewünschten Funktionen in ihrer Anwendung zu implementieren. Bewährte Methoden und Standards werden Richtlinien und Konventionen vorgeben, was zu konsistentem und wartbarem Programmcode beiträgt. Die Verwendung von Frameworks ermöglicht es Entwicklern Zeit zu sparen, da sie auf vorgefertigte Funktionen und Strukturen zurückgreifen, anstatt alles von Grund auf neu zu erstellen. Dies gestattet ihnen sich auf die Umsetzung von Anwendungslogik zu konzentrieren. Obwohl Frameworks eine vorgefertigte Struktur bieten, sind sie in der Regel flexibel und anpassbar. Entwickler können die Funktionalität des Frameworks erweitern oder anpassen, um spezifische Anforderungen zu erfüllen oder individuelle Anwendungen zu entwickeln [23, 75].

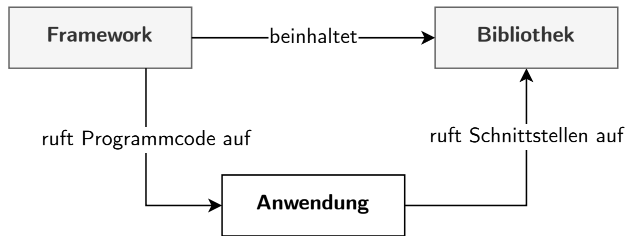


Abbildung 2.4: Aufrufe von Frameworks und Programmbibliotheken

Abgrenzung zu Programmbibliotheken. Programmbibliotheken und deren spezielle Ausprägung Frameworks sind beide Sammlungen von Code und Funktionen, die Entwicklern unterstützen, Softwareanwendungen zu erstellen. Sie erweitern den Funktionsumfang und reduzieren den Entwicklungsaufwand. Obgleich gewisse Parallelen auszumachen sind, bestehen Unterschiede zwischen beiden Konzepten [53]:

Der Kontrollfluss der Anwendungslogik wird durch Frameworks vorgegeben, wie die Anwendung strukturiert und organisiert zu sein hat. Entwickler müssen sich in der Regel an die Struktur und die vorgegebenen Konventionen des Frameworks halten, wie Vererbung von Klassen oder Verwendung von Annotationen. Im Gegensatz dazu stellen Programmbibliotheken lediglich einzelne Funktionen oder Module bereit, welche von Entwicklern nach Bedarf in ihren Programmcode integriert werden können. Die Kontrolle über den Fluss der Anwendungslogik obliegt dem Entwickler.

Entwicklungs- und Laufzeitumgebungen werden von Frameworks meist angeboten, die spezifisch auf die Bedürfnisse der Anwendung zugeschnitten ist. Dies umfasst eine Vielzahl von Funktionen wie Datenbankzugriff, Benutzerauthentifizierung, Routing in Webanwendungen umfassen, bspw. die Implementierung von Anwendungsservern gemäß dem Standard der Java Enterprise Edition. Demgegenüber fokussieren sich Programmbibliotheken normalerweise auf einen bestimmten Bereich oder eine spezifische Funktion und bieten klar definierte Funktionen und Dienste, die unabhängig von der Anwendungsdomäne sind.

Die höhere Ebene der Abstraktion bei Frameworks führt oft Architekturmuster oder Designparadigmen ein. Sie stellen eine Struktur bereit, in der Entwickler ihre Anwendung implementieren können. Programmbibliotheken bieten in der Regel eine niedrigere Abstraktionsebene und stellen einzelne Funktionen oder Dienste bereit, die in die Anwendung integriert werden können, ohne dass sie die gesamte Struktur oder den Fluss der Anwendung beeinflussen. Frameworks implementieren oft das Prinzip von *Inversion of Control (IoC)* und übernehmen die Anwendungssteuerung, während Programmbibliotheken Funktionen anbieten, die Entwickler explizit aufrufen müssen, um sie in ihre Anwendungen zu integrieren, vgl. Abbildung 2.4.

Dependency Injection Frameworks. Ein gängiger Weg, um Inversion of Control in Frameworks umzusetzen, ist die Verwendung von Annotationen. Eine weitere Möglichkeit der Umkehr der Ablauflogik ist die Bereitstellung und Auswertung von Konfigurationen aus einer Datei. Beispielsweise werden hier im Quellcode deklarierte Schnittstellen zu Implementierungen zugeordnet, bei Web-Anwendungen URL-Muster auf Controller-Klassen konfiguriert oder mit einem zeichenbasierten Platzhalter deklarierte Datenbank-Konnektoren reale Verbindungsparameter hinterlegt, die je Laufzeitumgebung variieren können, aber keiner Anpassung am Quellcode benötigen. Ein Anwendungsserver zur Ausspielung von Webanwendungen mit einer Implementierung nach Standard der Jakarta Servlets und Jakarta Server Pages, wie Apache Tomcat, wertet beispielsweise für jede Anwendung einen Application Deployment Descriptor aus [79]. Konfigurationen werden nach vorgegeben Schema in einer XML-Datei vorgehalten, welcher per Konvention im Unterverzeichnis WEB-INF/web.xml einer Anwendung hinterlegt ist und zur Bereitstellungszeit ausgewertet wird.

Eine Auswahl von Frameworks, die Inversion of Control durch Dependency Injection (DI) einsetzen, zeigt die Verwendung in verschiedenen Programmiersprachen:

C# (und .NET): In ASP.NET Core[5], einem Webframework für die Entwicklung von Webanwendungen in C#, wird Dependency Injection als zentrales Konzept verwendet. Ein integrierter DI-Container übernimmt die Registrierung, Verwaltung und Auflösung von Abhängigkeiten in der Anwendung. Abhängigkeiten werden oft als Service-Klassen implementiert. Diese Klassen bieten spezifische Funktionalitäten und werden in der Startup-Klasse der Anwendung registriert, um sie für andere Teile der Anwendung verfügbar zu machen. Der DI-Container unterstützt die Verwaltung verschiedene Lebenszyklen für registrierte Dienste. Zudem werden Abhängigkeiten erstellt und wiederverwendet. *Autofac*[6] ist ein weiteres DI-Framework für .NET, das sich nahtlos in ASP.NET Core integriert, um die Verwaltung von Abhängigkeiten in Webanwendungen zu erleichtern.

Python: In *Django*[7], einem populärem Webframework für die Entwicklung von Webanwendungen in Python, wird Dependency Injection auf eine etwas andere Weise als in anderen Sprachen und Frameworks gehandhabt. In Django erfolgt die Verwaltung von Abhängigkeiten häufig über die Einstellungen und Konfigurationsdateien. An dieser Stelle besteht für Entwickler die Möglichkeit, Module, Klassen und Funktionen zu registrieren und zu konfigurieren, sodass diese von anderen Teilen der Anwendung verwendet werden können. Die Middleware in Django kann als Dependency Injection betrachtet werden, die mit speziellen Hooks in der Lage ist den Anfrage/Antwort-Zyklus zu beeinflussen. Entwickler können Abhängigkeiten in klassenbasierten Ansichten über Mixins oder durch Vererbung von generalisierten Framework-Klassen verwalten und konfigurieren.

---

[5] `https://dotnet.microsoft.com/en-us/apps/aspnet`
[6] `https://autofac.org/`
[7] `https://www.djangoproject.com/`

PHP: Das Container-Framework *PHP-DI*[8] ermöglicht die automatische Injektion von Abhängigkeiten sowohl durch PHP-Annotations als auch durch Konfigurationsdateien umgesetzt werden. PHP-DI orientiert sich an die Standards des PHP-FIG (Framework Interoperability Group)[9], insbesondere an der PSR-11-Spezifikation für Container-Interfaces. Dadurch ist eine nahtlose Zusammenarbeit mit anderen PHP-Frameworks und -Bibliotheken, wie *Symfony*, *Laravel* oder *Zend Framework*, gewährleistet.

Java: Im Rahmen des Java-Standardisierungsprozesses werden akzeptierte Spezifikationsanfragen (JSR) veröffentlicht, welche von Implementierungen umgesetzt werden. JSR-299 (Java Contexts and Dependency Injection) [76], Teil des Standards der Java Enterprise Edition, basiert auf JSR-330 (Dependency Injection für Java) [77], bietet ein leistungsstarkes und umfassendes Modell für die Injektion von Abhängigkeiten, Lebenszyklusmanagement und Kontextverwaltung in Java Geschäftsanwendungen (Java EE), die von vielen Anwendungen und Diensten unterstützt wird, darunter WildFly[10], Eclipse GlassFish[11] oder Weld[12].

Spring[13] zählt zu den bekanntesten und am weitesten verbreiteten Dependency Injection Frameworks für die Programmiersprache Java. Spring erlaubt die Verwendung von XML-, Annotations- und Java-basierten Konfigurationen für die Dependency Injection.

Guice[14] und Dagger[15] sind abschließend als Vertreter von DI-Frameworks genannt, beide von Google verwaltet, die mit Java-Annotationen und rein Java-basierte Konfigurationen Abhängigkeiten verwalten.

2.4.3 Reflektion

Reflektion (Reflexion) im Kontext der Informatik bezeichnet die Fähigkeit eines Systems, seine eigenen Strukturen, Verhalten sowie die Umgebung zur Laufzeit zu analysieren und zu modifizieren. Dies ermöglicht es Softwareprogrammen, auf sich selbst zu reagieren, dynamisch Änderungen vorzunehmen und sich an verschiedene Situationen anzupassen, ohne dass der Quellcode statisch geändert werden muss [92, 50]. Reflexion kann für die folgenden Funktionalitäten genutzt werden:

---

[8] `https://php-di.org/`
[9] `https://www.php-fig.org/`
[10] `https://www.wildfly.org/`
[11] `https://glassfish.org/`
[12] `https://weld.cdi-spec.org/`
[13] `https://spring.io/`
[14] `https://github.com/google/guice`
[15] `https://dagger.io/`

- *Analyse von Metadaten*: Software kann zur Laufzeit ihre eigenen Metadaten analysieren, beispielsweise Klassennamen, Methodensignaturen, Attribute und Annotationen.
- *Dynamische Instanziierung*: Die Möglichkeit, neue Objekte zu generieren und bestehende Objekte zur Laufzeit zu instanziieren, basierend auf Metadaten und Konfigurationen.
- *Dynamisches Aufrufen von Methoden*: Die Software kann zur Laufzeit Methoden von Objekten aufrufen, deren Namen und Signatur zur Laufzeit bekannt sind.
- *Änderung von Datenstrukturen und -verhalten*: Die Software ist in der Lage, zur Laufzeit sowohl Datenstrukturen als auch das Datenverhalten zu modifizieren. Dies erfolgt mit dem Ziel, sich an veränderte Anforderungen anzupassen oder neue Funktionalitäten zu integrieren.
- *Interaktion mit Klassenladern*: Die Software ist in der Lage zur Laufzeit mit dem Klassenlader zu interagieren, um Klassen dynamisch zu laden, zu entladen oder neu zu definieren.

Reflexion wird in verschiedenen Bereichen der Softwareentwicklung eingesetzt, einschließlich Frameworks, Bibliotheken, Debugging-Tools, zur dynamischen Konfiguration und Laufzeitoptimierung. Durch Reflexion profitieren Softwareanwendungen von höherer Flexibilität, Dynamik und Abstraktion, jedoch kann es auch zu Leistungseinbußen und Sicherheitsproblemen führen, sofern sie nicht angemessen eingesetzt wird.

Mit Reflexion kann die Struktur aus dem Programmcode ausgelesen und bearbeitet werden. Zur Laufzeit ist es möglich Informationen zu Klasse zu erhalten, ohne diese Informationen zur Übersetzungszeit zu kennen. So ist es beispielsweise möglich die Konstruktoren und Methoden mit ihren Sichtbarkeiten, Parametertypen und Rückgabewerte auszulesen, die Datentypen und Variablenbezeichner von Attributen sowie welche Schnittstellen implementiert oder Vererbungen realisiert sind [110]. Neben lesenden Funktionalitäten erlaubt Reflexion die dynamische Erstellung neuer Objekte sowie die Methodenausführung an Instanzen, wodurch eine Modifikation des Laufverhaltens möglich ist. Die konkreten Möglichkeiten von Reflexion sind jedoch von der jeweils eingesetzten Programmiersprache abhängig.

Metadaten

Bibliotheken in Programmiersprachen, welche über Reflexion-Routinen verfügen, ermöglichen den Zugriff auf Metadaten zur Laufzeit, deren Auswertung sowie die Umsetzung von Abhängigkeiten und Objektbeziehungen. Die Ausgestaltung von Metadaten sind abhängig von den Entwurfsentscheidungen wie Sichtbarkeit, Kapselung sowie den Möglichkeiten der Programmiersprache:

Konfigurationsdateien werden häufig verwendet, um Metadaten zu speichern. Diese Dateien können in verschiedenen Formaten wie JSON, YAML oder XML vorliegen und entweder von dem Programm oder der Laufzeitumgebung bei der Bereitstellung oder Laufzeit geladen und ausgelesen werden.

Metadaten können auch als *Umgebungsvariablen* im Betriebssystem desjenigen Computers gespeichert werden, auf dem die Anwendung ausgeführt wird. Dies ermöglicht eine einfache Konfiguration und Anpassung der Metadaten für verschiedene Umgebungen. Allerdings erfordert die erweiterte Sichtbarkeit besondere Sorgfalt, insbesondere durch eindeutige Vergabe von Bezeichnern, um ein Überschreiben außerhalb des beabsichtigten Kontexts zu verhindern.

Des Weiteren können spezielle *Software-Komponenten* als Metadaten-Manager fungieren und es den Entwicklern ermöglichen, Metadaten zentral zu verwalten sowie in anderen Systemen und Anwendungen zu integrieren. Ein solcher Dienst kann die interne Datenhaltung unterschiedliche umsetzen, zum Beispiel über eine Datenbank, und über Anwendungsgrenzen hinweg als REST-API angesprochen werden.

Annotationen

Während in der Programmiersprache PHP Metadaten mit Annotationen als Kommentar im Quellcode realisiert werden, um nicht interpretiert zu werden, bieten andere Programmiersprachen Konstrukte zum Einfügen in den Quellcode an, bspw. benutzerdefinierte Attribute in C# im .NET-Framework [66] oder Annotationen in Java [113].

Ein häufiger Anwendungsfall ist das objektrelationale Mapping, object relational mapping (ORM), von objektorientierten Klassen auf die Datenhaltung in relationalen Datenbanken. Frameworks, die in unterschiedlichen Programmiersprachen eingesetzt werden können, bieten diese Funktionalität ebenfalls an. Dazu gehören beispielsweise *Doctrine*[16] in PHP, Hibernate oder *Jakarta Persistence API*[17] in Java. Bei diesen Frameworks handelt es sich um Implementierungen, die mit Annotationen arbeiten. Objekte, die von ORM-Implementierungen verarbeitet werden, werten Metadaten im Quellcode der Klassen aus, um Konfigurationen für Abfragen mit der Persistenzschicht anzuwenden.

Die Programmiersprache Java bietet mit dem Sprachkonstrukt der Annotationen die Möglichkeit Metadaten zu deklarieren. Obgleich sie keinen Einfluss auf die Byte-Code-Generierung ausüben, werden sie vom Compiler auf syntaktische Korrektheit geprüft. Listing 2.1 zeigt die Erstellung einer eigenen Annotation, welche Schnittstellendeklaration ähnelt. Diese wird mit Annotation des Java-Standards versehen, was hier die Phasen für den Zugriff der Reflexion sowie die Stellen des Einsatzes konfiguriert.

[16] `https://www.doctrine-project.org/`
[17] `https://jakarta.ee/specifications/persistence/`

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface MyCustomAnnotation {
    String value() default "";
}
```

Listing 2.1: Deklaration einer eigenen Annotation in Java

2.4.4 Laufzeitumgebung

Eine Laufzeitumgebung, engl. runtime environment, ist die Umgebung, in der eine Anwendung während ihrer Ausführung arbeitet. Dabei wird die notwendige Infrastruktur zur Verfügung gestellt, in der Anwendungen ausgeführt werden können. Dies umfasst die Bereitstellung von Diensten wie Speicherverwaltung, Verwaltung von Threads sowie Sicherheitsmechanismen.

Eine Laufzeitumgebung umfasst zudem Softwarekomponenten, die erforderlich sind, um die Anwendung auszuführen und mit ihrer Umgebung zu interagieren: In Bezug auf das Betriebssystem ist festzuhalten, die Basissoftware stellt die Verwaltung der Hardware-Ressourcen sowie Diensten wie der Dateiverwaltung, Speicherverwaltung und Prozessverwaltung sicher. Laufzeitbibliotheken stellen Sammlungen von Funktionen und Routinen bereit, welche von Anwendungen während ihrer Ausführung aufgerufen werden können. Sie bieten häufig Funktionalität für Aufgaben wie Eingabe/Ausgabe (I/O), Netzwerkkommunikation, Datenbankzugriff und mehr. Virtuelle Maschinen stellen eine Softwarekomponente dar, welche eine abstrahierte virtuelle Umgebung bereitstellt, auf der die jeweilige Anwendung ausgeführt werden kann. Exemplarische Vertreter sind die Java Virtual Machine (JVM) für Java-Anwendungen sowie die Common Language Runtime (CLR) für Anwendungen, welche auf der .NET-Plattform entwickelt wurden.

Eine Laufzeitumgebung stellt eine Reihe von allgemeinen Funktionen bereit, die sich auf eine Vielzahl von Softwareprogrammen anwenden lässt, die für diese Umgebung kompatibel umgesetzt sind, d.h. in einer akzeptierten Programmiersprache und der Verwendung unterstützter Sprachkonstrukte: Eine Anwendung kann auf einem bestimmten System oder einer bestimmten Plattform ausgeführt zu werden. Die Verwaltung von Ressourcen wie Speicher, Prozessoren und Eingabe- und Ausgabegeräten erfolgt durch die Laufzeitumgebung, wobei diese Aufgabe gegebenenfalls an andere Akteure übertragen wird. Dadurch wird sichergestellt, dass die Anwendungen effizient und konfliktfrei arbeiten. Funktionalitäten wie dynamische Bindungen, Fehlerbehandlung, Speicherzuweisung und -freigabe sowie Multithreading werden unterstützt, um die Anwendungsentwicklung zu erleichtern und die Leistung zu optimieren.

Middleware

Eine Middleware stellt eine Softwarekomponente dar, die in verteilten Systemen oder zwischen Anwendungen und Systemkomponenten zum Einsatz kommt. Ihre Funktion besteht in der Unterstützung der Integration und Interaktion der genannten Elemente.

Die wesentlichen Funktionalitäten umfassen unter anderem: Die *Vermittlung* zwischen verschiedenen Softwarekomponenten oder Diensten ist eine grundlegende Voraussetzung für die Interaktion und Kommunikation. Der Datenaustausch und die Integration wird erleichtert, unabhängig davon, ob die beteiligten Komponenten auf derselben oder über verschiedene Plattformen verteilt sind. Die beteiligten Anwendungen oder Systeme nutzen *erweiterte Funktionen* und Dienste, um Aufgaben zu automatisieren oder zu optimieren. Dies umfasst beispielsweise die Umsetzung von Sicherheitsmechanismen, die Verwaltung von Transaktionen oder die Verarbeitung von Nachrichten. Die Umsetzung von *Interoperabilität* zwischen heterogenen Systemen und Plattformen erfolgt durch die Unterstützung mehrerer Kommunikationsprotokolle, Datenformate und Schnittstellenstandards. Folglich können Anwendungen miteinander barriereärmer kommunizieren, unabhängig von ihrer technologischen Umgebung. Die Leistung und Kapazität von Systemen kann mit geringem Wartungsaufwand am Programmcode geändert werden, um sich an neue Anforderungen anzupassen. Vertikale und horizontale *Skalierungen* gestattet es mit steigender Last und wachsenden Datenmengen umzugehen.

Middleware-Implementierungen bieten eine Reihe von *Transparenzen*, um die Komplexität der Systemintegration zu reduzieren und die Entwicklung verteilter Anwendungen zu erleichtern: Der Zugriff auf entfernte Ressourcen kann ermöglicht werden, ohne dass sich Entwickler um deren physischen Standort kümmern zu müssen. Dies bedeutet, dass die Kommunikation mit entfernten Diensten oder Systemen transparent erfolgen kann, als ob sie lokal verfügbar wären (Ortstransparenz). Zugriffsmethoden und Protokolle können abstrahiert werden, sodass Entwickler auf Ressourcen zugreifen können, ohne sich mit den Details der Konfiguration, Netzwerkkommunikation oder Datenübertragung befassen zu müssen (Zugriffstransparenz). Dazu koordinieren Middleware-System den gleichzeitigen Zugriff mehrerer Benutzer auf gemeinsam genutzte Ressourcen, ohne dass die Anwendung dies explizit handhaben muss (Nebenläufigkeitstransparenz). Die Replikation von Diensten kann transparent gestaltet werden, sodass Anwendungen sich nicht der Tatsache bewusst sein müssen, dass mehrere Instanzen dieser Ressourcen vorhanden sind. Eine Middleware stellt automatisch Ressourcen bereit und gibt sie wieder frei, die für die Anforderungen einer Anwendung erforderlich sind (Replikationstransparenz). Die Migration zwischen verschiedenen Systemen oder Plattformen wird erleichtert, ohne dass die Anwendung selbst Änderungen vornehmen muss. Ebenso kann die Mobilität von Benutzern oder Geräten unterstützt werden, indem die Middleware eine nahtlose Weiterführung von Sitzungen oder Prozessen ermöglicht (Mobilitätstransparenz). Des Weiteren können Fehlererkennungen und -behandlungen abstrahiert werden, sodass An-

wendungen die Fehler transparent erkennen und auf sie reagieren können, ohne dass sie die Details der Fehlerbehandlung implementieren müssen (Fehlertransparenz). [13, 186, 161]

2.4.5 Stellvertreter

Das Stellvertreter-Entwurfsmuster, engl. *Proxy Pattern*, ist ein Vertreter aus der Klasse der Strukturmuster. Ein Stellvertreter dient als Platzhalter für den Zugriff auf ein konkretes Subjekt. Mögliche Ausprägungen und Verwendungen von Stellvertretern sind [57, 58]:

- *Remote-Proxy* stellt eine lokale Repräsentation für ein Objekt aus einem anderen Adressbereich, bspw. Java RMI[18] oder Remote EJB[19] im Java Enterprise Framework
- *Virtueller Proxy* lädt ein Objekt erst bei Bedarf, wenn dieses benötigt wird, bspw. eine Entity-Instanz, die über JPA[20] aus der Persistenzschicht abgerufen wird.
- *Schutzproxy* kontrolliert Zugriffe auf das Originalobjekt, bspw., um gegen Autorisation-Mechanismen zu validieren
- *Smart Reference Proxy* ersetzt einen einfachen Zeiger, der es gestattet weitere Aktionen bei Methodenaufrufen auszuführen, bspw. in einen transaktionalen Kontext einbetten oder Logging umzusetzen.

Ein statischer Stellvertreter besteht er aus einer Klasse, die dieselbe Schnittstelle implementiert wie das tatsächliche Zielobjekt. Diese Klasse enthält eine Referenz auf das tatsächliche Zielobjekt und delegiert die Aufrufe an dieses Zielobjekt. Zusätzlich zu den Funktionen, die das Zielobjekt bereitstellt, können im Stellvertreter weitere Funktionen bereitgestellt werden, beispielsweise die Überprüfung von Zugriffsberechtigungen, das Zwischenspeichern von Daten oder das Protokollieren von Aufrufen. Die Beziehung zum Zielobjekt wird zur Kompilierzeit festgelegt und unterliegt während der Laufzeit keiner Veränderung, weshalb sie als statisch bezeichnet wird. Dies impliziert, dass der statische Stellvertreter bei der Erstellung instanziiert wird und die Referenz auf das Zielobjekt während seiner gesamten Lebensdauer unverändert bleibt.

Ein dynamischer Stellvertreter, engl. Dynamic Proxy, kontrolliert den Zugriff auf ein Objekt, ohne das eigentliche Objekt zu modifizieren. Im Gegensatz zum statischen Stellvertreter wird der dynamische Stellvertreter zur Laufzeit erstellt und kann das Verhalten des Zielobjekts zur Laufzeit dynamisch modifizieren [10, 50]. Dies ermöglicht eine flexible Reaktion auf die jeweiligen Anforderungen oder Bedingungen sowie die Implementierung komplexer Logik. Dynamische

[18] Remote Method Invocation
[19] Enterprise Java Bean
[20] Java Persistence API

Stellvertreter finden häufig Anwendung in Umgebungen, in denen eine dynamische Interaktion mit Objekten erforderlich ist. Dies kann beispielsweise bei der Implementierung von entfernten Methodenaufrufen, der Erstellung von Platzhalterobjekten (Mock) für Unit-Tests oder der Implementierung von Abfang-Funktionalitäten in Frameworks der Fall sein.

Die Programmiersprache Java verfügt mit der Klasse java.lang.reflect.Proxy über einen dynamischen Stellvertreter. Mit dieser Klasse können Proxy-Objekte für beliebige Schnittstellen zur Laufzeit erzeugt werden, indem sie eine InvocationHandler-Implementierung verwendet, welche die Verarbeitung von Methodenaufrufen auf dem Proxy-Objekt übernimmt [43, 114].

Kapitel 3

Dynamische Mengen

Inhalt

3.1 Motivation **58**
3.2 Konzept **61**
3.2.1 Anwendungsintegration 61
3.2.2 Mitgliederverwaltung 63
3.2.3 Methodenaufruf 66
3.2.4 Dienstgüte 66
3.3 Methodenausführung **69**
3.3.1 Phasenmodell 69
3.3.2 Aufrufkontext 74
3.3.3 Nachbessernde Rückgabewerte 75
3.4 Programmierschnittstelle **77**
3.4.1 Konfiguration 77
3.4.2 Laufzeitumgebung 79
3.4.3 Deklaration und Instanziierung 80
3.4.4 Aggregation 81
3.4.5 Management 82
3.4.6 Erweiterung der Logik 84
3.4.7 Annotation API 88
3.5 Umsetzung **88**
3.5.1 Dynamischer Stellvertreter 90
3.5.2 Aufrufstrategie und -planer 91
3.5.3 Aggregation 91
3.6 Verwandte Arbeiten **92**
3.7 Diskussion **94**

3.1 Motivation

Programmierabstraktionen sind heute in der Informatik so allgegenwärtig, dass sie von Anwendungsentwicklern kaum noch wahrgenommen werden. In der Tat abstrahiert jedoch jede übergeordnete Programmiersprache von Maschinendetails und dem ausgeführten Maschinencode und bietet gleichzeitig effiziente Steuerungsstrukturen, die auf Syntaxelementen basieren, die natürlichen Sprachen ähneln.

Diesbezüglich lassen sich zahlreiche Beispiele anführen. Hochsprachen abstrahieren von Details der zugrunde liegenden Hardwarearchitektur und unterstützen die Erstellung plattformübergreifender Software. Arrays, Sequenzen und Wörterbücher stellen integrale Datenstrukturen vieler Programmiersprachen mit Kurznotationen zum Durchlaufen und Bearbeiten von Elementen dar. Des Weiteren werden mit Language Integrated Queries (LINQ) native SQL-ähnliche Funktionen hinzugefügt, um den Zugriff auf und die Verarbeitung von Daten unabhängig von ihrer Quelle (z. B. Arrays, Objektsammlungen, Datenbanken oder XML-Dokumente) zu vereinheitlichen. Dies erfolgt in Form von Abfragen von Anweisungen. Durch das Ausblenden bestimmter Implementierungsdetails reduzieren Programmierabstraktionen die Komplexität und ermöglichen es Entwicklern, sich besser auf die eigentliche Anwendungslogik zu konzentrieren.

Für Anwendungen in ubiquitären Umgebungen, die auf verteilten Ensembles kooperierender Geräte basieren, haben sich jedoch Programmierabstraktionen, die Entwickler gleichermaßen unterstützen noch nicht durchgesetzt. Eine wesentliche Herausforderung beim Ubiquitous Computing besteht insbesondere darin, die große Lücke zwischen Entwurfszeit und Laufzeit zu verringern. Um die Implementierung zur Entwurfszeit abzuschließen, müssen Entwickler bereits zur Laufzeit in der Lage sein, mit der Dynamik der Umgebung adäquat umzugehen. Diesbezüglich sind beispielsweise die Vorabfestlegung von Umgebungseinstellungen, die Implementierung diverser Ausführungsstrategien, die Vorhersehung und programmatische Behandlung möglicher Ausnahmen und Fehlerbedingungen zu nennen, obgleich zu diesem Zeitpunkt noch nicht einmal Klarheit darüber besteht, welche Dienste und Geräte zu einem späteren Zeitpunkt vorhanden sein werden. In Anbetracht der Komplexität der Anwendung ist ersichtlich, dass ein wesentlicher Unterschied besteht, ob die Anwendung ein, zehn oder sogar Hunderte von Geräten steuern muss.

In diesem Kapitel wird die Idee und die Funktionen dynamischer Mengen motiviert und aufgezeigt, wie Entwickler am meisten von dieser Abstraktion profitieren. Im Rahmen der Interaktion mit einer Menge gleichartiger, entfernter Objekte werden die folgenden Forschungsfragen erörtert: Es soll erörtert werden, auf welche Weise das fehlende Wissen über das Laufzeitverhalten von Entwicklern abstrahiert werden kann. Es gilt zu erforschen, wie die Art und Anzahl der zu bindenden Geräte verborgen werden können. Zudem ist zu untersuchen, welche Routineprozeduren und Fehlerbehandlungsstrategien abstrahiert werden können, d.h. in die Middleware ausgelagert werden sollten. Ferner ist zu be-

leuchten, welche Sprachkonstrukte sich zur Konfiguration dynamischer Mengen in welchen Phasen der Softwareentwicklung eignen. Schließlich ist zu erforschen, wie Entwickler mit Mengen interagieren und diese funktionell erweitern können.

Die Grundlage der Ausführungen dieses Kapitels bilden die eigenen Veröffentlichungen zu dynamischen Mengen für die Abstraktion von Objektgruppierungen und deren reflexive Anwendungsintegration [126], Schnittstellen und erweiterten Mengenfunktionalitäten [129].

Nachfolgend stellt Abschnitt 3.2 das Konzept der Programmierabstraktion vor. Der Abschnitt 3.3 präsentiert ein Phasenmodell zur Aufrufbehandlung entfernter Objekte und der Aggregation deren Rückgabewerte und Abschluss an den Aufrufer. Die Konfigurationsmöglichkeiten und die Schnittstellen für dynamische Mengen werden im anschließende Abschnitt 3.4 dargestellt und ausgewählte Aspekte der Implementierung in Abschnitt 3.5 beschrieben.

Beispiel

Die Charakteristika des IoT-Bereichs bedingen die Notwendigkeit und Entwicklungen von Middleware in einer verteilten Infrastruktur, die eine hohe Anzahl an Geräten mit unterschiedlichen Eigenschaften und einer begrenzten Ressourcenausstattung umfasst, finden spontane Interaktionen in einer dynamischen Netztopologie statt. Standortbewusstsein und Kontextualisierung durch Dienste sind essenziell, um die echtzeitfähigen Ergebnisse zu erstellen. Die Anforderungen an die Architektur beinhalten unter anderem Programmierabstraktionen mit definierten Schnittstellentypen zur Steigerung der Interoperabilität, welche servicebasiert Funktionserweiterungen ermöglicht. Kontextbewusstsein und Adaptivitätsstrategien erlauben die autonome Anpassung von Interaktionen an dynamische Umgebungen anhand von Richtlinien [135].

Die Assistenz in intelligenten Heimumgebungen [27] oder auch semi-öffentlichen Bereichen wie intelligenten Besprechungsräumen stellt einen vielversprechenden Anwendungsbereich für allgegenwärtige Systeme dar. Typenunterschiedliche Geräte beobachten Aspekte der Umgebung und Anwendungslogiken fusionieren die Daten, um anhand von Kontext und Konfiguration auf bestmögliche Assistenz zu entscheiden und ggf. die Umgebung durch Aktuatoren anzupassen.

Als motivierendes Beispiel kann ein intelligenter Besprechungsraum herangezogen, wie er auch als Labor an der Universität Rostock vorhanden ist. Um die Lichtverhältnisse für die Präsentation zu optimieren, könnten folgende Maßnahmen ergriffen werden: Das Ausschalten der Deckenbeleuchtung in der Nähe der vorgesehenen Projektionsflächen sowie eine Reduzierung der Beleuchtung in anderen Teilen des Raumes. Zusätzlich sollten an sonnigen Tagen die Jalousien (teilweise) geschlossen werden.

Listing 3.1 zeigt einen Auszug einer Schnittstellendeklaration zu einer Leuchte in der Programmiersprache Java. Die elementaren Methodensignaturen lassen Auf-

```
interface ILight {
   boolean isPowered();
   void    setPowered(boolean pow);
   boolean isDimmable();
   ...
}
```

Listing 3.1: Einfache Geschäfts-Schnittstelle

rufe über den Betriebszustand, dessen Änderung durch das An- und Ausschalten in Objektinstanzen implementieren.

Ein Anwendungsentwickler möchte möglicherweise nicht explizit jede einzelne Geräteinstanz oder -gruppe ansprechen, sondern Einschränkungen festlegen, die verdeutlichen, dass alle Geräte in der Lage sind, die Lichtverhältnisse zu verändern. Geeignete Geräte werden zur Laufzeit ausgewählt und dynamisch verwaltet, indem sie den Einschränkungen des bereitgestellten Mengenobjekts entsprechen.

Des Weiteren ist es erforderlich, dass eine Anwendungsgruppe die Funktionalitäten der bereits vorhandenen Geräte evaluiert. Ein einzelner Methodenaufruf von einer Anwendungskomponente an die Mengenreferenz, die auf jedes Mitgliedsgerät erweitert wird, muss individuell adaptiert werden, um die Gesamtaufgabe zu erfüllen. Ein Beispiel für eine zu lösende Aufgabe ist die Reduzierung der Beleuchtung in einem bestimmten Bereich auf 20%. Dabei ist zu berücksichtigen, dass einzelne Leuchten nicht dimmbar sind und lediglich ein- oder ausgeschaltet werden können. In diesem Zusammenhang stellt sich die Frage, welche Strategie geeignet ist. Eine Möglichkeit wäre die Ausschaltung der meisten Geräte in einer dynamischen Menge, während einige wenige eingeschaltet bleiben. Die Integration von Fensterjalousien des Raumes in diese Einstellung erfordert eine alternative Anpassung des Funktionsaufrufs für diese Gerätetypen, da sie bis zu einem gewissen Grad heruntergelassen werden müssen.

Die Umsetzung einer gestellten Aufgabe erfolgt durch die Erfassung der verfügbaren Objektvertreter von Geräten. welche mit ihren Zuständen und Fähigkeiten einen ortsbezogenen Kontext zu Laufzeit formen. Der individuelle Kontext dient dabei als Grundlage für Ausführungsstrategien einer Anwendung. Die adaptive Logik von Assistenzprogrammen für Smart-Home/Office-Umgebungen wird in dieser Arbeit nicht thematisiert. Die Programmierabstraktion der dynamischen Mengen kann domänenunabhängig Anwendung finden, wenn die Anzahl einer gleichen Schnittstelle implementierende Objektinstanzen zur Entwurfszeit unbekannt ist. Der Fokus der vorliegenden Arbeit liegt hier in der Anwendungsintegration und -konfiguration, der Mitgliederverwaltung, den Phasen eines Methodenaufrufes.

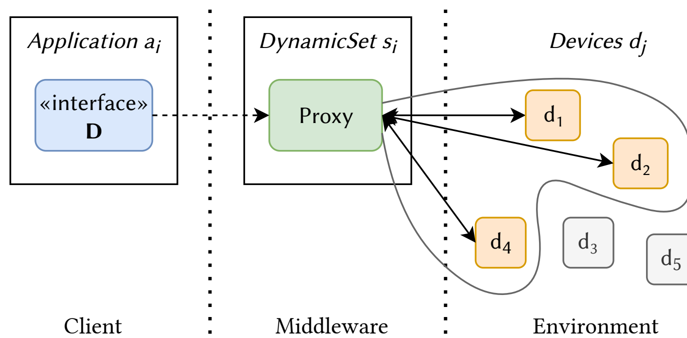


Abbildung 3.1: Konzept Programmierabstraktion Dynamische Mengen

3.2 Konzept

Die Interaktion mit einer variablen Anzahl von entfernten Geräten in verteilten Systemen ist von Natur aus komplex. Die Intention, Entwickler zu entlasten, impliziert, dass die Lösung dieser Aufgabe so einfach wie das Aufrufen von Methoden eines lokalen Objekts sein soll. Zu diesem Zweck bündeln dynamische Mengen, engl. dynamic sets, entfernte Objekte, die eine gemeinsame Schnittstelle implementieren. Die Menge sowie ihre Mitglieder werden durch einen einzelnen Stellvertreter dargestellt, der dieselbe Schnittstelle bietet. Durch das Aufrufen einer Methode auf dem Stellvertreter wird ein Aufruf repliziert und die entsprechende Methode auf den entfernten Objektinstanzen der Mitglieder der dynamischen Menge ausgeführt. Die zurückgegebene Werte der Objektinstanzen werden zu einem einzigen Ergebnis aggregiert. Somit werden Anwendungen, die dynamische Mengen nutzen, unabhängig von der Anzahl der entfernten Objekte, mit denen sie interagieren. Dies ermöglicht Entwicklern eine einheitliche Programmierung ihrer Anwendungen, unabhängig davon, ob sie ein oder hunderte Geräte steuern müssen. Abbildung 3.1 illustriert das vorgestellte Konzept schematisch.

3.2.1 Anwendungsintegration

Die imperative Programmierung ist ein etabliertes Paradigma in der Anwendungsentwicklung der Informatik. Im Rahmen dessen erstellen Programmierer im Quellcode Anweisungen entsprechend der Syntax der gewählten Programmiersprache und geben mit dieser explizit an, wie und in welcher Reihenfolge Aufgaben ausgeführt werden sollen. Zu diesem Zweck stehen diverse Konstrukte wie Speicherzuweisungen, arithmetische Operationen, Schleifen, Kontrollstrukturen und Prozeduren zur Verfügung.

Schnittstellen stellen in der Softwareentwicklung logische Berührungspunkte zwischen verschiedenen Prozessen oder Komponenten dar. Die Spezifikation einer

Schnittstelle dient als Vertrag zwischen dem Nutzer der Funktionalität und dem Anbieter einer Implementierung. Die funktionale Abstraktion von Schnittstellen wird durch Operationssignaturen bereitgestellt. Diese definieren *was* bereitgestellt wird, jedoch nicht *wie*. In der objektorientierten Programmierung werden Schnittstellen äquivalent wie Klassen umgesetzt, welche keine Attribute, sondern nur abstrakte Operationen (Methoden) deklarieren.

Für die Programmierabstraktion der dynamischen Mengen bilden Schnittstellen die Grundlage der Einbindung, da sie sich auf die Ansprechbarkeit mit einer gemeinsamen Syntax verlassen. Die zur Laufzeit zu bindenden Instanzen können zum Zeitpunkt des Entwurfs und der Umsetzung nicht bestimmt werden, da es aufgrund von Dynamiken in der Zusammensetzung des Ensembles sowie fehlender Zusicherungen in der Kommunikation in verteilten Systemen nicht möglich ist, die betreffenden Objekte zu identifizieren. Die Programmierabstraktion kann auf zwei Arten in eine Anwendung integriert werden:

Nutzung als Instanz. Die Verwendung dynamischer Mengen kann in einer Anwendung vollständig abstrahiert werden. Durch die Deklaration der Schnittstelle in einer Anwendung erhalten Entwickler den größten Abstraktionsgrad. Des Weiteren kann davon ausgegangen werden, dass der Anwendung die Verwendung dynamischer Mengen nicht bewusst ist.

Um die Deklaration der Schnittstelle in eine dynamische Menge zu überführen, ist die Angabe von Metadaten notwendig. Diese werden von der Ausführungsumgebung zur Bereitstellungszeit ausgelesen und in eine Konfiguration überführt. Die Laufzeitumgebung erstellt die notwendigen Datenstrukturen zur Verwaltung einer dynamischen Menge und injiziert in die Schnittstellendeklaration ein Stellvertreterobjekt. welches die Methodenaufrufe abfängt.

Nutzung als Menge. Einem Anwendungsentwickler, der sich der Tatsache bewusst ist, dass er mit der deklarierten Schnittstelle eine dynamische Menge bindet, wird die Möglichkeit offeriert, diese zu verwalten. Dies ist insbesondere erforderlich, um das Verhalten der Standardimplementierung an die eigenen Bedürfnisse anzupassen. Entwickler streben in der Regel danach, die Zusammenstellung der entfernten Objekte in einer dynamischen Menge sowie die Ausführungsstrategien für Methoden zu beeinflussen.

In Fortführung des Einführungsbeispiels, vgl. Abschnitt 3.1 zur Steuerung von Lichtquellen in ubiquitären Umgebungen, kann eine Anwendung (Klient) eine einzelne Lichtquelle steuern, d.h. eine Lampe ein- und ausschalten. Die Lampe implementiert die `ILight`-Schnittstelle, die im Voraus bekannt ist, durch Überprüfung des Quellcodes erkannt oder zur Laufzeit durch Reflexion abgeleitet werden kann. Die genannte Schnittstelle wird automatisch erweitert, indem vordefinierte Methoden hinzugefügt werden, um eine Reihe von Objekten des Typs Leuchte, d.h. `Light`-Instanzen, zur Laufzeit bequem zu verwalten. Beispielsweise können Methoden zum Registrieren von Beobachtern, engl. Listener, ergänzt werden,

so dass registrierte Abonnenten benachrichtigt werden, wenn Geräte der Menge beitreten oder diese verlassen. Der Stellvertreter, welcher die dynamische Menge von Leuchten repräsentiert, wird ebenfalls automatisch generiert. Er implementiert die kombinierte ILightSet-Schnittstelle, um sowohl die ursprünglichen Funktionen einer Leuchte ILight als auch die Funktionen zur Verwaltung der Menge IDynamicSet anzubieten.

3.2.2 Mitgliederverwaltung

Die Auswahl der Mengenmitglieder kann durch verschiedene Akteure in unterschiedlichen Phasen der Softwareentwicklung durch das Setzen von Kriterien über unterschiedliche Sprachkonstrukte erfolgen: Ein Anwendungsentwickler verwendet beispielsweise während der Implementierung imperative Funktionsaufrufe einer Erbauerklasse, um Rahmen der Bereitstellung, einen angepassten Deskriptor mit einer deklarativen Abfrage zu versehen, welche sich auf ein Attribut einer Klasse bezieht. Zur Laufzeit setzt ein Nutzer über Eingabe- oder Auswahl-Interaktionselemente einer grafischen Oberfläche neue Kriterien für eine Instanz, die vorhandene überschreiben. Die Verantwortung für die Zusammenstellung der dynamischen Menge und deren Aktualisierung wird vom Stellvertreter an die Middleware delegiert und dort über verfügbare Discovery-Mechanismen und Netzwerkadaptern übersetzt und an die Umgebung abgefragt.

Auswahlkriterien

In der Informatik werden Auswahlkriterien unter Zuhilfenahme unterschiedlicher Paradigmen und Konstrukte realisiert und evaluiert. Atomare Aussagen können Variablen, Konstanten oder prädikative Ausdrücke sein und Aussagen über Zustände, Ereignisse, Objekte oder Eigenschaften treffen. Ein logischer Ausdruck kann atomare Aussagen enthalten, die entweder wahr oder falsch sind, und in Kombination mit Operatoren komplexe Aussagen zu formulieren und zu analysieren. Beispiel für logische Operatoren sind Disjunktion (OR), Konjunktion (AND) und Negation (NOT).

Imperative Selektion. Die Definition von Auswahlkriterien oder Filtern in einer Weise, die explizit den Kontrollfluss des Programms angibt stellt imperative Selektion dar. er Ablauf der Entscheidungsfindung zur Auswahl von Elementen wird präzise definiert. Entscheidungen nach dem prozeduralen Paradigma können mittels bedingter Anweisungen (if-else) oder Schalteranweisungen (switch) getroffen werden. Es erfolgt eine explizite Angabe dazu, welcher Codeblock unter Berücksichtigung einer bestimmten Bedingung ausgeführt werden soll. Auswahl in der objektorientierten Programmierung kann durch Polymorphie und Vererbung getroffen werden. Die Implementierung einer Methode kann innerhalb einer Klasse auf unterschiedliche Weise erfolgen. Das System wählt automatisch die korrekte Implementierung basierend auf dem tatsächlichen Typ des Objekts

aus. In der funktionalen Programmierung können durch die Verwendung von Funktionen höherer Ordnung und Lambda-Ausdrücken Entscheidungen getroffen werden. Funktionen können als Argumente übergeben und basierend auf bestimmten Bedingungen angewendet werden.

Objekt-relational Abbilder (Object-relational Mapper, ORM) transformieren relationale Datenbankschema auf Objektklassen. Die Verwendung von Hilfsklassen erlaubt eine sprachlich nahe Interaktion mit den gewöhnlichen Objekten. Funktionen zum Abfragen von und Schreiben in Datenbanken werden unter Berücksichtigung der Dialekte des relationalen Datenbankmanagementsystems (RDBMS) in deklarative Sprachkonstrukte übersetzt. Die aus einem RDBMS erhaltenen Ergebnismengen werden jeweils auf die Objektklassen abgebildet, sodass eine weitere nahtlose Verwendung in der Anwendung möglich ist. ORM erlaubt Persistenzanbindungen und -interaktionen unter Verwendung der Programmiersprache der eigentlichen Anwendung, abstrahiert unterschiedliche Datenbankimplementierungen und damit verbundene Varianzen in der Abfragesprache und reduziert die Fehleranfälligkeit durch Überprüfung bereits zur Kompilierzeit. Als Beispiele sind aus dem Umfeld der Programmiersprache Java das quelloffene Framework *Hibernate*[1] oder die *Jakarta Persistence API*, welche Anwendungsserver implementieren, die der Java Enterprise Edition folgen. Für andere Programmiersprachen existieren vergleichbare Frameworks, beispielsweise im Python-Webframewerk *Django* oder *Symphony* für Webanwendungen, welche in PHP umgesetzt sind.

Deklarative Selektion. Ein Entwickler definiert die auszuwählenden Daten oder Elemente, wobei das System die Implementierungsdetails zur Erreichung dieser Auswahl übernimmt. In der Regel werden Abfragesprachen oder spezielle Syntax verwendet, um die Auswahlkriterien zu formulieren.

Die Structured Query Language (SQL) stellt eine Abfragesprache zur Verfügung, die für die Auswahl, Filterung und Manipulation von Datensätzen in relationalen Datenbanken und Managementsystemen eingesetzt werden kann. In Erweiterung bietet die Continuous Query Language (CQL) [6] für strombasierte Daten die Möglichkeiten zur Definition von Mengen als auch von Zeitfenstern. Cascading Style Sheets (CSS) dienen der Definition des Aussehens von HTML- und XML-Dokumenten. Die Formulierung von Regeln gestattet die Auswahl von Elementen basierend auf ihren Eigenschaften sowie die Zuweisung von Darstellungen. Reguläre Ausdrücke, engl. Regular Expressions (RegEx), stellen Musterbeschreibungen zur Verfügung, um Zeichenketten zu durchsuchen und zu manipulieren. Für die Abfragen auf XML-Daten wird XML Path Language (XPath) genutzt, um durch die Definition von Pfaden spezifische Elemente oder Attribute in einem XML-Dokument zu identifizieren und darauf zuzugreifen. Die Abfragesprache XQuery ermöglicht die Suche und Manipulation von XML-Daten.

[1] `https://hibernate.org/`

Selektion in dynamischen Mengen

Für die Auswahl der Mitglieder können unterschiedliche Selektionsparadigmen angewendet werden. Um die Unabhängigkeit gegenüber unterschiedlichen Plattformen und der verwendeten Programmiersprachen zu gewährleisten sowie die Serialisierung bei der Übertragung der Auswahlkriterien über ein verteiltes System zu ermöglichen, wird eine Indirektion durch eine deklarative Abfragesprache empfohlen. Die Zeichenkette wird von der Middleware zur Bereitstellungs- oder Laufzeit ausgelesen, zerlegt und mit der verwendeten Programmiersprache gegen Objektzustände evaluiert. Vor der Evaluierung der Auswahlkriterien gegenüber Objekt-Kandidaten und deren späterer Nutzung müssen diese zunächst entdeckt werden (engl. discovery). Sofern die Objekte nicht lokal vorgehalten sind, existieren Mechanismen, um diese in verteilten Systemen zu erreichen, vgl. Abschnitt 2.1.4.

Auswahldimensionen. Es können deklarative Auswahlbeschränkungen angegeben werden, die bestimmen, welche entfernten Objekte zur Menge hinzugefügt werden. Neben der Prüfung, ob ein entferntes Objekt die erforderlichen Schnittstellen implementiert, können Beschränkungen Prädikate enthalten den Zustand lesende, d.h. Getter-Methoden einer Schnittstelle auswerten, bspw. isPowered() == TRUE. Darüber hinaus können einfache Prädikate zu komplexeren Booleschen Ausdrücken kombiniert werden. Die Referenz auf ein entferntes Objekt wird nur dann als Mitglied einer dynamischen Menge hinzugefügt, wenn alle angegebenen Bedingungen erfüllt sind.

Alternativ besteht die Möglichkeit, Mengen auf Basis anderer bereits definierter dynamischer Mengen anzugeben. Durch Hinzufügen weiterer Einschränkungen zu einer vorhandenen Menge wird eine Teilmenge erstellt. Darüber hinaus ist es mithilfe von Mengenoperationen möglich, die Vereinigung, den Schnittpunkt oder das relative Komplement von zwei oder mehr Mengen zu erstellen. Auf diese Weise können vielseitige Mengenhierarchien gebildet werden, wobei vorhandene Mengendefinitionen wiederverwendet und nicht wiederholt dieselben Einschränkungen angegeben werden müssen.

Konfigurationsstufen. Die Definition oder Anpassung von Auswahlbeschränkungen für dynamische Elemente kann in verschiedenen Phasen erfolgen: zum Zeitpunkt des Entwurfs der Anwendung, bei der Bereitstellung der Anwendung sowie während der Laufzeit. Im Rahmen der Anwendungsentwicklung hat ein Entwickler die Möglichkeit, eine Entität als dynamische Menge zu kennzeichnen. Bei der Bereitstellung der Anwendung ist es häufig erforderlich, die Einstellungen an die tatsächliche Umgebung anzupassen. Zu diesem Zweck ist es möglich, eine Konfigurationsdatei bereitzustellen, die mit Anmerkungen versehene Auswahlbeschränkungen ändert oder überschreibt. Beim Start der Anwendung analysiert die Middleware Annotationen und Konfigurationen gleichermaßen, wobei letztere im Konfliktfall Vorrang haben. Auf die analysierten Einschränkungen kann auch

zur Laufzeit zugegriffen werden. Mithilfe der vom Stellvertreter implementierten erweiterten Verwaltungsschnittstelle können Auswahlkriterien über Methodenaufrufe überprüft und geändert werden, während die Anwendung ausgeführt wird. Die Middleware sucht regelmäßig nach neuen entfernten Objekten, die die definierten Einschränkungen erfüllen, sowie nach verwaisten Referenzen. Proxy-Objekte abhängiger Gruppen werden über relevante Änderungen informiert.

3.2.3 Methodenaufruf

Das Aufrufen einer Methode an einer dynamischen Menge, genauer an ihrem Stellvertreter, welcher die Menge entfernter Objekte darstellt, wird für jedes Mitglied die entsprechende Methode aufgerufen.

Die Middleware hat die Aufgabe, Elemente der Methodenaufrufe entsprechend der Möglichkeiten des Protokolls zu strukturieren, zu serialisieren, und zu jedem entfernten Objekt zu transportieren, die Methode dort tatsächlich auszuführen und die Aufrufergebnisse auf vergleichbarer Weise zurückzutransportieren. Darüber hinaus ist es auch erforderlich, diese Ergebnisse zu einem einzigen Rückgabewert zusammenzufassen, um der ursprünglichen Methodensignatur zu entsprechen.

Obwohl die Middleware Standardaggregationen bereitstellt, beispielsweise Durchschnitt, Mehrheit für numerische Typen sowie Konjunktion für boolesche Werte, welche typisch sichere Ergebnisse liefern, erfüllen diese Ergebnisse möglicherweise nicht den Zweck der Anwendung. In diesem Fall können Entwickler über Metadaten angeben, welche Aggregation verwendet werden soll. Die Middleware bietet eine umfassende Bibliothek mit standardisierten Aggregationsfunktionen aus dem Bereich der deskriptiven Statistik sowie Funktionen zur Verarbeitung von Textdaten. Alternativ können Entwickler auch eigene benutzerdefinierte Aggregationsfunktionen schreiben, insbesondere für komplexere Rückgabetypen wie Objekte.

3.2.4 Dienstgüte

Der Begriff der Dienstgüteaspekte, engl. Quality of Service (QoS), bezeichnet das Serviceniveau einer Anwendung oder eines Systems. Dazu zählen beispielsweise Übertragungsmetriken in verteilten Systemen, vgl. Abschnitt 2.3.2.

QoS für dynamisch Mengen lassen sich nicht ausschließlich aus den bekannten Gebieten wie dem Paket-Routing in Netzwerken herleiten. Die Wahrnehmung der Dienstgüte ist hier nicht durch menschliche Wahrnehmung von Endnutzern gegeben, sondern wird vom Anwendungsentwickler definiert und evaluiert, der die Programmierabstraktion in seiner Anwendung verwendet. Für dynamische Mengen werden Metriken aus anderen Anwendungsgebieten abgeleitet, mit denen die Leistung angefordert wird sowie die erzielte Dienstgüte abrufbar ist.

Die Behandlung von Ausnahmen bildet einen weiteren Aspekt, der im Rahmen der Dienstgüteaspekte zu berücksichtigen ist. Dabei stellt sich die Frage, ob ein Methodenaufruf für die dynamische Gruppe fehlschlagen sollte, wenn der Aufruf für eines oder mehrere ihrer Mitglieder fehlschlägt, oder ob bestimmte Fehler vernachlässigt werden können. Im Folgenden werden alternative Ausführungsstrategien, verschiedene Möglichkeiten zur Fehlerbehandlung und Dienstgüteaspekte erörtert. Analog zu Auswahlbeschränkungen können alle genannten Strategien sowie Aggregationsfunktionen sowohl zur Entwurfszeit konfiguriert als auch bei Bereitstellung der Anwendung oder zur Laufzeit dynamisch angepasst werden.

Die Wahl unterschiedlicher Ausführungsstrategien kann eine Veränderung der Semantik von Methodenaufrufen zur Folge haben. Methoden werden standardmäßig auf den festgelegten Mitgliedern asynchron aufgerufen und parallel ausgeführt, wobei der Kontrollfluss der aufrufenden Anwendung blockiert wird, bis alle Ergebnisse erfasst und zusammengefasst sind. Alternativ können Methoden für Mitglieder in einer bestimmten Reihenfolge nacheinander synchron aufgerufen werden. Oder es ist sogar möglich, einen Anycast-Aufruf zu realisieren, d.h. die Methode nur für ein einzelnes Mitglied der dynamischen Menge aufzurufen, während bspw. ein Rundlaufverfahren für den Lastausgleich verwendet wird.

Ein strukturierte Kontrollzyklus wie MAPE-K (vgl. Abschnitt 2.3) fungiert als Blaupause für die Bearbeitung der Methodenaufrufe durch die Programmierabstraktion. Die Selektion und das Aufrufverhalten von entfernten Geräten können auf Basis von Metriken, welche die vorherigen Aufrufe betreffen, evaluiert werden. Für die Bewertung der Dienstgüte sind verschiedene Metriken bekannt, die auch auf die Aufrufstrategie von dynamischen Mengen angepasst Anwendung finden:

Verzögerung (Latenz). Die Zeitspanne, die vergeht, bis eine Reaktion auf ein Ereignis in einem Computernetzwerk sichtbar wird, wird als zeitliche Differenz zwischen dem Senden eines Datenpakets durch den Sender und dessen Eintreffen beim Empfänger bezeichnet. [118] Die Paketumlaufzeit, engl. Round-Trip-Time (RTT), umfasst die Laufzeit zwischen Routingstationen, die Verarbeitungszeit auf diesen sowie die Verbreitung auf Start- und Zielhost. Die halbierte RTT ergibt die approximierte Latenz zwischen zwei Hosts.

Für dynamische Mengen ist die gesamte Paketlaufzeit von Relevanz, da im Anfrage-/Antwort-Interaktionsmuster auf eine gesendete Anfrage die erhalte Antwort mit dem Rückgabewert der aufgerufenen Prozedur relevant ist.

Durchsatz (Bandbreite). Der Durchsatz bezeichnet die maximal mögliche Anzahl an Bits, die in einem Zeitintervall über die beobachtete Stelle des Netzwerks übertragen werden können. Die Durchsatzrate ist eine physikalische Eigenschaft des Übertragungsmediums, welche bspw. durch dessen Aufbau, Dicke und Länge und Material bestimmt ist [168].

Ein entfernter Methodenaufruf über dynamische Mengen misst im Anfrage-/Antwort-Interaktionsmuster die Übertragungsgeschwindkeiten der Aufruf- und Antwortdaten. Während Aufrufparameter bzw. Rückgabewerte aus primitiven Datentypen einen überschaubare Übertragungslast suggerieren, können komplexe Datenstrukturen in der Übertragung signifikante Netzlast erzeugen. Die Netzlast wird entsprechend der Anzahl der Mitglieder einer dynamischen Menge, die angefragt werden, faktorisiert, je nachdem, welches Interaktionsmuster gewählt wurde.

Aufrufzeit. Der Zeitintervall nachdem am Stellvertreter einer dynamischen Menge ein Methodenaufruf empfangen wurde, bis zur Rückgabe eines aggregierten Ergebniswertes, bestimmt die maximale Ausführungszeit (Timeout) bis zum Abbruch. Ohne Angabe kann der Aufruf ohne zeitliche Einschränkung dauern bis der Aufruf durch andere Dienstgüte-Parameter als erfolgreich durchgeführt oder endgültig gescheitert behandelt wurde.

Aufrufdichte. In Übereinstimmung mit den definierten Auswahlkriterien kann eine dynamische Menge aus einer signifikant großen Anzahl von entfernten Objekten bestehen. Bei einem Methodenaufruf ist es nicht immer erforderlich, mit allen Mitgliedern zu interagieren. Die Ausführung auf einer Teilmenge kann eine ausreichende Option sein, um ein vergleichbar stabiles Ergebnis nach der Aggregation zu erhalten und gleichzeitig Reserven für alternative Ausführungsstrategien bereitzustellen.

Ergebnisdichte. In verteilten Systemen können Unterbrechungen der Kommunikationskanäle auftreten, ebenso wie eine Fehlfunktion des entfernten Dienstes. Infolgedessen bleibt erwartete Antwort vom entfernten Objekt aus oder es wird mit einem Fehler geantwortet. Mit diesem Parameter kann konfiguriert werden, wie viele valide Antworten – absolut und/oder prozentual – mindestens erforderlich sind, um die Ergebnismenge in einen verlässlichen Aggregationswert zu überführen.

Leistungsfähigkeit. Der Parameter der Leistungsfähigkeit wird anhand von Statistiken bereits durchgeführter Methodenaufrufe mit dem entfernten Objekt definiert, wodurch sich eine Verlässlichkeit hinsichtlich der geplanten Interaktion ableiten lässt. Die Leistungsfähigkeit basiert auf einer Vielzahl von Dienstgüteaspekten. Die Fehlerrate ergibt sich aus dem Antwortverhalten früherer Methodenaufrufe während der Laufzeit. So entsteht die Fehlerrate aus dem Antwortverhalten früherer Methodenaufrufe während Laufzeitstatistiken auf der Basis gemessener Latenzen auf zukünftige Verzögerungen für erwartete Antwortzeiten schließen lassen. Ergebnisstatistiken lassen mit Glättungsoperationen auf älteren Rückgabewerten auftretende Ausreißer zu identifizieren.

3.3 Methodenausführung

Die Bearbeitung der Methodenausführungen an mehreren entfernten Objekten als Mitglieder einer dynamischen Menge soll in einer strukturierten Form erfolgen, um den Anforderungen verschiedener Konfigurationsaspekte gerecht zu werden. Dabei kann eine Konfiguration entweder global durch die Laufzeitumgebung oder lokal durch den Anwendungs- oder Aufrufkontext gesetzt werden.

3.3.1 Phasenmodell

Das folgend vorgestellte Phasenmodell nahm das Lebenszykluskonzept zur Dienstgüteunterstützung für Webservices als Vorlage [12]. Es wurde für die Anwendung auf dynamische Mengen entwickelt [166]. Das Modell unterteilt sich in mehrere Stufen beginnend vom eingehenden Methodenaufruf von der Anwendung am Stellvertreter-Objekt einer dynamischen Menge bis zur Rückgabe eines aggregierten Ergebniswertes.

Für eine Phase vorgesehene Dienstgüte-Aspekte werden aus den Konfigurationen ermittelt. Zusätzlich existieren globale Konfigurationsaspekte, die über mehrere Phasen eines Aufrufs überwacht werden und übergeordnete Entscheidungen treffen. Ein Beispiel für aufruf-globale Konfiguration ist die maximale Antwortzeit, die eine Anwendung einer Middleware einräumt, bis ein Rückgabewert erwartet wird. Jede Phase terminiert mit einem internen Ergebnis, welche von der anschließenden Phase konsumiert wird und in deren Verarbeitung einfließt. Die Übergänge zwischen den Phasen sind im nachfolgenden Modell (Abb. 3.2) dargestellt, womit es auch möglich ist adaptiv auf die den aktuellen Ausführungszustand reagieren zu können, um Verbesserungen im Rahmen der zur Verfügung stehenden zeitlichen und anzufragenden Ressourcen zu erreichen. Die Phasen des Modells werden nachfolgend jeweils mit Eingangswerten, Verarbeitungsschritten sowie Phasenergebnissen vorstellt.

Auswahl

Aus allen Mitgliedern einer dynamischen Menge wird eine Teilmenge erstellt, an deren entfernten Objekten der Methodenaufruf ausgeführt wird. Ziel ist es die Mitglieder auszuwählen, die mit hoher Wahrscheinlichkeit die an die dynamische Menge gestellten Dienstgüteaspekte erfüllen können. Die Bewertung der Qualität beruht hier auf statistischen Daten vergangener Aufrufe oder erfolgt bekannten Wahlalgorithmen. Die folgenden Aspekte beeinflussen die Auswahl:

Stichprobe. Eine Stichprobe betrachtet nicht die Gesamtheit, sondern eine Teilmenge, aus der auf die Charakteristik der Grundmenge schließen lässt. Im Kontext dynamischer Mengen ist hierzu die Teilauswahl von Mitgliedern einer

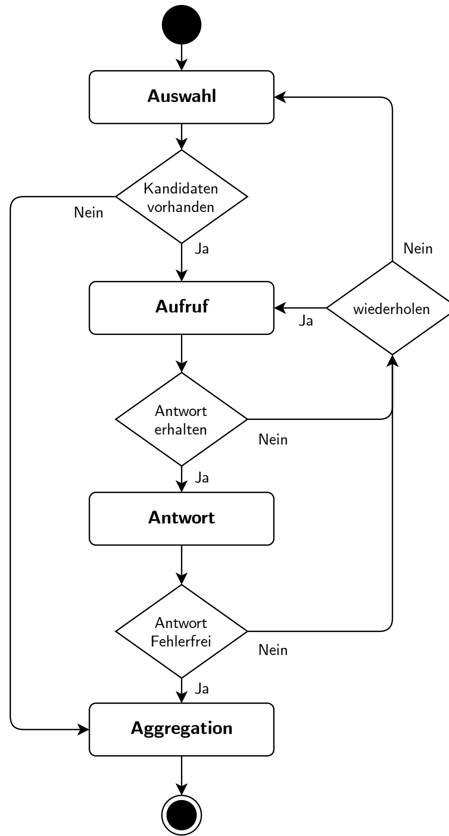


Abbildung 3.2: Phasenmodell für die Methodenausführung in dynamischen Mengen

Menge gemeint, um unnötigen Mehraufwand zu vermeiden. Gründe zum Abschluss einer Stichprobe sind bspw. wenn eine zusätzliche Mitgliederauswahl das Aggregationsergebnis nicht mehr signifikant beeinflusst oder davon ausgegangen wird, dass Mitgliederkandidaten nicht das responsive und valide Verhalten aufbringen können, was den Gesamtaufruf verzögern kann.

Bei Mengen mit einer sehr großen Anzahl von Mitgliedern ist die Reduktion auf eine Teilmenge sinnvoll, um mit deren erwarteten Antwortverhalten und Rückgabewerten die gestellten Dienstgüteaspekte zu erfüllen. Die Stichprobengröße wird relativ zur Anzahl der aktuellen Anzahl der Mitglieder einer dynamischen Menge prozentual oder absolut angegeben. Parameter stellen die Anforderungen der minimalen oder maximalen Stichprobengröße, die erreicht werden soll, um diesen Dienstgüteaspekt zu erfüllen.

Sortierung. Die Mitglieder einer dynamischen Menge werden vor der Auswahl anhand von Kriterien bewertet und absteigend sortiert. Eine hohe Priorisierung macht das Mitglied zu einem geeigneten Kandidaten mit dessen Auswahl die Dienstgüteaspekte erreicht werden können. Mögliche Strategien zur Priorisierung von Elementen können sein:

- Zufall: Im einfachsten Fall wird jedem Mitglied einer Menge zufällig eine Priorität zugewiesen.
- Rundlaufverfahren: Aufbauend auf einer Zufallspriorisierung wird beim Round-Robin-Verfahren [168] nach erfolgter Auswahl das Mengenmitglied markiert. Bei einer erneuten Auswahliteration werden nur Mengenmitglieder ohne Markierung berücksichtigt. Wurden alle Mitglieder ausgewählt – und markiert – werden die Markierungen aller Mengenmitglieder gelöscht und der ursprüngliche Zustand ist wieder hergestellt.
- Mitgliederattribute: Eigenschaften der Objekte bilden Gruppierungsgrundlage, wonach bspw. Sensoren an unterschiedlichen Standorten oder von verschiedenen Herstellern nacheinander angefragt werden.
- Aufrufhistorie: Eine oder die Kombination mehrerer Merkmale aus den statischen Daten von früheren Aufrufen an das entfernte Objekt dienen der Priorisierung des Mengenmitglieds. Je nach Umsetzung der Auswahlstrategie können Übertragungsstatistiken wie Erreichbarkeit, Latenz, Antwortverhalten bewertet werden.

Überbuchung. Zum Erreichen einer Ergebnisdichte auf Basis der Kardinalität der Grundmenge oder einer Stichprobe kann unter Berücksichtigung von statistischen Aufrufdaten eine Mehrauswahl umgesetzt werden. Zeigen historische Daten bei den ausgewählten Aufrufkandidaten bspw. unzureichendes Antwortverhalten, so kann bereits in der aktuellen Auswahlphase eine zusätzliche Aufrufdichte zur beabsichtigten Ergebnisdichte beitragen, fehlerhafte oder ausbleibende Antworten nicht durch erneute Auswahl- und Aufrufphasen kompensieren zu müssen.

Am Ende der Auswahlphase ist eine Teilmenge aus der Grundmenge aller Mitglieder der dynamischen Menge erstellt, die in der folgenden Phase für Methodenaufrufe an den entfernten Objekten vorgesehen ist. Ist die Teilmenge leer, d.h. es wurde keine Aufrufkandidaten ausgewählt, wird die Aggregationsphase aufgerufen.

Aufruf

Die Aufrufphase erhält eine Menge mit priorisierten Elementen aus der Grundmenge der Mitglieder einer dynamischen Menge aus der Auswahlphase. In eine Warteschlange überführt ist die Phase für die Methodenaufrufe an den assoziierten entfernten Objekten zuständig. Entsprechend des jeweils ausgewählten

Transportprotokolls werden die Werte der Methodensignatur und einen entsprechenden Netzwerk-Klienten übergeben.

Parallelisierung. Der Grad der Parallelisierung gibt an, wie viele Methodenaufrufe an entfernten Objekten zur gleichen Zeit durchgeführt werden. Zwischen den Extrema von der zeitgleichen Verarbeitung eines oder aller Aufrufkandidaten stellen Abstufungen Mischformen der Parallelität dar:

- Sequenziell: nur ein Aufruf wird zur gleichen Zeit bearbeitet. Ist ein Aufruf abgeschlossen, wird der Nächste mit nachfolgender Priorität angegangen.
- Parallel: alle Elemente aus der Warteschlange werden zur gleichen Zeit gestartet.
- Blockweise: Eine Mischform aus den beiden vorgenannten Punkte teilt die Elemente der Warteschlange in Blöcke auf. Die Blöcke werden sequenziell abgearbeitet, der Aufruf innerhalb der Blöcke erfolgt jedoch parallel.

Die Entscheidung für Parallelität und deren Ausnutzung ist durch unterschiedliche Bedingungen motiviert. Eine kurz gesetzte maximale Gesamtausführungszeit eines Methodenaufrufs an einer dynamischen Menge bedingt eine schnellere unabhängige Bearbeitung der Auswahl, was durch Parallelität ermöglicht wird. Bei sequenzieller Abarbeitung der Warteschlange kann ein langlaufender, ggf. nicht antwortender Methodenaufruf, anteilig viel Zeit in Anspruch nehmen, was den Dienstgüteaspekt zur maximalen Bearbeitung des Methodenaufrufs an der dynamischen Menge gefährdet. Dem gegenüber stehen bei einer hohen Parallelität höhere Anforderungen an die Hardware-Ressourcen sowie Verwaltung von Nebenläufigkeiten mehrerer Kommunikationskanäle.

Wertegültigkeit. Zum Erhalt des beabsichtigten Rückgabewertes eines entfernten Methodenaufrufes stehen zwei grundsätzliche Kommunikationsparadigmen zur Auswahl: In der anfrageorientierten Kommunikation wird die Methodenausführung angefragt (Pull) und der Rückgabewert zurückgegeben. In der aktualisierungsorientierten Kommunikation wird der Rückgabewert an zuvor registrierte Interessenten geschoben (Push).

Aus früheren Methodenaufrufen kann der letzte Rückgabewert mit Ankunftszeit lokal beim empfangenden Klienten in einem Cache zwischengespeichert werden. Bei folgenden Anfragen von einem Mengen-Stellvertreter auf dem Knoten wird der lokal zwischengespeicherte Wert zurückgegeben und der entfernte Methodenaufruf über das Netzwerk übersprungen, wenn die konfigurierte Wertegültigkeit größer als die Differenz zwischen Ankunfts- und aktueller Zeit ist. Hierbei wird der Eintrag aus Datenstruktur der Aufrufkandidaten entfernt und der Wert der Datenstruktur für die spätere Aggregationsphase hinzugefügt.

Am Ende dieser Phase wurden aus der Menge der Aufrufkandidaten eine Teilmenge zum Aufruf gebracht und jede Anfrage mit Zeitpunkt und Kennung des

entfernten Objektes in ein Korrelationsregister eingetragen. Diese temporäre Datenhaltung dient als Verbindung zur nachfolgenden Antwortphase, in der eingehende Antworten den Anfragen zugeordnet werden. Existieren zeit-gültige zwischengespeicherte Rückgabewerte aus früheren Aufrufen, wird die Anfrage über das Netzwerk übersprungen und der Wert für die Aggregation bereitgestellt.

Aufrufgültigkeit. Netzwerkfehler oder Verarbeitungsfehler am Knoten des entfernten Objektes können die Übertragung der Antwort des entfernten Methodenaufrufes verzögern oder diese kann komplett ausbleiben. Hierdurch können Ressourcen wie Einträge in Datenstrukturen verwaisen bzw. Verbindungsressourcen unnötig belegt und der Kontrollfluss im Phasenmodell ausgebremst werden. Es kann eine maximale Aufrufzeit definiert werden, um dies zu verhindern. Bei Überschreiten der Zeitspanne wird der Aufruf terminiert, d.h. der Eintrag aus der aktuellen Ausführungsmenge entfernt, die Statistik mit einem fehlgeschlagenen Aufrufversuch aktualisiert und der Eintrag aus dem Korrelationsregister gelöscht.

Am Ende der Phase wurden die ausgewählten Aufrufkandidaten zur Ausführung gebracht. Existieren zeitgültige Werte im Zwischenspeicher aus früheren erfolgreichen Anfragen, werden diese in die Datenstruktur der Antwortmenge für den Aufruf hinzugefügt und der Eintrag aus der laufen Aufrufmenge entfernt.

Erzeugt ein Methodenaufruf während der Aufrufphase bereits einen Laufzeit-, Übertragungsfehler oder antwortet das entfernte Objekt mit einem Fehler, kann entsprechend der aufrufglobalen Konfiguration ein erneuter Aufruf am gleichen Objekt gestartet werden oder das Element als fehlerhaft entfernt und für eine wiederholte Auswahlphase ein zusätzlicher Ersatzkandidat angefordert werden.

Antwort

Die Antwortphase überwacht die zur Ausführung gebrachten Methodenaufrufe und steuert entsprechend der Rückgaben: Antworten von entfernten Objektaufrufen werden empfangen, der Eintrag im Korrelationsregister gelöscht und der Aufrufstatistik des entfernten Objektes ein erfolgreicher Methodenaufruf zusammen mit der Umlaufzeit, d.h. das Zeitintervall vom Absenden der Anfrage bis Empfang der Antwort, hinzugefügt Abschließend wird der empfangene Rückgabewert als letztempfangener in den lokalen Cache mit der Ankunftszeit zwischengespeichert sowie der aufruf-lokalen Aggregationsmenge hinzugefügt, insofern die Aggregationsphase für den Aufruf noch nicht ausgeführt wurde.

Werden Fehler als Antwort zurückgegeben oder bleiben Antworten innerhalb des Zeitfensters aus, wird die Statistik um einen fehlgeschlagenen Methodenaufruf aktualisiert. Hier kann die Ausführungslogik nachsteuern und den Kontrollfluss adaptiv erneut auf vorhergehende Phasen richten, um fehlgeschlagene Aufrufe zu kompensieren und die beabsichtigte Anzahl an gültigen Antworten zu erhalten:

Zum einen kann das gleiche Objekt in einer weiteren Aufrufphase erneut angefragt werden, um ggf. temporäre Fehler auszugleichen bzw. durch ein erweitertes Zeitbudget Netzwerk- oder Verarbeitungslast auf der Gegenseite zu berücksichtigen. Alternativ kann auch eine weiteren Auswahlphase ein Ersatz aus der verbleibenden Grundmenge der Mengenmitglieder ausgewählt werden, welches noch nicht zum Aufruf gebracht wurde.

Am Ende der Phase sind valide Rückgabewerte der Aggregation bereitgestellt und können dort (vor)verarbeitet werden, während für ausgebliebene oder fehlerhafte Antworten Kompensationsstrategien durch wiederholten Aufruf oder Ersatzmitglieder versucht werden. Statistiken zum Aufrufverhalten und Zwischenspeicher für lokal vorgehaltene Rückgabewerte zum entfernten Methodenaufruf wurden aktualisiert.

Aggregation

In der der finalen Phase des Methodenaufrufs werden die Rückgabewerte erfolgreicher entfernter Methodenaufrufe durch die konfigurierte Aggregationsfunktion zu einem zusammengefasst.

Wenn jedoch bisher genügend gültige Ergebnisse ohne Ausnahmen eingegangen sind, kann ihre Aggregation bereits zu einem Rückgabewert von ausreichender Qualität führen. Beendet der alle Phasen überspannende Aufrufkontext die Aggregation in diesem Zustand, werden eine fehlerhafte Ausführung in Form von Sonderwerten oder Ausnahmeerscheinungen kommuniziert. Daher können Dienstgüte-Strategien die Mindestanzahl oder den -anteil von Mengenmitgliedern der für eine Aggregation erforderlichen Ergebnisse angeben oder die Wartezeit auf Ergebnisse begrenzen.

Bei Terminieren des Phasenmodells aus der finalen Aggregationsphase steht eine Rückgabewert entsprechend der Methodensignatur bereit, wie er aus der Anwendung initial aufgerufen wurde und kann an diese übergeben werden.

3.3.2 Aufrufkontext

Das im vorherigen Abschnitt vorgestellte Phasenmodell steht in Verbindung mit Datenstrukturen, die Rückgabewerte und Statistiken vorheriger Methodenaufrufe enthalten sowie Auswahl- und Korrelations-Einträge verwalten. Ziel ist es anhand verschiedener Konfigurationen und Statistiken optimierte Methodenausführungen umzusetzen.

Eine Anwendung kann mehrere Schnittstellen mit der Programmierabstraktion dynamischer Mengen deklarieren und zur Laufzeit unterschiedliche Methodenaufrufe über die Instanzen hinweg parallel ausführen. Des Weiteren können je nach Implementierung mehrere Anwendungen in einer Laufzeitumgebung bereitgestellt werden. Ein entferntes Objekt kann Mitglied in einer oder von mehreren

dynamischen Mengen sein. Der Methodenaufruf eines entfernten Objekts darf aus lokaler Sicht nicht exklusiv durch eine dynamische Menge verwaltet werden; bspw. könnten Statistiken unterschiedlicher dynamischer Mengen das Auswahlverhalten des gleichen entfernten Objektes unterschiedlich bewerten oder Ankunftszeiten letzter Methodenaufrufe nicht vollständig sichtbar sein.

Eine Kontexthierarchie schafft Struktur zur Datenhaltung und relevanten Sichtbarkeiten, um Kriterien im Auswahlverfahren zu erhalten und Bewertungen zur Ausführung zurückzugeben. Der hierarchische Ansatz gestattet Kontextobjekte mehrfach auf verschiedenen Ebenen anderen Sichtbarkeiten zu definieren [166]:

Global. Ein globaler Kontext existiert in einer Laufzeitumgebung nur einmalig und steht dort der Auswahl und Aufrufen aller dynamischen Mengen zur Verfügung. In dem Kontext werden Aufrufstatistiken aller bekannten entfernten Objekte dauerhaft geführt, unabhängig ihrer Zugehörigkeit zu temporär keiner, einer oder mehrerer dynamischen Mengen. Statistikeinträge sind Ergebnisse von Aufrufen, wie Zeitpunkt und Dauer der Antwort auf eine Anfrage (Rundlaufzeit), Datenübertragungsrate sowie Fehlerbeschreibungen oder ausgebliebene Antworten. Aufgrund einer Vielzahl von entfernten Objekten ist es zur Ressourceneffizienz empfehlenswert Einträge regelmäßig auf eine definierte Anzahl oder Zeitfenster zu reduzieren, d.h. Rohdaten zu aggregieren und anschließend zu löschen.

Schnittstelle. Der Kontext kann von allen Auswahlverfahren des gleichen Typs verwendet werden, d.h. Aufrufe an dynamischen Mengen, welche die gleiche Schnittstelle implementieren. Für eine Lastverteilung bspw. durch das Rundumlauf-Auswahlverfahren sind Daten der letzten Aufrufe über dynamische Mengen hinweg relevant. Kontextobjekte werden nur angelegt, wenn sie durch bspw. das Auswahlverfahren einer dynamischen Menge angefordert werden.

Instanz. Der Kontext auf kleinster Ebene sammelt Statistiken über entfernte Objekte, ausgelöst durch einen Methodenaufruf einer dynamischen Menge. Die Kapselung lässt keinen Zugriff auf die Daten von anderen dynamischen Mengen zu. Kontextobjekte werden explizit durch den Stellvertreter einer dynamischen Menge initiiert und terminieren bei Ende dessen Lebenszyklus.

3.3.3 Nachbessernde Rückgabewerte

Die Aufrufstrategie im vorgestellten Phasenmodell, vgl. Abschnitt 3.3.1, arbeitet in der Standardausführung blockierend vom Methodenaufruf am dynamischen Stellvertreter bis zur Rückgabe des Aggregationswertes. Wird eine anderer Verarbeitungsfluss gewünscht, bedarf es zusätzlicher Konstrukte: Die Ausführung in einem separaten Thread ermöglicht dem Aufrufer parallel weiterarbeiten. Zu

diesem Zweck wird ein erweitertes Future-Objekt zurückgegeben, mit dem der Aufrufer den aktuellen Status des dynamischen Aufrufs beobachten kann.

Die Programmierkonstrukte Promise und Future sind Konzepte, die in der asynchronen Programmierung verwendet werden, um die Ausführung von Aufgaben zu modellieren, die in der Zukunft abgeschlossen werden. Ein Promise ist ein Mechanismus, ähnlich einer Schnittstelle, der verwendet wird, um einen zukünftigen Wert zu produzieren. Ein Future repräsentiert eine zukünftige Ergebnis- oder Fehlermeldung einer asynchronen Berechnung. Das bedeutet, dass eine Funktion, die einen Future zurückgibt, sofort zurückkehrt, ohne auf das Ergebnis der Berechnung zu warten. Stattdessen kann der Aufrufer des Futures das Ergebnis später abrufen, wenn es verfügbar ist [90].

Future und darauf das aufbauende *Completable Future* sind Konzepte und Programmierkonstrukte, die in verschiedenen Programmiersprachen und Bibliotheken implementiert sind. Ein Future wird normalerweise von einem Executor (einem Thread-Pool) erstellt, der eine asynchrone Aufgabe ausführt. Dieser startet die Aufgabe und gibt sofort ein Future zurück, welcher das Ergebnis oder den Fehler der Aufgabe repräsentiert. Um das Ergebnis eines Future abzurufen, kann die Methode get() aufgerufen werden. Diese Methode blockiert, bis das Ergebnis verfügbar ist. Wenn das Ergebnis noch nicht verfügbar ist, blockiert get() den aufrufenden Thread, bis das Ergebnis bereit ist. Um zu überprüfen, ob ein Future abgeschlossen ist, kann die Methode isDone() aufgerufen werden, die einen Wahrheitswert zurückgibt. Zwischen verschiedenen Futures können Abhängigkeiten aufgebaut werden. Dies bedeutet, dass Aktionen nur dann ausgeführt werden, wenn vorangehende mit bestimmten Ergebnissen beendet werden.

In Java wurden Futures durch die Einführung des java.util.concurrent.Future-Interfaces in Version 5, java.util.concurrent.CompletableFuture als Erweiterung und Implementierung in Version 8 eingeführt und bietet zusätzliche Funktionen für die asynchrone Programmierung [177]. In JavaScript und Node.js gibt es die Promises API, die ähnliche Funktionen wie Futures bietet. Promises ermöglichen es, asynchrone Operationen zu modellieren und das Ergebnis oder den Fehler einer Operation zu behandeln, sobald sie abgeschlossen ist. In Python gibt es die concurrent.futures-Bibliothek, die eine Implementierung von Futures für asynchrone Programmierung bietet. Diese Bibliothek stellt die Future-Klasse bereit, die ähnliche Funktionen wie das java.util.concurrent.Future-Interface in Java bietet. In C# bietet der System.Threading.Tasks-Namensraum die Klasse, die eine Implementierung von Futures und asynchronen Aufgaben in .NET darstellt. Diese Klasse ermöglicht es, asynchrone Operationen zu starten und das Ergebnis oder den Fehler einer Operation zu behandeln, sobald sie abgeschlossen ist.

In der Aufrufausführung dynamischer Mengen erfolgt Ausführung in zwei Aktionen: i) der eigentliche Aufruf des entfernten Objektes sowie ii) die Verarbeitung der Antwort. Dabei ist die Verarbeitung der Antwort von der vorherigen Aktion abhängig. Das zeitgesteuerte Ereignis des ScheduledExecutorService kann lediglich den Abschluss der Aufrufaktion feststellen und die nachfolgende Ver-

arbeitung ignorieren. Dies gewährleistet eine korrekte Entscheidung hinsichtlich der Terminierung des Gesamtaufrufes und Rückgabe eines aggregierten Wertes oder einer Fehlermeldung [166].

3.4 Programmierschnittstelle

Programmierschnittstellen, engl. Application Programming Interfaces (APIs), sind zu einem fundamentalen Bestandteil moderner Softwareentwicklung geworden und spielen eine entscheidende Rolle bei der Interaktion zwischen verschiedenen Softwarekomponenten, Plattformen und Diensten. Verschiedene API-Typen bieten lokalen oder verteilten Zugriff auf Programmbibliotheken, das Betriebssystem oder Dienste über das Internet. APIs definieren die Methoden und Protokolle, über die Anwendungen miteinander kommunizieren können, und ermöglichen es Entwicklern, auf standardisierte Weise auf die Funktionalität anderer Software zuzugreifen und diese zu nutzen.

In diesem Abschnitt wird die Umsetzung exemplarisch in Java aufgezeigt, um in einer derzeit ausgereiften und in der Geschäftswelt verwendeten Programmiersprache die lokale Anwendungsintegration aufzuzeigen. Ein Export der Funktionalitäten der Programmierabstraktion und öffentliche Bereitstellung bspw. über eine REST-Schnittstelle für den entfernten Zugriff ist eine denkbare Fortführung.

3.4.1 Konfiguration

Durch Auswahl von Optionen und setzen von Richtlinien werden dynamische Mengen konfiguriert, dass ihre Semantik von der Standardsemantik abweicht. Die Konfiguration wird von verschiedenen Rollen in unterschiedlichen Phasen eines Software–Lebenszyklus vorgenommen (vgl. Abschnitt 2.4.1), beginnend vom Schnittstellenbereitsteller und Anwendungsentwickler zur Entwurfszeit, vom lokalen Administrator zur Bereitstellungszeit und vom tatsächlichen Benutzer zur Laufzeit.

Die resultierende Semantik der dynamischen Menge ergibt sich dann aus den kombinierten Konfigurationen, bei denen kaskadierend eine spätere Konfiguration, die einen bestimmten Aspekt betrifft, normalerweise eine frühere Konfiguration überschreibt, die denselben Aspekt betrifft. Optionale, universell auf verschiedene Aspekte anwendbare Sprachergänzungen ermöglichen eine gesonderte Priorisierung, die das Standardüberschreiben durch nachgelagerte Konfigurationsstufen ausschließen können.

Die Konfiguration erfolgt in Datenstrukturen je eingesetzter Programmiersprache oder in Dateiformate. Für Konfigurationsdateien haben sich textbasierte Dateiformate etabliert, da sie Einstellungen und Präferenzen auch im Fall einer Unterbrechung der Laufzeit einer Anwendung persistent vorhalten, plattformübergreifend einsetzbar sind sowie je nach Komplexität und formatierter

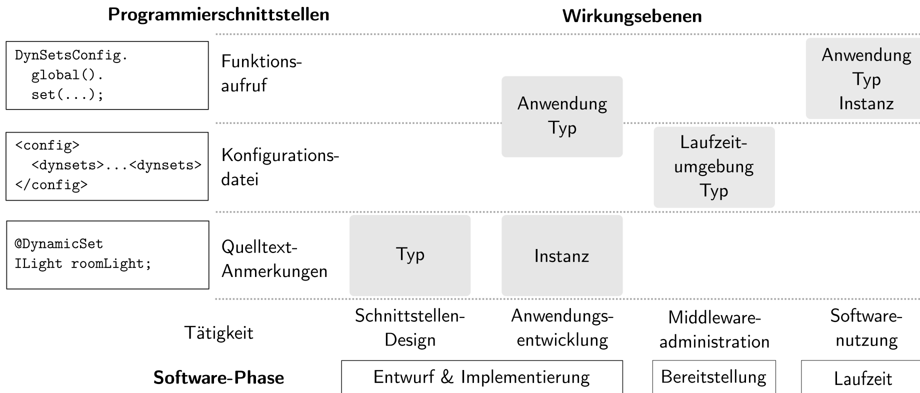


Abbildung 3.3: Konfigurationsoptionen für dynamischer Mengen

Darstellung der Syntax eine erschließbare Lesbarkeit für Menschen aufweisen. Die Verbreitung und Akzeptanz ist auch mitbegründet durch Verfügbarkeiten von Anwendungsprogrammen zum Lesen, Durchsuchen, Zusammenführen, Formatieren und Schreiben für eine große Abdeckung von Programmiersprachen. Weitverbreitete Dateiformate für Konfigurationsdateien sind bspw. XML, JSON und YAML, aber auch INI (Initialisierungsdatei) oder .properties (Schlüssel-Wert-Eigenschaften für die Programmiersprache Java).

Abbildung 3.3 zeigt die möglichen Konfigurationen dynamischer Mengen über die drei motivierten Programmierschnittstellen in Phasen im Software-Lebenszyklus. Dabei sind unterschiedlichen Wirkungsebenen der Konfiguration ermöglicht. Die Konfiguration von dynamischen Mengen erfolgt durch Metadaten direkt im Programmcode durch Verwendung von Sprachkonstrukten für Metadaten, wie Annotationen, oder diese ausgelagert in Konfigurationsdateien. Während der Umsetzung einer Anwendungsentwicklung definieren Bereitsteller von Schnittstellen das Standardausführungsverhalten über die Schnittstellendeklaration sowie Anwendungsentwickler für die typen- oder instanzweise Verwendung in dem speziellen Anwendungskontext. Zur Bereitstellungszeit können Administratoren die Semantik überschreiben oder um spezifische Konfigurationsoptionen ergänzen. Dies wird erreicht, indem Konfigurationsdateien, hier ein Bereitstellungsdeskriptor (engl. Deployment Descriptor), zusammen mit der Anwendung bereitgestellt werden. Zur Laufzeit kann die Konfiguration mithilfe der Konfigurations-API dynamischer Mengen angepasst werden. Diese API kann beispielsweise verwendet werden, um Laufzeitoptimierungen zu implementieren oder um dem Benutzer zu ermöglichen, seine persönlichen Präferenzen auszudrücken.

Das Zusammenführen mehrerer Konfigurationen zu unterschiedlichen Aspekten dynamischer Mengen ist in Abbildung 3.4 schematisch dargestellt. Universelle Konfigurationsaspekte werden von lokalen Konfigurationen überschrieben und bilden die anwendbare Konfiguration zur Laufzeit. Bspw. gibt ein Schnittstellendesigner über Annotationen die Aggregationsfunktion für eine Methoden an,

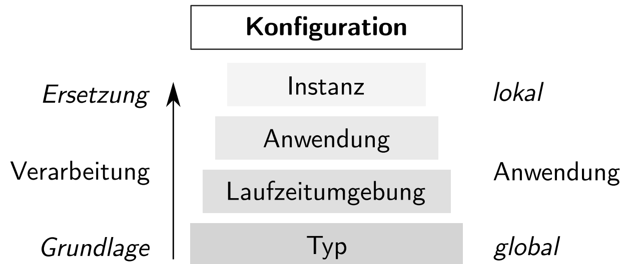


Abbildung 3.4: Kaskadierende Konfigurationsgenerierung

die später vom Anwendungsentwickler bei Deklaration einer Schnittstelle und Verwendung als Dynamische Menge mittels Annotation in eine alternative Aggregation überschrieben wird. Die in der Implementierung gesetzte maximale Rundlaufzeit für Methodenaufrufe über dynamische Mengen, mag von einem Middleware-Administrator bereits über eine für die Laufzeitumgebung anwendbare Konfigurationsdatei geändert worden sein, dann zur Laufzeit über eine grafische Oberfläche der Anwendung durch einen Nutzer wieder überschreiben, um ein beabsichtigtes Laufzeitverhalten zu erreichen.

3.4.2 Laufzeitumgebung

Die Funktionalitäten von Laufzeitumgebungen wurden in Abschnitt 2.4.4 vorgestellt. Das Designprinzip *Inversion of Control (IoC)* kehrt die Steuerung der Ablauflogik einer Anwendung um (vgl. Einführung in Abschnitt 2.4.1): Anstatt dass eine Anwendung die Kontrolle über ihre eigenen Abhängigkeiten hat, gibt sie diese an ein Framework oder eine Laufzeitumgebung ab. Diese übernehmen die Verantwortung für die Erstellung und Verwaltung von Objektgraphen sowie deren Interaktionen.

Die Programmierabstraktion der dynamischen Mengen nutzt die vorgestellten Entwurfsmuster und Programmierrichtlinien aus. Schnittstellen für deren Aufruf werden zum einen objektorientiert als Klassen- und Methoden deklariert sowie am Beispiel der Programmiersprache Java mit Annotation für ein intuitives Einweben in den Quellcode vorgestellt. Die umgesetzte Programmierabstraktion nutzt Schnittstellen als Grundlage, um aus Sicht der Anwendung portabel auf andere Implementierungen dynamischen Mengen zu bleiben und einen möglichen *Vendor-Lockin* zu vermeiden.

Listing 3.2 zeigt die Definition der Annotation `DynamicSet` in der Programmiersprache Java mit optionalem Parameter `name`. Die Annotation mit System-Annotation spezifiziert den Zugriff auf die Annotation, hier zur Laufzeit ermöglicht, damit Frameworks auch bei der Instanziierung umgebener Anwendungsklassen Zugriff auf die Annotation haben. Die Einsetzbarkeit ist auf Felder einer Klasse gesetzt. Durch Reflexionsroutinen wird für das Feld ein dynamischer Mengen-Stellvertreter erzeugt und dem Feld zugewiesen.

```
1 import java.lang.annotation.*;
2
3 @Retention(RetentionPolicy.RUNTIME)
4 @Target(ElementType.FIELD)
5 public @interface DynamicSet {
6     String name() default "";
7 }
```

Listing 3.2: Annotationsdeklaration

Am Beispiel des Java-Frameworks *Spring (Boot)* [180] werden die notwendigen Schritte skizziert, um die dynamischen Mengen als Programmierabstraktion umzusetzen. Die vorgenannte und nachfolgend vorgestellten Schnittstellen zu Dynamischen Mengen als Annotationen oder Funktionsaufrufe werden als gegeben gesehen und über eine Bibliothek integriert. Annotierte Anwendungsklassen in Spring werden als *Bean* verwaltet, sie werden nicht mit dem *new*-Operator von Anwendungsentwickler instanziiert, sondern das Framework übernimmt die nachgelagerte Erstellung (lazy loading) und verwaltet den entstehenden Objektgraphen. Eine als *Komponente* annotierte Klasse wird bei bei der Bereitstellung einer Spring-Anwendung ausgelesen. Implementiert eine Komponente die Schnittstelle BeanPostprocessor[2] können über die Methode postProcessBeforeInitialization(Object bean, String beanName) Anweisungen gesetzt werden, die nach Erstellung der Instanz ausgeführt werden sollen. Jetzt ermöglicht der Einsatz von Reflexionsroutinen das Überprüfen der Felder auf Annotationen. Im Fall von dynamischen Mengen wird ein dynamischer Stellvertreter erstellt, Datenstrukturen zur Mitgliederverwaltung erstellt und Fallunterscheidungen für Methodenaufrufe der Schnittstelle sowie dynamischer Mengen getroffen.

3.4.3 Deklaration und Instanziierung

Eine Instanz einer dynamischen Menge wird während der Anwendungsentwicklung deklariert. Beim Einsatz von Annotationen wird @DynamicSet vor einem Klassenattribut platziert. Darüber hinaus können mithilfe der Annotation @SelectConstraint zusammen mit der Mengendefinition deklarativ Auswahlkriterien angegeben werden, die ein Objekt erfüllen muss, um Mitglied der dynamischen Menge zu werden. Die Auswahlkriterien bestehen aus Einschränkungen, die Attribute eines Objekts auswerten oder Werte entsprechender Getter-Methoden zurückgeben. Eine dynamische Menge, die zur Entwurfszeit deklariert wurde, wird von der Middleware-Laufzeitumgebung erstellt und der Anwendung mithilfe der Abhängigkeitsinjektion bereitgestellt. Ein dynamischer Proxy für die dynamische Menge wird mithilfe des Reflexionsmechanismus von Java automatisch aus der jeweiligen Objektschnittstelle generiert. Gemäß der Standardsemantik dynamischer Mengen löst der Aufruf einer Methode auf dem lokalen

[2] org.springframework.beans.factory.config.BeanPostprocessor

```
1 @DynamicSet("WindowLights")
2 @SelectionConstraint ("powered=true AND location={{currentRoom}}")
3 @Aggregation(clazz=AllAggregation.class, method="isPowered,
      isDimmable")
4 ILight light;
5
6 lights.isPowered();
7
8 Object o = DynSetBuilder.get(ILight.class, "WindowLights");
9 ILight light = (ILight) o;
10 IDynSet lightSet = (IDynSet) o;
11
12 light.isPowered();
13 lightSet.count();
```

Listing 3.3: Anwendungsintegration Dynamische Mengen.

Stellvertreter einen entsprechenden Methodenaufruf für alle derzeit in der Menge enthaltenen Mitglieder aus. Listing 3.3 zeigt, wie der Code der Anwendung MyLightApp so annotiert wird, dass eine dynamische Menge (deren Mitglieder vom Typ ILight sind) in das Klassenmitglied *light* eingefügt wird. Dies geschieht, wenn die Instanz von der umgebenden Anwendungsklasse von der Laufzeitumgebung erstellt wird. Durch die angegebenen Auswahlkriterien wird sichergestellt, dass nur Leuchten, die sich im angegebenen Raum befinden und eingeschaltet sind, Mitglieder der Menge werden. Nach der Injektion kann die Anwendung Methoden auf dem eingestellten Proxy aufrufen.

3.4.4 Aggregation

Da die Signaturen einer Methode einer dynamischen Menge und derselben Methode eines entsprechenden Einzelobjekts identisch sind, ist es erforderlich, die Rückgabewerte, die sich aus der Replikation von Aufrufen der Methode an Mengenmitglieder ergeben, zu einem einzigen Rückgabewert zusammenzufassen, wenn mehr als ein Aufruf erfolgt. Um dies zu erreichen, kann ein Programmierer aus einer Reihe vordefinierter Aggregationsmethoden wählen (z.B. Minimum, Maximum, Durchschnitt, Mehrheit, zuerst). Abhängig von der Semantik der Aggregationsmethode ist eine unterschiedliche Anzahl bzw. ein Teil der verfügbaren Ergebniswerte Voraussetzung, bevor der einzelne Rückgabewert ermittelt und an die Anwendung zurückgegeben werden kann. Beispielsweise erfordert die Rückgabe des Maximums, dass alle Rückgabewerte verfügbar sind, während für die Rückgabe des ersten Werts ein einzelner Rückgabewert ausreicht. Neben der Verwendung vordefinierter Aggregationsmethoden besteht für Entwickler auch die Möglichkeit eigene zu implementieren.

Listing 3.4 zeigt eine anwendungsspezifische Aggregation mithilfe einer separaten Klasse. Die Annotation @OnAggregate kennzeichnet die allValuesTrue-Methode als Aggregationsmethode, an die ein Array von Werten übergeben wird und die einen einzelnen Wert zurückgibt. Die Verwendung dieser neuen Aggregations-

```
1 class AllAggregation <T > {
2   @OnAggregate
3   boolean allValuesTrue ( Object [] o ) {...}
4 }
```

Listing 3.4: Aggregationsmethode für Dynamische Mengen

methode wird in Listing 3.3 mithilfe der Annotation @OnAggregation deklariert. Hier gibt die Anmerkung an, dass die neue Aggregationsmethode in der Klasse AllAggregation implementiert ist und dass diese auf die Methoden isPowered und isDimmable angewendet werden soll.

3.4.5 Management

Die bisher beschriebene Funktionalität dynamischer Mengen ist für die Anwendung vollständig transparent. Ein Entwickler muss den Anwendungscode nicht anpassen, da jede Anwendung, die mit einem einzelnen Objekt umgehen kann, auch mit einer dynamischen Menge umgehen kann. Der Bereitsteller einer Anwendung kann jedoch eine dynamische Menge so konfigurieren (z.B. eine bestimmte Aggregationsmethode über einen Bereitstellungsdeskriptor bereitstellen). Es passt besser zur beabsichtigten Anwendungsfunktionalität. Wenn die Anwendung selbst weiß, dass sie mit einer dynamischen Menge interagiert, ergeben sich zusätzliche Möglichkeiten und die Anwendung kann Methoden einer dynamischen Menge aufrufen, die nicht in der ursprünglichen Geschäftsschnittstelle enthalten sind. Dies bedeutet, die Middleware erweitert die Mengenschnittstelle mit Methoden, die beispielsweise verwendet werden können, um neue Mengen aus vorhandenen Mengen abzuleiten, über die Mengenmitglieder zu iterieren und die aktuelle Mitgliederanzahl anzufordern, Statistiken zu erstellen oder die Auswahlkriterien einer Menge zu ändern.

Die manuelle Verwaltungsschnittstelle wird vom Programmierer verwendet, nachdem die Menge erstellt und von der Laufzeitumgebung eingefügt wurde. Listing 3.5 zeigt die entsprechende Schnittstelle DynSet, die die Set-Schnittstelle der *Collection API* von Java erweitert. Der Programmierer kann die geerbten Methoden verwenden, um über die einzelnen Mitglieder der Menge zu iterieren (iterator), die aktuelle Anzahl der Mengenmitglieder abzurufen (size), zu testen, ob ein Objekt derzeit ein Mengenmitglied ist (contains) und Objekte zum Set hinzuzufügen und daraus zu entfernen. Der erste Methodenblock ermöglicht es, aus vorhandenen Mengen eine neue dynamische Menge abzuleiten. Eine Teilmenge kann erstellt werden, indem einer vorhandenen Menge weitere Einschränkungen hinzugefügt werden, und mit Methoden, die den üblichen Mengenoperationen entsprechen, ist es möglich, die Vereinigung, Schnittmenge oder relative Ergänzung mehrerer Mengen zu erstellen. Der zweite Block ermöglicht es, die Auswahlkriterien der dynamischen Menge über die Methode setConstraints festzulegen. Dies veranlasst die Middleware, die neuen Kriterien auszuwerten und die Menge entsprechend zu aktualisieren, sodass nach der Rückkehr des Aufrufs genau

```
interface DynSet<T> extends Set {
  // derivation from existing sets
  DynSet  subset(Constraint ... c);
  DynSet  union(DynSet... s);
  DynSet  intersection(DynSet... s);
  DynSet  complement(DynSet a, DynSet b);
  DynSet  difference(DynSet a, DynSet b);
  // selection constraints
  void    select();
  void    setConstraints(Constraint... c);
  Constraint[] getConstraints();
  // member updating
  void    setAutoUpdate(boolean b);
  boolean isAutoUpdate();
  // set listener
  void    addListener(Object listener);
  void    removeListener(Object listener);
}
```

Listing 3.5: Verwaltungsschnittstelle Dynamische Mengen.

die Objekte Mitglied der Menge sind, die die neuen Kriterien erfüllen. Um die Middleware dazu zu veranlassen, die aktuellen Auswahlkriterien neu auszuwerten, ohne sie zu ändern, kann die Auswahlmethode select aufgerufen werden.

Die automatische Verwaltungsschnittstelle ist auch in Listing 3.5 dargestellt. Die Methoden des vorletzten Methodenblocks können verwendet werden, um eine Funktionalität namens „Auto Update“ mithilfe der setAutoUpdate-Methode ein- oder auszuschalten. Wenn es aktiviert ist, wertet die Middleware die Auswahlkriterien der dynamischen Menge selbstständig neu aus, um die Mengenmitglieder regelmäßig zu aktualisieren. Außerdem löst der Aufruf mehrerer Methoden der manuellen Verwaltungsschnittstelle eine Laufzeitausnahme aus. Dadurch sollen unbeabsichtigte Interferenzen zwischen manuellen Methodenaufrufen und denen, die durch die Funktion zur automatischen Aktualisierung ausgelöst werden, vermieden werden.

Damit die Anwendung auf bestimmte Ereignisse reagieren kann, ist es möglich, mithilfe der Methode addListener Listener zu einer dynamischen Menge hinzuzufügen, während vorhandene Listener mit der Methode removeListener entfernt werden können. Listing 3.6 zeigt einen Quellcodeausschnitt wie Listener mit der Methode addListener hinzugefügt werden können. In diesem Beispiel sind die Methoden foo und bar mit den Annotationen @OnMemberAdded und @OnMemberRemoved gekennzeichnet, um die hinzugefügten bzw. entfernten Mengenmitglieder als Parameter abzurufen. Alternativ ist es auch möglich, Anwendungsmethoden des Objekts, das eine dynamische Menge als Mitglied hat (z.B. Anwendungsklasse), mit diesen beiden Annotationen zu annotieren. Für den Fall, dass ein Objekt mehr als einen einzige dynamische Menge hat als Mitglied muss der dynamische Zielsatz als Parameter für die Annotation angegeben werden, z.B. @OnMemberAdded(„light“).

```
1 light.addListener ( new Object () {
2   @OnMemberAdded
3   void foo ( Object ... o ) { ... };
4
5   @OnMemberRemoved
6   void bar ( Object ... o ) { ... };
7 }) ;
```

Listing 3.6: Ereignisverarbeitung dynamischer Mengen

3.4.6 Erweiterung der Logik

Durch die Programmierabstraktion der dynamischen Mengen werden Methodenaufrufe an der in einer Anwendung deklarierten Schnittstelle von der Laufzeitumgebung übersetzt und transparent für alle Mengenmitglieder ausgeführt. Dies erleichtert einerseits die Steuerung der durch eine dynamische Menge repräsentierten Objekte erheblich, beschränkt andererseits dadurch die nutzbaren Funktionen auf die eines einzelnen Geräts, wodurch das Potenzial vieler Geräte verschwendet wird.

Selbst wenn beispielsweise keine Lampe in einer dynamischen Menge dimmbar ist, kann die Gesamthelligkeit durch Einschalten einer kleineren oder größeren Teilmenge der Lampen angepasst werden. Dadurch erhalten Anwendungen einen Mehrwert, der über die Funktionalität und damit die Schnittstelle eines einzelnen Geräts hinausgeht. Um dieses Potenzial zu nutzen, wird Entwicklern die Deklaration benutzerdefinierter Methoden ermöglicht, die Geschäftslogik mit festgelegter Semantik kombinieren. Ein Entwickler kann eine neue Schnittstelle definieren, die die ursprüngliche Schnittstelle und die Management-Schnittstelle dynamischer Mengen um zusätzliche deklarierte Geschäftsmethoden erweitert, deren Ausführungsrichtlinien durch Metadaten angegeben wird.

Zu diesem Zweck kann der Entwickler eine neue Schnittstelle definieren wie in Listing 3.7 für die beispielhafte Beleuchtungsanwendung dargestellt. Die Schnittstelle CustomLights erweitert die Geschäfts-Schnittstelle ILight und die Verwaltungsschnittstelle DynSet<ILight> für dynamische Sets. Daher kann ein entsprechender Stellvertreter, der von der Middleware generiert und injiziert wird, wie bisher sowohl als einfache Leuchte verwendet als auch als dynamische Menge von Leuchten verwaltet werden. Für letzteres ist jedoch keine zusätzliche Typumwandlung mehr erforderlich. Darüber hinaus kann ein Entwickler weitere Geschäftsmethoden deklarieren, die später auch vom generierten Stellvertreter implementiert werden. Die tatsächlich zu verwendende Implementierungslogik wird jedoch bereitgestellt, indem entweder vorhandene Geschäftsmethoden und Mengenfunktionen neu eingewebt werden (bspw. verschiedene Ergebnisaggregationen) oder Methodenaufrufe an ergänzende Komponenten delegiert werden, die den erforderlichen Code enthalten. Diese beiden Möglichkeiten werden im Folgenden beschrieben.

```java
interface CustomLights
    extends ILight, DynSet<ILight> {
  // remix of existing functions
  @Aggregation(
    clazz=Average.class,
    method="isPowered" )
  float getBrightness();
  // mixin of additional functions
  @Mixin(StepSwitch.class)
  @Config(name="steps", value="10")
  StepSwitch stepswitch();
  default void brighter() {
    stepswitch().forMore(
      Invoke.method("setPowered")
            .args(true) );
    stepswitch().more();
} }
```

Listing 3.7: Custom interface with enriched business methods.

Mengen-Methoden

Die Aggregation von Rückgabewerten zu einem einzigen Datenwert ist Voraussetzung, um eine Methode transparent auf einer Menge von Objekten aufzurufen, als ob die Methode auf einem einzelnen Objekt aufgerufen würde. Als direkte Folge muss der aggregierte Wert vom gleichen Typ sein wie der ursprüngliche Rückgabewert. Darüber hinaus muss der aggregierte Wert im Anwendungskontext, in dem die Methode aufgerufen wird, Sinn ergeben, ohne den Entwickler zu überraschen. Dies beschränkt die anwendbaren Aggregationsfunktionen oft auf wenige Auswahlmöglichkeiten wie Erste, Mehrheit oder Minimum und Maximum. Geschäftsmethoden, die in der erweiterten Schnittstelle neu definiert werden, sind jedoch nicht an diese Beschränkungen gebunden. Daher können ursprüngliche Geschäftsmethoden mit verschiedenen Aggregationsfunktionen kombiniert und gemischt werden, um zusätzliche Daten und Informationen über die dynamische Menge zu berechnen und abzufragen. Angenommen, die dynamische Menge in der motivierten Beleuchtungsanwendung besteht aus nicht dimmbaren Leuchten, kann die Gesamthelligkeit als den Anteil der eingeschalteten Leuchten definiert werden.

Entsprechend wird die Methode getBrightness in Zeile 7 von Listing 3.7 deklariert, um den aktuellen Helligkeitswert abzufragen. Für die Berechnung des tatsächlich Wertes, weist die @Aggregation-Annotation der Methode darüber die Middleware an, die Ergebnisse der isPowered-Methode zu mitteln (d.h. Average.class zu verwenden). Diese Methode wird von der ursprünglichen Schnittstelle der Leuchten bereitgestellt und bestimmt, ob eine Leuchte ein- oder ausgeschaltet ist. Wenn man die Methode für alle Mengenmitglieder aufruft und den Durchschnitt berechnet (d.h. true wird als 1,0 und false als 0,0 interpretiert), erhält man schließlich den Anteil der leuchtenden Lampen in der Menge.

```
1 class StepSwitch {
2
3     public void forMore(InvokeConfig ic){...};
4     public void more() {...};
5     public void forLess(InvokeConfig ic){...};
6     public void less() {...};
7     public void configure() {...};
8     ...
9 }
```

Listing 3.8: Exemplary dynamic set declaration.

Mixins für erweiterte Funktionalität

Nicht alle neu deklarierten Methoden können einfach implementiert werden, indem der Aufruf an eine bereits vorhandene Geschäftsmethode des Geräts weitergeleitet und eine andere Aggregationsfunktion für erhaltene Rückgabewerte verwendet wird, um die gewünschten Ergebnisse zu berechnen. Erweiterte Funktionen erfordern häufig auch eine sehr spezifische und zusätzliche Implementierungslogik. Neben dem Schreiben eigener Implementierungen können ergänzende Komponenten und Bibliotheken notwendige Logik und fehlende Funktionen auf wiederverwendbare Weise gebündelt bereitstellen, die flexibel in eigenen Code eingemischt werden können. Während mehrere Programmiersprachen wie Python, Go oder Ruby das Konzept von Mixins nativ unterstützen, muss dieses Entwurfsmuster in anderen Sprachen nachgeahmt werden. Für die Umsetzung für dynamischen Mengen in der Programmiersprache Java können Mixins realisiert werden, indem Getter-basierte Abhängigkeitsinjektionen in Kombination mit Standardmethoden genutzt werden, die in Java 8 verfügbar sind.

In den Zeilen 9-16 von Listing 3.7 ist ein Beispiel. Die Hauptidee besteht darin, für die Beleuchtungsanwendung einen Stufenschalter zu realisieren, der die Lampen der dynamischen Menge schrittweise einschaltet, anstatt sie alle auf einmal einzuschalten. Zu diesem Zweck werden die Mitglieder der dynamischen Menge in gleich große Teilmengen unterteilt. Bei jedem weiteren Aufruf schaltet der Schalter die Lampen einer weiteren Untergruppe ein und erhöht so die Gesamthelligkeit. Da eine solche schrittweise Aufrufstrategie auch in anderen Anwendungskontexten sinnvoll ist, ist die Implementierung in eine eigene StepSwitch-Klasse ausgegliedert, vgl. Listing 3.8. Die @Mixin-Annotation weist die Middleware an, auch ein neues StepSwitch-Objekt zu instanziieren, wenn eine dynamische Menge von CustomLights erstellt wird. Name/Wert-Paare, bereitgestellt durch die optionale @Config-Annotationen, werden an das instanziierte Objekt übergeben und zu Konfigurationszwecken verwendet, um beispielsweise die Anzahl der Schaltschritte auf 10 festzulegen, bis die Lampen des gesamten Sets eingeschaltet sind. Immer wenn später die stepswitch-Methode aufgerufen wird, wird dieses StepSwitch-Objekt zurückgegeben. Um die Funktionen nutzen zu können, wird auf die Verwendung von Standardmethoden aufgebaut. Mit Standardmethoden können Entwickler Funktionen zu Schnittstellen hinzufügen, indem sie Methodendeklarationen mit einer Standardimplementierung versehen,

Annotation	Attribute	Datentypen (Standardwerte)
Mengendeklaration		
DynamicSet	name	String
Mitgliederauswahl		
SelectionConstraint	predicate	<? extends AbstractSetConstraint>
	query	String
	static	bool(false)
	instances	String[]
	operation	SetLogical(.UNION)
Ausführung		
Latency	max	String
Throughput	all	bool(true)
	minCount	int
	minPercent	float
Multiprobe	count	int(0)
	delay	String("1s")
ValueAge	cached	bool(true)
	max	String("1s")
Aggregation		
Aggregation	method	String[]
	clazz	<? extends AbstractSetAggregation>
OnAggregate		
Listener		
OnMemberAdded		
OnMemberRemoved		
Funktionserweiterung		
Mixin	clazz	Class
Generisch		
Config	name	String
	value	String

Tabelle 3.1: Annotation API

die automatisch geerbt wird. Auf diese Weise können wir einen Aufruf der brighter-Methode zum Einschalten weiterer Lichter im DynamicSet bequem an die entsprechenden Funktionen des StepSwitch weiterleiten. Daher wird zunächst ein kompilierten Aufrufbehandler übergeben, der die Methode und die Argumente angibt, die für eine einzelne Lampe (d.h. setPowered(true)) aufgerufen werden sollen, um sie einzuschalten. Anschließend lässt der StepSwitch den Aufruf an eine Teilmenge der Leuchten senden Ebenso kann eine Verdunkelungsmethode definiert und implementiert werden, welche die Lampen nach und nach wieder ausschaltet.

3.4.7 Annotation API

Die Konfiguration dynamischer Mengen ist neben Konfigurationsdateien und Methodenaufrufen der API im Quellcode auch über Annotationen ermöglicht. In der Fallstudie unter Verwendung der Programmiersprache Java wird die Deklaration zur Implementierungszeit fokussiert. Tabelle 3.1 fasst die definierten Annotationen zusammen, deren Einsatz teilweise in den vorherigen Abschnitten genauer beschrieben wurde. Soweit nicht anders vermerkt sind die Annotationen für die Verwendung zu Feldern definiert und mit der Aufbewahrungsregel sie auch zur Laufzeit verwenden zu können, damit die Middleware sie bei Instanziierung der umgebenen Anwendungsklasse auslesen und anwenden kann. Annotation beginnend mit „On“ sind für den Einsatz zu Methoden vorgesehen, die bei Ereignissen zu einer Menge von der Middleware aufgerufen werden.

Ein DynamicSet markiert eine Schnittstellendeklaration als Kandidat für eine dynamische Menge. SelectionConstraint lässt Einschränkungen zur Auswahl von Mengenmitgliedern beschreiben. Dies kann durch Referenz auf eine abgeleitete Klasse geschehen, in der imperativ Prädikate programmiert werden, deklarativ eine Zeichenkette angegeben werden, die es von einer Implementierung der Middleware auszulesen, transformieren und anzuwenden gilt, oder Namen bereits deklarierter Mengen aufgezählt werden, die mit mengentheoretischen Kombination die Mitglieder der Menge definieren. Der Abschnitt *Ausführung* gibt durch Dienstgüte motivierten Konfigurationen für das Phasenmodell (vgl. Abschnitt 3.3.1) wieder, die in einer Aufrufstrategie angewendet werden (vgl. Abschnitt 3.5.2). Angaben zur maximalen Ausführungszeit für Methodenaufrufe an einer dynamischen Menge, deren Durchsatz zu Mengenmitgliedern, Wiederholungsausführungen und Wertegültigkeiten lokale zwischengespeicherter Rückgabewerte lassen die globale Konfiguration in der Middleware überschreiben. Die Anwendung von Annotation zu Aggregation, Listener und Funktionserweiterung wurden in vorherigen Abschnitten aufgezeigt.

3.5 Umsetzung

Die in vorherigen Abschnitten vorgestellten Konzepte zur Anwendungsintegration von dynamischen Mengen werden folgend für die Umsetzung skizziert. Abbildung 3.5 zeigt das Komponentenmodell der vorgeschlagen Architektur. Ein dynamischer Aufruf startet am Anfang im Knotenpunkt Anwendung. Erfolgt ein Methodenaufruf am Einstiegspunkt, dem dynamischen Stellvertreter, beginnt die Laufzeitumgebung damit, Konfigurationen zu ermitteln und die Objektaufrufe in einer Aufrufstrategie zu planen, welche durch das Phasenmodell im Abschnitt 3.3.1 beschrieben wurde. Der Aufrufplaner übernimmt die High-Level-Interaktion mit den Transportkomponente, welche paradigmen- und protokollspezifisch die Aufrufe über das Netzwerk ausführt. Einzelne Antworten werden zu den Anfragen korreliert und für eine Zusammenfassung weitergeleitet. Der

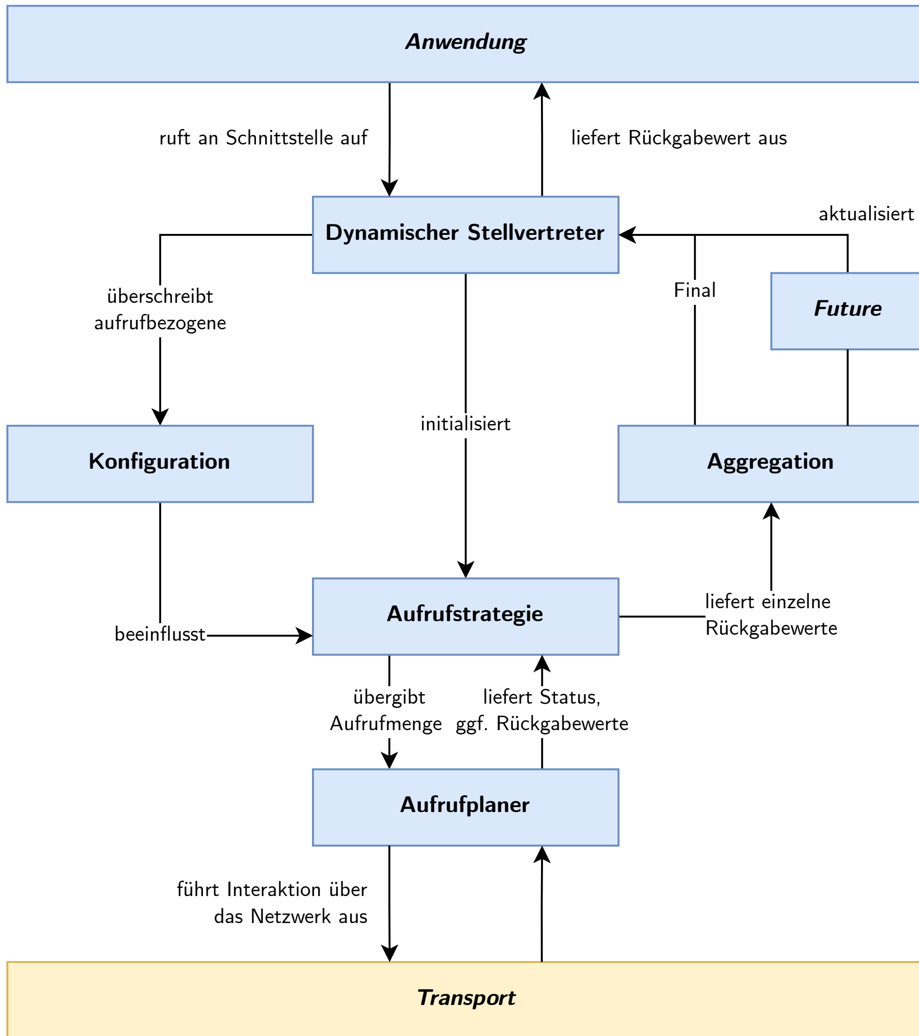


Abbildung 3.5: Schematische Darstellung der Methodenausführung dynamische Mengen

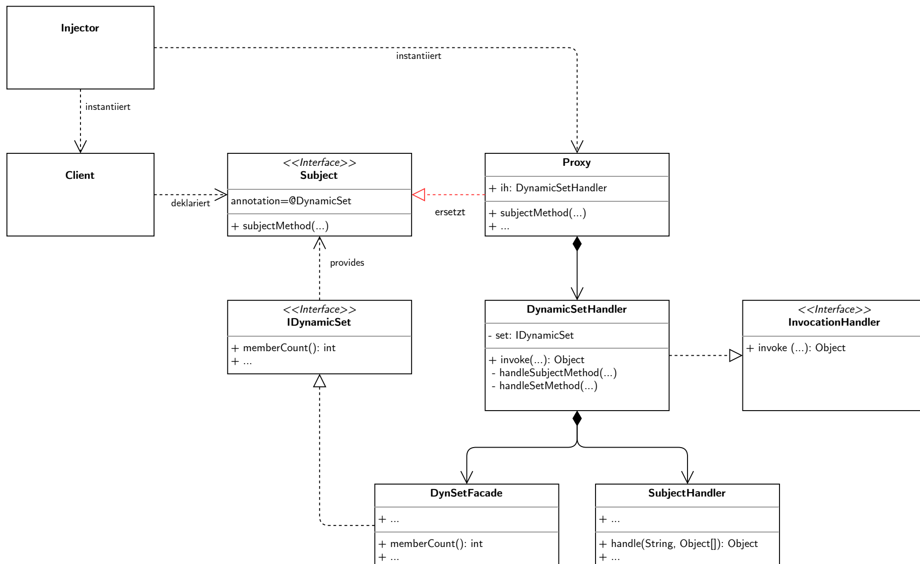


Abbildung 3.6: UML-Klassendiagramm, Stellvertreter für dynamische Mengen

dort berechnete Aggregationswert wird dem Stellvertreter zur Rückmeldung an die aufrufende Anwendung übergeben.

3.5.1 Dynamischer Stellvertreter

Der Stellvertreter ist Schnittstelle und Einstiegspunkt für Methodenaufrufe über dynamische Mengen. Abbildung 3.6 zeigt ein UML-Klassendiagramm, um den Ansatz mit einem dynamischen Stellvertreter zu veranschaulichen. Der *Klient* deklariert zur Entwurfszeit die Schnittstelle *Subject*, hier mit Annotation DynamicSet. Während der Bereitstellung der Anwendung instanziiert der *Injektor* einer Laufzeitumgebung den *Klienten*. Die Instanziierung durch einen von der Anwendung übergeordneten Kontext ist erforderlich, damit der zu erstellende Objektgraph von *Injektor* verwaltet werden kann. Das Auslesen des Quellcodes durch Reflexion ermöglicht das Injizieren des beabsichtigten Stellvertreter-Objektes in die Deklaration der Schnittstelle zur Bereitstellungszeit. Als Injektor abgebildet steht es für die Komponenten eines Frameworks, welches den Kontrollfluss über die Anwendung hält, vgl. 2.4.4. Die Schnittstelle des Subjekt wird um die Schnittstelle der dynamischen Menge erweitert (IDynamicSet), welche je nach Art der Einbindung in den Klienten von der Laufzeitumgebung ausgeführt werden. Der Stellvertreter bindet mit einem Aufrufbehandler (DynamicSetHandler, engl. invocation handler) ein, der Aufrufe der spezifischen Methoden des *Subjekt* (SubjectHandler) sowie die generischen der Mengenabstraktion (DynSetFacade) übernimmt und delegiert.

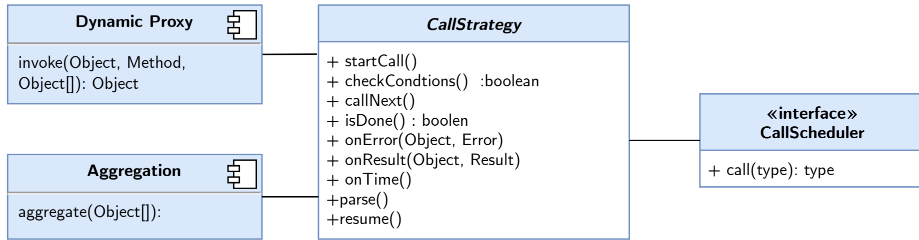


Abbildung 3.7: UML-Klassendiagramm, Aufrufstrategie

3.5.2 Aufrufstrategie und -planer

Die Aufrufstrategie hat Zugriff auf Beschreibungsobjekte der aufzurufenden entfernten Objekte. Jedes ist eindeutig identifizierbar, um in der Aufrufplanung und -ausführung jederzeit den aktuellen Zustand zu erhalten. Zusätzlich sind in den Beschreibungen statistische Informationen hinterlegt, die das vergangene Aufruf- und Antwortverhalten bewerten sowie die letzten Rückgabewerte attribut-lesender Methoden mit Zeitstempel. Für die Umsetzung der Aufrufplanungen greift die Aufrufstrategie auf den globalen Kontext der Laufzeitumgebung und Anwendung sowie den lokalen Kontext der dynamischen Menge bzw. nur des Aufrufs zurück. Über eine Schleife werden Objektkandidaten ausgewählt entsprechend des gebildeten Kontexts der umgesetzten Strategie. Die Menge wird dem Aufrufplaner übergeben und auf Antworten gewartet. Das Warten bzw. die Schleife kann pausiert werden, um Anpassungen, wie den Grad an Parallelität oder die Kandidatenmenge zu aktualisieren. Bei jedem Schleifendurchlauf werden die gesammelten Bedingungen überprüft, wie Zeitbudget, erwartetes oder fehlerhaftes Antwortverhalten, um Fortführen oder Terminierung entscheiden zu können.

Die Implementierung des CallScheduler nimmt die Beschreibungsobjekte der Aufrufkandidaten an und verwaltet in einer Standardimplementierung für jeden Aufruf einen eigenen Thread. Über die Fabrikmethode der Executor-Bibliothek in Java kann der Parallelitätsgrad durch die Thread-Poolgröße angepasst sowie durch die Registrierung eines Zeitereignisses am ScheduledExecutorService die Laufzeit limitiert werden, falls die Antwort nicht innerhalb der akzeptierten Zeitspanne abgeschlossen wird.

3.5.3 Aggregation

Die Aufrufstrategie ist unter einem eindeutigen Identifikator in der Laufzeitumgebung initiiert. Folglich kann der Aufrufplaner Antworten entfernte Objekte über die Aufrufstrategie der Aggregation zu übergeben. Entsprechend des Aufrufkontext wird hier auf ein finales Aggregationsergebnis gewartet, bis die (Teil-)menge an Mitgliedern einer dynamischen Menge erfolgreich den entfernten

Methodenaufruf behandelt hat, oder ein dynamischer Aggregationswert iterativ gebildet wird, der an registrierte Beobachter aktualisiert wird.

3.6 Verwandte Arbeiten

In der Literatur gibt es viel Forschung zu Programmierabstraktionen und insbesondere zur transparenten Gruppierung von Objekten. Im Folgenden werden jene Ansätze diskutiert, die dem hier vorgestellten Ansatz ähnlich sind.

Im Bereich der Sensornetzwerke gibt es mehrere Ansätze, die die Idee der Sensorfusion unterstützen und Ähnlichkeiten mit unserem Ansatz aufweisen. Die meisten Ansätze in diesem Bereich vermischen jedoch Code für Middleware-Funktionalität und Anwendungslogik, um kompakte Implementierungen mit geringem Platzbedarf zu erstellen. Beispielsweise schlagen Aberer et al. [1] die Middleware *Global Sensor Networks (GSN)* vor, deren zentrales Konzept die Abstraktion *virtueller Sensor* ist. Deklarativ formulierte XML-basierte Bereitstellungsdeskriptoren können mit einfachen SQL-Abfragen über lokale und entfernte Sensordaten zu kombiniert werden. Unabhängig davon schlugen Kabadayi et al. [134] auch virtuelle Sensoren vor. Dabei ist ein virtueller Sensor ein Softwaresensor, der Messwerte von mehreren physischen Sensoren kombiniert (z. B. zur Verbesserung der Qualität), die deklarativ angegeben werden können.

Im Bereich kontextsensitiver Anwendungen schlagen Sehic et al. [148] ein Programmiermodell für kontextsensitive Anwendungen im großen Maßstab vor, das als Origins Model bezeichnet wird. In diesem Modell ist ein Origin die elementare Anwendungskomponente und abstrahiert eine einzelne Kontextquelle. Er kann beispielsweise eine ursprüngliche Datenquelle umschließen oder Daten filtern, aggregieren, zusammenstellen und ableiten, indem er Daten anderer Ursprünge verwendet, die deklarativ mit Auswahlkriterien angegeben sind. Basierend auf dieser Auswahl können die Daten von den ausgewählten Ursprüngen entweder synchron durch einen normalen Aufruf oder asynchron durch Verwendung eines Futures angefordert werden. Im letzteren Fall kann ein Completion Handler definiert werden, der aufgerufen wird, wenn die angeforderten Daten verfügbar sind. Die Verwendung von Futures ermöglicht *Promise Pipelining*, dass die Latenz reduzieren kann, indem ein Verarbeitungsvorgang auf einem Future statt auf einem tatsächlichen Wert definiert wird. Die folgenden vier Ansätze sind im Bereich der verteilten Systeme angesiedelt. Diese Ansätze und unsere Arbeit haben gemeinsam, dass sie auf derselben Idee basieren, nämlich transparenten Zugriff auf eine Objektgruppe bereitzustellen.

Vicaire et al. [179] präsentieren die Programmierabstraktion *Bundle* für cyberphysische Systeme. Zum einen erfolgt Definition einer Gruppe von Sensoren und Aktoren und die Spezifikation, welche Aktionen diese Geräte tun sollen. Die Sichtbarkeit einer Menge bleibt gegeben und Aufrufreplikation von sowie Aggregation zu einer einzelnen Repräsentanz Teil der Arbeit.

Eugster et al. [44] stellen eine Programmierabstraktion namens *Distributed Asynchronous Collection (DAC)* vor, die eine Reihe von Ereignisobjekten zu einer Sammlung zusammenfasst. Der Zugriff auf eine Sammlung erfolgt über einen lokalen Proxy, der die Verteilung verbirgt und transparenten Zugriff bietet. Die Schnittstelle von DACs konzentriert sich hauptsächlich auf die Verwaltung der Mitglieder der Sammlung.

Zusätzlich zu Pull-basierten Methoden (z.B. `add` und `contains`) mit einem einzigen Rückgabewert bieten DACs Push-basierte Methoden, mit denen ein Client einen Rückruf registrieren kann, um in Zukunft benachrichtigt zu werden, z.B. wenn der Sammlung ein neues Objekt hinzugefügt wird. Diese Methoden werden dem zugrunde liegenden themenbasierten Publish/Subscribe-Paradigma zugeordnet. Der Hauptunterschied zwischen DACs und dynamischen Mengen besteht darin, dass DACs sich darauf konzentrieren, Ereignisobjekte, die Zustandsänderungen von Objekten darstellen (entweder per Pull oder Push), an Anwendungen zu übermitteln, während dynamische Mengen sich darauf konzentrieren, Anwendungen zu ermöglichen, Methodenaufrufe (entweder synchron oder asynchron) transparent an eine Reihe von Remote-Objekten auszugeben. Ein weiterer Unterschied besteht darin, dass DACs auf publish/subscribe angewiesen sind, während dynamische Mengen transparent verschiedene Transportprotokolle und Interaktionsparadigmen unterstützen. Felber et al. [48, 47] beschreiben einen *Object Group Service (OGS)*, der Fehlertoleranz und hohe Verfügbarkeit durch transparente Objektreplikation bietet. Der Dienst bietet einer Anwendung, Methoden einer Objektgruppe an über einen lokalen Proxy auf völlig transparente Weise aufzurufen, d.h. die Anwendung kann nicht zwischen einer Objektgruppe und einem Singleton-Objekt unterscheiden, das dieselbe Schnittstelle implementiert. Aufgrund dieser Transparenz können Anwendungen fehlertolerant gemacht werden, ohne ihren Code ändern zu müssen. Um Fehlertoleranz für harmlose Fehler zu erreichen, reicht es aus, einen beliebigen Rückgabewert zurückzugeben. So kann ein Aufruf zurückkehren, wenn die erste Antwort eintrifft, es kann aber auch so konfiguriert werden, dass der Aufruf zurückkehrt, nachdem die Mehrzahl der Antworten oder alle Antworten eingetroffen sind.

Fault-Tolerant CORBA (FT-CORBA) (vgl. [109, Kapitel 23]) und neuere Ansätze im Bereich Webservices [143] verwenden ebenfalls transparente Objektreplikation. Dynamische Mengen sind jedoch nicht auf Objektreplikation beschränkt, da sie beliebige Objekte bündeln können, die dieselbe Schnittstelle implementieren, und sie bieten neben Aufrufmechanismen auch erweiterte Funktionen.

Picco et al. [121] schlagen *Distributed Abstract Data Types (DADTs)* vor, die eine Menge von Instanzen eines abstrakten Datentyps in einem verteilten System logisch kapseln. Ein Anwendungsprogrammierer kann eine Schnittstelle für einen DADT definieren (dessen Methoden auf der Menge arbeiten) und er oder sie kann den Umfang eines DADT einschränken, indem er eine Ansicht definiert, die eine Teilmenge auf deklarative Weise darstellt. Methoden können transparent aufgerufen werden, zum Beispiel auf einer einzelnen Instanz, auf allen Instanzen, oder ein deklarativer Selektor kann verwendet werden, um die Zielinstanzen zu

bestimmen. Es ist auch möglich, über die Mitglieder zu iterieren und auf einzelne Mitglieder zuzugreifen.

3.7 Diskussion

Die Programmierabstraktion der dynamischen Mengen wurde in diesem Kapitel vorgestellt. Zur Entwurfszeit kennen Anwendungsentwickler von nicht die Beschaffenheit der Systemumgebung. Die Programmierabstraktion der dynamischen Mengen überbrückt die Lücke zwischen Entwurfs- und Laufzeit. Sie ermöglicht die Deklaration von Schnittstellen späterer Geschäftsobjekte, die Metadaten ergänzt werden, und zur Laufzeit durch einen Stellvertreter überschrieben werden, der Methodenaufrufe abfängt, repliziert und Antworten aggregiert. Die Anwendungsintegration erfolgt entweder abstrahiert als Instanz oder als Menge, was dem Anwendungsentwickler erweiterte Funktionalitäten gestattet. Eine Laufzeitumgebung übernimmt den Programmfluss der Anwendung und liest reflexiv Annotation aus dem Quellcode oder Konfigurationsdateien aus. Konfigurationen enthalten Auswahlkriterien für die Auswahl der Mitglieder einer Menge, überschreiben Aggregationsfunktionen, um aus mehreren einen Rückgabewert entsprechend der Methodensignatur zusammenzufassen, sowie Ausführungsrichtlinien an die Laufzeitumgebung hinsichtlich Dienstgüte-Aspekten. Zur Methodenausführung wurde ein Phasenmodell präsentiert, welches unter Berücksichtigung der erhaltenen Konfigurationen Objektkandidaten für Methodenaufrufe auswählt und terminiert.

Durch die nahe Einbindung entfernter Objekte als Schnittstellendeklaration in eine Anwendung, können Methodenaufrufe vom Anwendungsentwickler zur Implementierungszeit oder Benutzer zur Laufzeit so häufig erfolgen, als wären sie lokal. Wird durch Detailgrad der Kommunikation durch Abstraktion verborgen, können hochfrequente Aufrufe im Verständnis an einem lokalen Objekt schädliche Auswirkungen auf die Netzlast, die durch die Replikation an die Mengenmitglieder faktorisiert wird. Daher ist die Notwendigkeit adaptiven Kommunikationsmechanismen gefordert, die mit unterschiedlichen Adressierungen Objekte anfragen oder deren Zustände aktualisieren lässt.

Kapitel 4

Adaptive Unicast- und Multicast-Kommunikation

Inhalt

4.1 Motivation **96**
4.2 Interaktionsmuster **97**
4.2.1 Unicast Pull 98
4.2.2 Unicast Push 98
4.2.3 Multicast Push 99
4.2.4 Multicast Pull 99
4.3 Middleware-Architektur **100**
4.4 Heuristiken **101**
4.4.1 Verbindungsbasierte Heuristik 101
4.4.2 Geräte-Heuristik 102
4.4.3 Hybride Heuristik 103
4.4.4 Heuristiken mit Multicast Pull 105
4.4.5 Werteglättung 108
4.5 Umsetzung **109**
4.5.1 Aufrufe 109
4.5.2 Statistiken 110
4.5.3 Evaluierung 112
4.5.4 Algorithmen 113
4.6 Evaluation **118**
4.6.1 Gleiche Anfrageraten 122
4.6.2 Gespreizte Anfrageraten 125
4.6.3 Kostenfunktionen 127
4.6.4 Konfigurationen 129
4.6.5 Mengengrößen 132
4.6.6 Zeitliche Änderungen der Ereignisraten 133
4.7 Verwandte Arbeiten **135**
4.8 Diskussion **136**

4.1 Motivation

Die durch die dynamischen Mengen erreichte Transparenz ermöglicht es, dass die Anwendung zur Entwurfszeit nicht unbedingt wissen muss, mit wie vielen Objekten eines Typs sie zur Laufzeit interagiert. Die anwendungsnahe Integration erleichtert die Entwicklung, kann jedoch später zur Laufzeit auch zu signifikanten Leistungsproblemen führen.

Beispielsweise ruft eine Anwendung eine Methode eines Stellvertreters mit hoher Frequenz auf, um Informationen über den aktuellen Status abzurufen. Für jeden Aufruf wird eine Anforderung an alle Mitglieder der dynamischen Mengen gesendet, jedes Objekt antwortet mit einer Antwort und alle Ergebnisse werden vom Stellvertreter aggregiert, wenn sie eingetroffen sind. Dies kann nicht nur zu einer Überlastung des Netzwerks führen, sondern auch die Knoten überlasten, auf denen die Objekte und die Mengen gehostet werden. Bei Methoden, die nur Statusinformationen zurückgeben, ist dieser polling-basierte Ansatz besonders ineffizient, wenn die Anwendung eine Methode mit einer viel höheren Häufigkeit aufruft, als sich die angeforderten Informationen tatsächlich ändern. In diesem Fall gibt die Mehrheit der Aufrufe keine neuen Informationen zurück. In einem solchen Szenario wäre es von Vorteil, wenn die Objekte verfügbare Updates an den Proxy senden würden, anstatt die Anwendung nach Informationsaktualisierungen abfragen zu lassen. Der Proxy speichert dann die neuesten Informationen für jedes Objekt im Cache, sodass die aktuellen Informationen sofort zurückgegeben werden können, wenn die Anwendung sie anfordert. Dieser Push-basierte Ansatz funktioniert zwar gut für Methoden, die nur Statusinformationen zurückgeben, kann jedoch offensichtlich nicht einfach auf Methoden angewendet werden, die den Status der Objekte ändern. Darüber hinaus kann Push auch zu Leistungsproblemen führen, wenn (zumindest einige) der Objekte Informationsaktualisierungen mit hoher Frequenz senden. Dies ist insbesondere dann ineffizient, wenn die Anwendung neue Informationen mit viel geringerer Frequenz anfragt. In diesem Fall überschreiben die meisten Informationsaktualisierungen (von den jeweiligen Objekten) Informationen, die nicht an die Anwendung übermittelt wurden. Diese Argumentation zeigt, dass jeder der beiden Ansätze für bestimmte Szenarien geeignet ist, während er für andere ineffizient ist. Darüber hinaus wird offensichtlich, dass Push und Pull dual sind, da sie in den Szenarien effizient sind, in denen der andere Ansatz suboptimal eingesetzt ist. Da es im Allgemeinen schwierig ist, zur Entwurfszeit zu entscheiden, welche Art von Szenario zur Laufzeit für welches Paar von Anwendungen und Objekten vorherrscht, werden adaptive Ansätze untersucht, die zur Laufzeit zwischen Push und Pull wechseln.

Heuristiken sind in vielen wissenschaftlichen Bereichen von Bedeutung, besonders in der Informatik. Sie bieten vereinfachte Ansätze für komplexe Fragestellungen, jedoch ohne eine optimale Lösung garantieren zu können. Sie basieren auf Erfahrungswerten und sowie vereinfachten Annahmen, um die Komplexität eine Aufgabe zu reduzieren und eine schnelle Lösung anzubieten. In Situationen, in denen die Problemgröße oder -komplexität eine exakte Lösung zeitnah nicht

möglich macht, sind Heuristiken ein alternativer Ansatz, wie für Optimierungen oder im Bereich Künstliche Intelligenz. *Greedy* Heuristiken treffen schrittweise Entscheidungen mit lokaler Sicht, um schnell und günstig (vgl. gierig, engl. greedy) eine Teillösung zu erhalten, die zur Gesamtlösung beiträgt.

Zur adaptiven Gruppenkommunikation unter Verwendung der Programmierabstraktion der dynamischen Mengen geben Heuristiken den notwendigen Grad an Vereinfachung, um in einer überschaubaren Zeit Entscheidungen zu treffen. Das Ergebnis basiert auf lokaler Datenlage ohne vorhandenes Wissen der Netzwerktopologie, Zusammensetzung und Platzierung des Geräteensembles sowie Kommunikationsparadigmen. Je nach Ausgestaltung der Heuristik werden periodisch Statistiken zwischen Klienten ausgetauscht, um die notwendigen Zähler lokal vorliegen zu haben. Damit evaluiert ein Klient regelmäßig und ermittelt unter Hinzunahme einer Hysterese die günstigste Kommunikationsart für die Wirkungsweise entsprechend der Heuristik. Zur Förderung des Leseflusses werden Anwendungen a, dynamischen Mengen S_a sowie Geräte d beschrieben, deren Interaktionen in Heuristiken h_x einfließen und deren Ergebnisse diese beeinflussen. Gemeint sind hier jedoch die Stellvertreter (proxies), welche als Software die Repräsentation gegenüber dem lokalen Klienten darstellen.

Im vorliegenden Kapitel erfolgt eine Diskussion ausgewählter Forschungsfragen, darunter: Es soll erörtert werden, auf welche Weise Gruppenkommunikation mit einer geringeren Anzahl an Interaktionen adaptiv umgesetzt werden kann. Es soll erörtert werden, welche Heuristiken dazu geeignet sind, die Komplexität verteilter Anwendungen zu verbergen, um auf dieser Grundlage Kostenabschätzungen treffen zu können. Es soll eruiert werden, welches Wissen minimal notwendig ist und welche Abstraktionen hilfreich sein können.

Die Ausführungen basieren auf eigenen Veröffentlichungen [130] und [127]. Nach einer Einführung in die Interaktionsmuster in Abschnitt 4.2 und die Wirkungsweise der Methodenaufrufe in adaptiver Anwendung in Abschnitt 4.3 werden in Abschnitt 4.4 die Heuristiken vorgestellt. Abschnitt 4.6 präsentiert abschließend die Simulationsergebnisse unter verschiedenen Konfigurationen.

4.2 Interaktionsmuster

Dieser Abschnitt beschreibt die grundlegenden Interaktionsmuster, die von den Heuristiken verwendet werden, und stellt das entsprechende Kostenmodell vor. Die Interaktionsmuster können danach kategorisiert werden, ob sie Pull- oder Push-Kommunikation verwenden sowie ob sie Unicast oder Multicast senden. Es werden die Kombinationen Unicast-Pull, Unicast-Push, Multicast-Push und Multicast-Pull betrachtet. Das Kostenmodell ermöglicht es Proxys dynamischer Mengen sowie den darin gebundenen Objekten, die miteinander interagieren, die Kommunikationskosten abzuschätzen und damit Anpassungsentscheidungen ableiten. In den folgenden vier Unterabschnitten erläutern die Interaktionsmuster, die daraufhin in der Middleware angewendet werden.

Folgende Notationen werden dazu eingeführt: Die Menge A bezeichnet alle Anwendungen $\{a_1, \ldots, a_i\}$ bereitgestellt, während die Menge D alle Geräte $d_1, \ldots, d_j$ enthält. Darüber hinaus hat jede Anwendung a eine zugehörige dynamische Menge S_a, die eine Untermenge von Geräten enthält, die dem Filter der Anwendung F_a entsprechen: $S_a = \{d \in D \mid F_a(d)\}$. Schließlich gilt für jedes Gerät d, $A_d = \{a \mid d \in S_a\}$ ist die Menge aller Anwendungen, in denen das Gerät d Mitglied einer dynamischen Menge S_a ist.

4.2.1 Unicast Pull

Ruft eine Anwendung a eine Methode auf einer dynamischen Menge S_a auf, sorgt der Proxy der dynamischen Menge dafür, dass der Aufruf repliziert und an die Geräte weitergeleitet wird, die zu dem Zeitpunkt Mitglieder der Menge sind. Auf jedem dieser Geräte wird die Methode aufgerufen und der Rückgabewert an den Proxy zurückgesendet, wo die Middleware die Rückgabewerte zu einem Endergebnis aggregiert.

Das Aufrufen einer Methode an einer dynamischen Menge führt dazu, dass für jedes Gruppenmitglied zwei Nachrichten gesendet werden: eine Nachricht für die Anforderung und eine für die entsprechende Antwort (Pull-Kommunikation). Die Kommunikationskosten C_{UC}^{Pull} werden als Anzahl der Anfrage- und Antwortnachrichten definiert, die durch Unicast-Pull über einen beobachteten Zeitraum verursacht werden. Diese hängen von der Größe der dynamischen Menge $|S_a|$ ab sowie der Anzahl der Methodenaufrufe r_a in einem beobachteten Zeitraum:

$$C_{UC}^{Pull} = 2 \cdot r_a \cdot |S_a| \tag{4.1}$$

4.2.2 Unicast Push

Für viele Anwendungen sind aktuelle Daten essenziell und daher rufen sie an Geräten häufig Daten ab. Je nach Anfragefrequenz und Datenaktualisierung können dabei redundante Anfragen entstehen. In solchen Situationen besteht ein alternativer Ansatz darin, Datenänderungen von Geräten an interessierte Anwendungen zu übertragen, sobald diese verfügbar sind. Ohne explizite Aufforderung sendet das Gerät d eine Nachricht mit den neuen Daten an alle Anwendungen in A_d (Push Kommunikation).

Die Kommunikationskosten für Unicast-Push C_{UC}^{Push} entsprechen der Anzahl von Nachrichten, die von Gerät d an Anwendungen in A_d gesendet werden. Die Kosten sind somit proportional sowohl zur Größe von A_d als auch zur Anzahl der Datenaktualisierungen u_d , die während des beobachteten Zeitraums am Gerät d aufgetreten sind. Für Unicast-Push berechnet sich die Anzahl der Nachrichten also wie folgt:

$$C_{UC}^{Push} = u_d \cdot |A_d| \tag{4.2}$$

4.2.3 Multicast Push

Beim Unicast-Push sendet das Gerät eine Nachricht mit der Aktualisierung einzeln an jede Anwendung $a \in A_d$. Durch Nutzung von Multicast-Kommunikation sendet das Gerät eine Aktualisierung an alle Anwendungen gleichzeitig unter der Voraussetzung, dass die Netzwerkinfrastruktur Multicast-Kommunikation (z.B. IP-Multicast) unterstützt und die Multicast-Gruppen angemessen konfiguriert sind. Netzwerkkomponenten (z.B. Router und Switches) replizieren dabei gegebenenfalls eine Nachricht automatisch an den Knoten, wo sich Routen zu Empfängern teilen. Eine Multicast-Nachricht an n Empfänger ist teurer als nur eine einzelne Unicast-Nachricht, jedoch günstiger als n einzelne Unicast-Nachrichten.

Die genauen Kosten für den Multicast-Baum sind jedoch schwer zu bestimmen, da sie sowohl stark von der Netzwerktopologie als auch von der Lage der darin befindlichen Empfänger abhängen. Die Kommunikationskosten einer Multicast-Nachricht für n Empfänger werden mit einer Kostenfunktion $f(n)$ abgeschätzt, wobei $f(n)$ eine Funktion von n wie die Wurzelfunktion $\sqrt{n}$ ist. Diese vereinfachten Kostenfunktionen auf Basis der Anzahl der Kommunikationsempfänger vernachlässigen die Netzwerktopologie und versuchen nicht die reale Sprungzahl (Hop Count) zu berechnen oder anderweitig einen Multicast-Baum abzuschätzen, wie Steiner-Baum-Approximation.

Somit ergeben sich die Push-Kommunikationskosten für eine Anzahl von Datenaktualisierungen u_d von einem Gerät d an die Anzahl der empfangenden Anwendungen $|A_d|$ mit Multicast-Push wie folgt:

$$C_{MC}^{Push} = u_d \cdot f(|A_d|) \tag{4.3}$$

4.2.4 Multicast Pull

Mehrere Middleware-Implementierungen (wie [93, 159]) bieten an, Multicast-Kommunikation auszunutzen, um eine Anfrage effizient an mehrere Empfänger zu übermitteln. Unter Nutzung eines solchen Anforderungsschemas leitet der Stellvertreter einer dynamischen Menge somit einen Methodenaufruf an alle Geräte der Menge unter Verwendung nur einer Multicast-Nachricht weiter. Allerdings sendet jedes aufgerufene Gerät sein Ergebnis einzeln per Unicast-Kommunikation an den Proxy zurück. Daher reduziert Multicast-Pull die Kosten für die Anfrage, während die Antwort im Vergleich zum Unicast-Pull unverändert bleibt.

Die Abschätzung der Kosten einer Multicast-Nachricht an n Empfänger mit $f(n)$ erfolgt wie im Unterabschnitt zuvor. Die Kommunikationskosten C_{MC}^{Pull} für das Aufrufen von r_a Anfragen an der Anzahl der Geräte einer dynamischen Menge $|S_a|$, die Multicast-Anfragen an die Geräte und Unicast-Antworten zurück an die Anwendung a verwenden, werden berechnet durch:

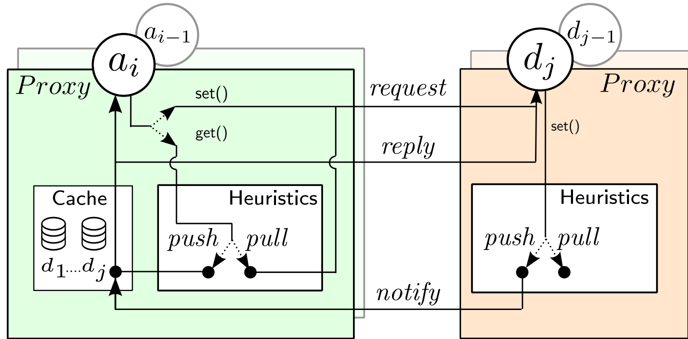


Abbildung 4.1: Middleware-Architektur

$$C_{MC}^{Pull} = r_a \cdot (f(|S_a|) + |S_a|) \tag{4.4}$$

4.3 Middleware-Architektur

Die Middleware-Architektur des Ansatzes ist in Abbildung 4.1 dargestellt. Das Ziel ist es zwischen Pull- und Push-Kommunikation umschalten zu können. Auf der linken Seite ist der Proxy einer dynamischen Menge einer Anwendung dargestellt, auf der rechten Seite ist der Proxy eines der Geräte, die Mitglied der dynamischen Menge sind gezeigt. Auf die Felder einer Instanz kann nicht direkt gegriffen werden, sondern nur über öffentliche Methoden, welche deren Zuweisung zurückgeben. Zur Vereinfachung werden Getter- und Setter-Methoden abgebildet. Get-Methoden bezeichnen eine Konvention in der objekt-orientierten Programmierung (OOP) zum Lesen von Werten eines Feldes [10]. Sie repräsentieren idempotente Funktionsaufrufe, die Daten des Gerätes abrufen und deren wiederholter Aufruf dessen Zustand nicht verändert. Im Gegenzug stehen Set-Methoden für zustandsverändernde Funktionsaufrufe, die entweder den Zustand des Gerätes verändern bzw. Logik ausführen. Ruft eine Anwendung eine Set-Methode an der dynamischen Menge auf, wird eine Anfrage an die Geräte gesendet und die resultierende Antwort an den Anwendungs-Stellvertreter zurückgesendet. Wenn die Anwendung eine Get-Methode aufruft und sich die Kommunikation im Pull-Modus befindet, findet die gleiche Anfrage/Antwort-Interaktion statt. Befinden sich mehrere Geräte im dynamischen Set der Anwendung, findet für jedes dieser Geräte eine Anfrage-/Antwort-Kommunikation statt. Befindet sich die Kommunikation für das Kommunikationspaar (a_i, d_j) im Push-Modus, gibt der Anwendungs-Proxy den lokal zwischengespeicherte Wert sofort an die Anwendung zurück. Tritt an dem Gerät eine Aktualisierung auf, wird eine Notifikation an den Anwendungs-Proxy gesendet, wo der zwischengespeicherte Wert entsprechend aktualisiert wird.

Wird eine Anfrage oder eine Notifikation an mehrere Empfänger (d.h. Geräte bzw. Anwendungen) gesendet, werden entweder einzelne Unicast-Nachrichten oder eine einzelne Multicast-Nachricht verwendet. Die Entscheidung, ob die Kommunikation von Pull auf Push oder zurück umgeschaltet wird und ob Unicast oder Multicast verwendet wird, basiert auf gesammelten Statistiken, die bei den Anwendungs-Proxys und den Geräte-Proxys gehalten werden. Weiterführend wird berücksichtigt, Notifikationen mithilfe von Multicast nur an eine Teilmenge von Anwendungs-Proxys zu übertragen, während die verbleibenden weiter Datenaktualisierungen von Geräten abrufen.

4.4 Heuristiken

Die Middleware-Architektur ermöglicht es, zur Laufzeit das Kommunikationsparadigma zwischen Pull und Push umzuschalten. Die folgend vorgestellten Anpassungsheuristiken reduzieren die Komplexität durch Betrachtung einzelner oder gruppierter Verbindungen von bzw. zu einem oder mehrerer Klienten sowie der Abstraktion der Transportrouten in einer Netzwerktopologie durch eine Kostenfunktion.

4.4.1 Verbindungsbasierte Heuristik

Die erste Heuristik h_{lnk} betrachtet jede Verbindung (a, d) zwischen einer Anwendung a und einem Gerät d einzeln, vgl. Abbildung 4.2a. Eine Verbindung kann in einem der folgenden beiden Modi vorliegen: Entweder ruft die Anwendung a Daten über Anfragen via Unicast-Pull von einem Gerät d ab oder das Gerät überträgt aktualisierte Daten via Unicast-Push an die Anwendung. Die Heuristik verwendet die Gleichungen 4.5 und 4.6, um die Kosten von pull bzw. push-basierter Kommunikation zu vergleichen und schaltet die Verbindung (a, d) in den Modus, von dem erwartet wird, dass er niedrigere Kosten verursacht. Somit verwendet die Heuristik *Unicast Pull*, wenn

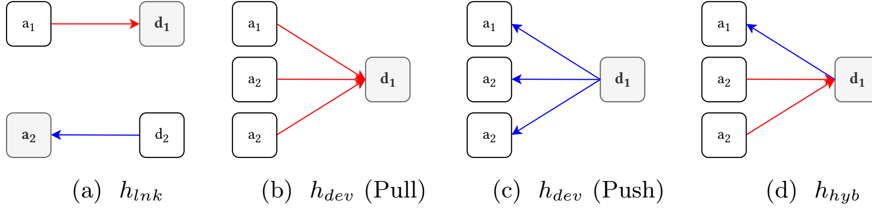
$$C_{UC}^{Pull} + T = 2 \cdot r_a + T < u_d = C_{UC}^{Push}, \tag{4.5}$$

und *Unicast Push*, wenn

$$C_{UC}^{Push} + T = u_d + T < 2 \cdot r_a = C_{UC}^{Pull}, \tag{4.6}$$

wobei T ein konfigurierbarer Schwellenwert einer Hysterese ist, die Schwingungen zwischen beiden Modi entgegenwirkt.

Um die Kosten zu berechnen und zu vergleichen, stellt die Heuristik die Anzahl der Anforderungen und Aktualisierungen in der letzten Beobachtungszeitraum gegenüber: Die Anfragerate (genannt Get-Rate) r_a der Anwendung als auch die Häufigkeit der Datenaktualisierungen (genannt Set-Rate) u_d des Geräts. Da im



(a) h_{lnk} (b) h_{dev} (Pull) (c) h_{dev} (Push) (d) h_{hyb}

Abbildung 4.2: Wirkungsweise der Interaktionsmuster

Pull-Modus Datenaktualisierungen des Gerätes nicht automatisch an die Anwendungsseite weitergeleitet werden, erfolgt die Überwachung der Set-Rate am Gerät d. Dem Gegenüber wird im Push-Modus die Überwachung der Anforderungen (Get-Rate) an der Anwendung durchgeführt, da die Anfragen nicht an das Gerät weitergeleitet werden. Daher erfordert die Heuristik h_{lnk} eine verteilte Implementierung und Auswertung der oben angegebenen Entscheidungskriterien. Auf diese Weise schaltet das Gerät die Verbindung (a, d) vom Pull- in den Push-Modus um, während die Anwendung sie wieder vom Push- in den Pull-Modus umschaltet.

4.4.2 Geräte-Heuristik

Beim Senden von Datenaktualisierungen an mehrere Empfänger ist die Verwendung von Multicast-Kommunikation vorteilhaft. Im Vergleich zu Unicast ist die Multicast-Kommunikation in der Regel umso effizienter, je mehr Empfänger mit einer einzigen Nachricht erreicht werden können. In diesem Zusammenhang verwendet die gerätebasierte Heuristik h_{dev} Multicast anstelle von Unicast, um ihre Datenaktualisierungen bereitzustellen. Basierend auf den durch Gleichungen 4.1 und 4.3 gegebenen Kostenfunktionen schaltet die Heuristik für alle Anwendungen A_d (d.h. für alle dynamischen Mengen, in denen das Gerät d Mitglied ist) um, wenn dadurch der gesamte Netzwerkverkehr bezüglich des Gerätes d reduziert wird, vgl. Abbildung 4.2b und 4.2c. Die Heuristik h_{dev} verwendet für alle Anwendungen A_d *Unicast Pull*, wenn

$$\sum_{a \in A_d} C_{UC}^{Pull} + T = 2 \sum_{a \in A_d} r_a + T < u_d \cdot f(|A_d|) = C_{MC}^{Push}, \tag{4.7}$$

und *Multicast Push*, wenn

$$C_{MC}^{Push} + T = u_d \cdot f(|A_d|) + T < 2 \sum_{a \in A_d} r_a = \sum_{a \in A_d} C_{UC}^{Pull}. \tag{4.8}$$

Grundsätzlich summiert die Heuristik h_{dev} die Anfrageraten (Pull) aller Anwendungen A_d, in deren dynamischer Menge das Gerät Mitglied ist, und vergleicht die Summe der errechneten Pull-Kosten mit den Kosten einer Multicast-

Verbreitung der Datenaktualisierungen u_d an alle dynamischen Mengen A_d, in denen es Mitglied ist. Es ist daher sinnvoll, die Entscheidungskriterien auf Geräteseite auszuwerten. Dies erfordert jedoch, dass während der Multicast-Push-Kommunikation periodisch die Anfragestatistiken (Pull) der Anwendungen an das Gerät übertragen werden, damit dort aktualisierte Werte vorliegen, um auf Schwankungen über einen Beobachtungszeitraum reagieren zu können. Der Schwellenwert T bestimmt die Größe der Hysterese, die Schwingungen zwischen beiden Modi entgegenwirkt.

4.4.3 Hybride Heuristik

Bei stark unterschiedlichen Anfrageraten von verschiedenen Anwendungen an ein Gerät ist es nicht sinnvoll, mit allen Anwendungen mit dem gleichen Kommunikationsparadigma zu kommunizieren, d.h. zu entscheiden, ob Datenaktualisierungen an alle oder keine Anwendung gesendet werden (push). Obgleich Multicast-Push eine Reduktion der Gesamtnetzwerkkosten bewirken kann, kann das Pushen von Informationsaktualisierungen für einige Anwendungen einen signifikanten Overhead verursachen, insbesondere wenn die Set-Rate erheblich höher ist als die Get-Rate der Anwendung. Um diesen Aspekt zu adressieren, wechselt die dritte gerätebasierte Heuristik h_{hyb} nicht ausschließlich zwischen anfragegesteuertem Unicast-Pull und Multicast-Push zur Aktualisierung neuer Daten. Stattdessen verwendet h_{hyb} beide Modi gleichzeitig und klassifiziert jede Anwendung, um sie entweder einer Pull-Gruppe oder einer Push-Gruppe zuzuordnen, je nachdem, welche Kosten für das gegebene Anfrage- bzw. Aktualisierungsmuster insgesamt aus lokaler Perspektive des Gerätes am günstigsten ist, vgl. Abbildung 4.2d. Durch periodische Anpassungen der Gruppenzuordnung kann sich die Heuristik dynamisch an sich ändernde Nachrichtenraten anpassen. Da die Pull- oder Push-Gruppe auch leer sein kann, umfasst die Heuristik h_{hyb} beide Modi der vorherigen Heuristik h_{dev} als Grenzfälle, in denen Datenaktualisierungen entweder an keine oder an alle Anwendungen gepusht werden. Darüber hinaus ermöglicht h_{hyb} feingranulare Konfigurationen, die zwischen diesen beiden Extremen liegen.

Für die Heuristik h_{hyb} besteht die Herausforderung darin, die Gruppenzuweisung zu finden, die am besten zum Anfrage- und Aktualisierungsmuster passt. Daher wird die Menge der Anwendungen A_d, in deren dynamischen Mengen das Gerät d Mitglied ist, in zwei getrennte Gruppen R_d und U_d aufgeteilt, die anfragegesteuerten Unicast-Pull bzw. aktualisierungsgesteuerten Multicast-Push verwenden. Während die jeweiligen Netzkosten nach den Gleichungen 4.1 und 4.3 abgeschätzt werden können, variiert die Heuristik die Gruppenzuordnung, um die Gesamtkosten zu reduzieren, d.h.

$$\min \left\{ C_{UC}^{Pull}(R_d) + C_{MC}^{Push}(U_d) \;\middle|\; \begin{array}{l} R_d \cup U_d = A_d, \\ R_d \cap U_d = \emptyset \end{array} \right\} \tag{4.9}$$

wobei

$$C_{UC}^{Pull}(R_d) + C_{MC}^{Push}(U_d) = 2\sum_{a\in R_d} r_a \quad + \quad u_d \cdot f(|U_d|). \qquad (4.10)$$

Der Algorithmus 1 löst dieses Optimierungsproblem. Die Prozedur PARTITION() erhält als Eingabe die Menge der Anwendungen A_d sowie die erwartete Anzahl von Datenaktualisierungen u_d als Eingabe (Zeile 1). Zunächst werden die Anwendungen nach aufsteigenden Anfrageraten (pull) sortiert (Zeile 2). Danach wird mit einer leeren Pull-Gruppe und einer vollen Push-Gruppe begonnen, die die gesamten Anwendungen enthält. Die entsprechenden Pull- und Pushkosten (Zeilen 3 und 4) werden berechnet und die Ausgangskonfiguration k als beste bisher gefundene Zuordnung C_{min} auf Basis eines Push an alle Anwendungen gespeichert. Die for-Schleife erweitert die Pull-Gruppe $\{a_1, \ldots, a_i\}$ und schrumpft die Push-Gruppe $\{a_{i+1}, \ldots, a_n\}$ um ein Element pro Iteration (Zeile 6). Auf diese Weise wird die Anwendung mit der niedrigsten Anfragerate in der Push-Gruppe in jeder Iteration der Pull-Gruppe neu zugewiesen. Die Pullkosten werden um die Anfragekosten der zusätzlichen Partition erhöht (Zeile 7), Pushkosten werden mit der dekrementierten Anzahl der Anwendungen überschrieben (Zeile 8). Ist diese Konfiguration jene mit niedrigeren Gesamtkosten (Zeile 9), wird sie als aktuelles Minimum gespeichert (Zeile 11). Daher wird nach Abschluss der Iteration über alle Anwendungen die beste Zuweisung für R_d und U_d vorgenommen (Zeilen 14 und 15) und zurückgegeben.

Die zurückgegebene Konfiguration ist in Bezug auf die aktuelle Anfrage- und Aktualisierungsraten optimal. Dennoch werden die Pull- und Push-Gruppen nur dann angepasst, wenn diese Konfiguration die Gesamtkosten deutlich senkt und den Schwellwert T gegenüber der aktuell aktiven Gruppenzuordnung unterschreitet. Auf diese Weise wirkt T Schwingungen entgegen und es wird den Mehrkosten der Kommunikation über den Paradigmenwechsel an ausgewählte Anwendungen begegnet.

Der Algorithmus überprüft nur einen kleinen Bruchteil aller möglichen Konfigurationen. Der zentrale Ansatz besteht darin, die Anwendungen A_d nach ihrer Anfragekosten, d.h. bei Unicast Pull nach ihren Get-Raten, absteigend zu sortieren. Dies ist möglich, da Anwendungen in der Push-Gruppe immer höhere Get-Raten haben als solche in der Pull-Gruppe:

Lemma 1. *Wenn R_d und U_d eine optimale Pull- und Push-Gruppenzuordnung sind (lt zu Gl. 4.9), dann gilt $r_a \leq r_{a'}$ für alle Anwendungen $a \in R_d$ und $a' \in U_d$.*

Beweis. Beweis durch Widerspruch. Angenommen, R_d und U_d sind kostenoptimal, aber $\exists a \in R_d, a' \in U_d : r_a > r_{a'}$. Vertausche nun a und a' so dass $R'_d := (R_d \setminus \{a\}) \cup \{a'\}$ und $U'_d := (U_d \setminus \{a'\}) \cup \{a\}$. Dies reduziert die Pull-Kosten $C_{UC}^{Pull}(R'_d) = C_{UC}^{Pull}(R_d \setminus \{a, a'\}) + 2r_{a'} < C_{UC}^{Pull}(R_d \setminus \{a, a'\}) + 2r_a = C_{UC}^{Pull}(R_d)$, während die Pushkosten $C_{MC}^{Push}(U'_d) = u_d \cdot f(|U'_d|) = u_d \cdot f(|U_d|) = C_{MC}^{Push}(U_d)$ konstant bleiben da sich die Anzahl der Multicast-Empfänger nicht geändert hat $|U'_d| = |U_d|$. Somit war die anfängliche Zuweisung von R_d und U_d nicht minimal, was der obigen Annahme widerspricht.

Algorithm 1 Aufteilung der Anwendungen in die Pull- und Push-Gruppe

1: **procedure** PARTITION($A_d = \{a_1, \ldots, a_n\}$, u_d)
2: $A_d \leftarrow$ **sort**(A_d)
3: $C_{UC}^{Pull} \leftarrow 0$
4: $C_{min} \leftarrow C_{MC}^{Push} \leftarrow u_d \cdot f(n)$
5: $k \leftarrow 0$
6: **for** $i \leftarrow 1$ **to** n **do**
7: $C_{UC}^{Pull} \leftarrow C_{UC}^{Pull} + 2 \cdot a_i.r$
8: $C_{MC}^{Push} \leftarrow u_d \cdot f(n-i)$
9: **if** $C_{UC}^{Pull} + C_{MC}^{Push} < C_{min}$ **then**
10: $k \leftarrow i$
11: $C_{min} \leftarrow C_{UC}^{Pull} + C_{MC}^{Push}$
12: **end if**
13: **end for**
14: $R_d \leftarrow \{a_1, \ldots, a_k\}$
15: $U_d \leftarrow \{a_{k+1}, \ldots, a_n\}$
16: **return** R_d, U_d
17: **end procedure**

4.4.4 Heuristiken mit Multicast Pull

In Abschnitt 4.2.4 wurde Multicast Pull als ein weiteres praktikables Interaktionsmuster vorgestellt, das jedoch noch nicht in den Anpassungsstrategien genutzt wurde. Dafür gibt es mehrere Gründe: Erstens ermöglichen es zwar viele Middleware-Implementierungen den Entwicklern, auf bequeme Weise mit Gruppen von Objekten oder Prozessen zu kommunizieren, aber nur wenige bauen auf eine effiziente netzwerkgestützte Multicast-Kommunikation (z.B. IP-Multicast), da dies andere Aspekte wie die Einrichtung der Konfiguration oder die Fehlerbehandlung zur Gewährleistung eines zuverlässigen Nachrichtentransports erschweren kann. Zweitens ist der Nutzen von Multicast Pull begrenzt. Trotz der Kosteneinsparungen durch die Verwendung von Multicast zur effizienten Übertragung der Datenanforderung an alle Geräte der dynamischen Mengen bestehen die Antworten der Geräte weiterhin aus einzelnen Unicast-Nachrichten. Auf diese Weise können nur die Hälfte der Pull-Interaktion verbessert werden. Und drittens ist es nicht einfach, Multicast-Pull gegen Multicast-Push zu tauschen, da sich die Empfängergruppen unterscheiden, d.h. die Geräte einer dynamischen Menge und die Anwendungen, in deren dynamischer Menge das Gerät Mitglied ist. Außerdem gibt es keine Eins-zu-eins-Korrespondenz zwischen Anwendungen und Geräten, da jede von ihnen auch unterschiedliche Anfrage- und Aktualisierungsraten hat. Um den letzten Aspekt anzusprechen, bedarf es einer Möglichkeit, den Kostenanteil einer Multicast-Nachricht zu bestimmen, die einem einzelnen Empfänger zugeschrieben wird. Dieser Bruchteil wird abgeschätzt, indem die Gesamtkosten der Multicast-Nachricht durch die Anzahl der Empfänger dividiert wird. Für Multicast-Pull und Gleichung 4.4 erhalten wir

$$C_{MC^*}^{Pull}(d) = \frac{r_a \cdot (f(|S_a|) + |S_a|)}{|S_a|} = r_a \cdot \left(1 + \frac{f(|S_a|)}{|S_a|}\right) \tag{4.11}$$

als Kostenanteil für ein einzelnes Gerät d in der dynamischen Menge S_a der Anwendung a für r_a Pull-Anfragen und die entsprechenden Antworten. Darüber hinaus kann diese Schätzung als Ersatz für die Berechnung der Pull-Kosten in der gerätebasierten und der hybriden Heuristik eingesetzt werden. Diese neuen Varianten mit Multicast-Pull sind durch einen Stern kenntlich gemacht und werden h_{dev} bzw. h_{hyb} genannt.

Um die link-basierte Heuristik, welche Multicast keinerlei Berücksichtigung beimisst, zu modifizieren, ist es erforderlich, auch die Push-Seite in die Überlegungen miteinzubeziehen. Ebenso erhalten wir für Multicast-Push und Gleichung 4.3

$$C_{MC^*}^{Push}(a) = \frac{u_d \cdot f(|A_d|)}{|A_d|} = u_d \cdot \frac{f(|A_d|)}{|A_d|} \tag{4.12}$$

als Kostenanteil für u_d Multicast-Datenaktualisierungen, die zu einer einzelnen Anwendung $a \in A_d$ gesendet werden, in derer dynamischen Menge das Gerät d Mitglied ist. Die resultierende Multicast-Variante der Link-basierten Heuristik nennen wir h^*_{lnk}.

Für die Erfassung der Statistik und die korrekte Berechnung der Kosten ist es nun unerlässlich ist zu wissen, an wie viele Empfänger eine Multicast-Nachricht gesendet wird. In der Middleware-Implementierung wird daher diese Informationen jeweils in den Multicast-Anforderungs- und Datenaktualisierungsnachrichten mitgesendet. Dazu bezeichnet R_a die Teilmenge der Geräte, an die eine Anfrage (*request*) von einer Anwendung a über die dynamische Menge S_a mit Multicast-Pull gesendet wurde und U_a die disjunkte Teilmenge (*update*) der Geräte von denen Datenaktualisierungen empfangen werden, wobei gilt $R_a \cup U_a = S_a$ und $R_a \cap U_a = \emptyset$. Analoge Teilmengen werden definiert, ausgehend von der Menge der Anwendungen A_d zu Gerät d, in denen das Gerät Mitglied in der dynamischen Menge ist, entsprechend des verwendeten Kommunikationsparadigmas R_d *(request)* oder U_d *(update)*, vgl. Formel 4.9.

Mit Erweiterung von Multicast-Pull besteht die Notwendigkeit der Anpassung der Heuristiken bei Zählweise der Nachrichtenempfänger unter Berücksichtigung des aktuell verwendeten Kommunikationsparadigmas je Verbindung (a, d).

Mit der Buchführung über die Anzahl der Empfänger einer Multicast-Pull-Nachricht $|R_a|$ – gesendet von Anwendung a sowie die Übermittlung an alle, über die dynamische Menge gebundenen Geräte S_a – muss der evaluierende Klient, d.h. je nach Heuristik und deren Zustand entweder die Anwendung a oder das Gerät d, sicherstellen, dass bei Auswertung der Nachrichtenempfänger auf der Verbindung (a, d) der Empfänger d selbst in der Anzahl der Nachrichten-Empfänger enthalten ist, um die potentiell anwendbaren Pull-Kosten berechnen zu können. Bei Betrachtung der Verbindung (a, d) wird die Menge der Empfänger einer Multicast-Pull-Nachricht R_a, gesendet von einer Anwendung a, mit

dem Gerät $d \in S_a$ vereint $R'_a = R_a \cup d$. Zu einer Multicast-Pull-Nachricht wird nur die Anzahl der Empfänger, jedoch nicht Mitglieder der Menge festgehalten sowie mit einer Pull-Nachricht mitgesendet. Daher inkrementiert der evaluierende Klient bei Betrachtung der Verbindung (a, d) die Anzahl der Empfänger der Multicast-Nachricht, insofern die Anwendung a zu dem Zeitpunkt die Daten von d aktualisiert bekommt, d.h. in Push-Kommunikation gesetzt ist. Damit werden die potentiellen Anfragekosten ($pull$) in der Heuristik genauer abgeschätzt unter der Annahme, dass auch die Verbindung (a, d) in der Menge der Empfänger der Multicast-Pull-Nachricht enthalten sein würde. Die um eins inkrementierte Anzahl der Empfänger lässt die anteiligen Pull-Kosten für die Verbindung (a, d) entsprechend der verwendeten Kostenfunktion mit $f(|R'_a|)$ sinken.

Analog findet die Anpassung der Empfänger einer Multicast-Push-Nachrichten statt, je nach Typ und Zustand der verwendeten Heuristik auf Seite der Anwendung a oder am Gerät d. Hierbei wird vor der Evaluierung in Betrachtung der Verbindung (a, d) die Anzahl der Empfänger einer Multicast-Pushnachricht $|U_d|$, gesendet vom Gerät d, mit der Anwendung $a \in A_d$ vereint $U'_d = U_d \cup a$. Die Anzahl der Empfänger wird um eins inkrementiert, insofern die Verbindung (a, d) aktuell im Pull-Modus kommuniziert, um die potentiellen Pushkosten mit $f(U'_d)$ genauer bestimmen und den aktuellen Pullkosten gegenüberstellen zu können.

Tabelle 4.1 zeigt die vorgestellten Heuristiken in Nachrichtenadressierung mit Unicast- und Multicast-Einsatz und die anwendbaren Kostenabschätzungen, welche mit den Anfrage- bzw. Aktualisierungsraten multipliziert werden. Die verteilte verbindungs-orientierte Heuristik h_{lnk} evaluiert je nach Kommunikationsparadigma der Verbindung (a, d) auf Anwendungs- (a) oder Geräteseite (d). In der Ausprägung Multicast werden die um eins inkrementierten Empfängermengen des Nachrichtentyps berücksichtigt, dass sich die Verbindung aktuell im gegenteiligen Kommunikationsparadigma befindet. Die geräte-basiert Heuristik h_{dev} berücksichtigt bei der Aufsummierung der Multicast-Pullkosten auf Geräteseite die aktualisierte Empfängermenge der Anfrage, d.h. der inkrementierten Empfängeranzahl, insofern die Verbindung aktuell im Push-Modus konfiguriert ist.

Die nicht in der Tabelle aufgeführte hybride Heuristik h^*_{hyb} übernimmt die Bewertung der Multicast-Pullkosten vergleichbar wie bei h^*_{dev}. Der Algorithmus 1 zur Partitionierung der Anwendungsmenge eines Gerätes wird für die Verwendung von Multicast-Pull angepasst: Die Sortierung der mit dem Gerät assoziierten Anwendungen A_d erfolgt absteigend auf Basis nun vorab berechneten Pullkosten entsprechend Anfragerate und der konfigurierten Kostenfunktion unter Berücksichtigung eines eventuellen Inkrement der übermittelten Empfängeranzahl der Multicast-Pullnachrichten, insofern die Verbindung mit dem Gerät, an dem evaluiert wird, im Push-Modus konfiguriert ist. Anschließend wird unverändert für jede Partitionierung mit zunehmender Anzahl an anfragenden Anwendungen und abnehmender Anzahl von zu aktualisierenden Anwendungen die Summe aus Multicast-Pull- und Push-Kosten berechnet, um die optimale Partition mit den minimalen Gesamtkosten zu erhalten.

H	Adressierung a	d	Kommunikation $P(a,d)$	eval. Klient	Kosten Pull $r_a\cdot$	Push $u_d\cdot$
h_{lnk}	UC	UC	Pull	d	2	1
			Push	a		
h^*_{lnk}	MC	MC	Pull	d	$\frac{f(\|R'_a\|)}{\|R'_a\|}$	$\frac{f(\|U'_d\|)}{\|U'_d\|}$
			Push	a		
h_{dev}	UC	MC		d	$\sum_{a\in A_d} 2$	$f(\|A_d\|)$
h^*_{dev}	MC				$\sum_{a\in A_d} \frac{f(\|R'_a\|)}{\|R'_a\|}$	

Tabelle 4.1: Evaluierung Heuristiken

4.4.5 Werteglättung

Jeder Stellvertreter hält Datenstrukturen, um die lokalen Interaktionen des Klienten und solche der Gegenstelle zählen und einer Heuristik zuzuführen zu können. Der Proxy einer dynamischen Menge enthält eine Zähler-Datenstruktur für die lokalen Anfragen der zugrundeliegenden Anwendung a vor, welche an Geräte repliziert werden (Pull), sowie eine Menge von $|S_a|$ Datenstrukturen zur Aufnahme der aktualisierten Daten von den gebundenen Geräten (push). Analog hält der Stellvertreter eines Gerätes d die gespiegelten Datenstrukturen vor, hier eine für die lokalen Datenaktualisierungen des zugrundeliegenden Gerätes (Push) sowie eine Menge mit $|A_d|$ Einträgen zum Zählen der Anfragen von den Anwendungen.

In den Zähler-Datenstrukturen wird für jede Interaktion mit einer Methode an einer in der dynamischen Menge oder am Gerät definierten Schnittstelle ein Eintrag angelegt. Die Einträge der Zähler-Datenstrukturen nehmen verschiedene Werte auf: Anzahl der Aktionen und Anzahl der Empfänger je Aktion. Des Weiteren sind Nutzdaten relevant. So werden vom Gerät via Push aktualisierte Rückgabewerte einer Schnittstellenmethode vorgehalten und bei Folgeanfragen aus einer Anwendung aus dem lokalen Zwischenspeicher zurückgegeben.

Um einen Zählerüberlauf zu verhindern, werden Höchstgrenzen für die Statistiken der einzelnen Zähler festgelegt. Es ist jedoch auch erforderlich, dass die Zählerstände im Laufe der Zeit abnehmen und mit fortschreitender Zeit an Bedeutung verlieren. Für dieses Zweck wird exponentielles Glätten angewendet. Der aktuelle Zählwert wird durch eine rekursive Berechnung mit der folgenden Formel ermittelt:

$$x^*_t = \alpha \cdot x_t + (1-\alpha) \cdot x^*_{t-1} \qquad 0 \leq \alpha \leq 1 \tag{4.13}$$

x^*_t ist der neue geglättete Zählstand, α der Glättungsfaktor zwischen 0 und 1, der auf die gezählten Ereignisse x_t seit dem letzten Glätten angewendet wird

und x^*_{t-1} alte davor geglättete Zählstand. Die Auflösung der Rekursion ergibt die folgende Formel:

$$x^*_t = \alpha \cdot x_t + \alpha(1-\alpha) \cdot x_{t-1} + \alpha(1-\alpha)^2 \cdot x_{t-2} + \dots \qquad 0 \leq \alpha \leq 1 \qquad (4.14)$$

Der Einfluss früherer Zählstände verringert sich im Laufe der Zeit zunehmend [160, 165]. Es lässt sich mithilfe eines Zählers, der die Anzahl der Bewertungen vorhält, bestimmen, wie häufig Vergessen wird. Es kommt jedoch vor, dass die Zählerstände bei jedem Klienten zu unterschiedlichen Zeitpunkten vergessen werden, wenn immer nach einer bestimmten Anzahl an Evaluierungen vergessen wird. Andererseits werden auch die Zählerstände von Datenproduzenten und -konsumenten unterschiedlich oft vergessen. Allerdings haben diese Probleme beim Unicast keine negativen Auswirkungen. Aufgrund der individuellen Behandlung jedes Datenproduzenten stellt es kein Problem dar, dass Konsumenten zu unterschiedlichen Zeitpunkten vergessen werden.

4.5 Umsetzung

Die Implementierung der vorgestellten Heuristiken unter Berücksichtigung von Statistiken bedarf einiger Prozeduren. Zuerst müssen die Aufrufe oder Aktualisierungen in lokalen Statistiken an Anwendungen oder Geräten gehalten werden. Zur Wartung der Statistiken sowie Abfederung von Ausreißern (Burst durch höhere Frequenzen) werden die Werte regelmäßig geglättet und normalisiert.

4.5.1 Aufrufe

Wird von einer Anwendung a ein Methodenaufruf an den Stellvertreter der dynamischen Menge gestellt, greifen die Heuristiken für die lesenden Methoden (getter). Die lokale Anfragestatistik (Pull) zu Frequenzzählung (*actions*) wird um eins inkrementiert, vgl. localStats Abschnitt 4.5.2. Entsprechend der Teilmenge der gebunden Geräte S_a, an die die Anfrage repliziert wird, wird deren Anzahl in die Anfragestatistik zu den Empfängern (*destinations*) erhöht. Bei einem Gerät d empfangene Anfragen werden dort in der Verbindungsstatistik gezählt, vgl. linkStats Abschnitt 4.5.2. Die Aktionen (*actions*) werden durch den eingehenden Pull um eins erhöht. Als Metadaten wird der Anfrage die Anzahl der Empfänger mitgeschickt, welche zur entsprechenden Statistik (*destinations*) auf Geräteseite aufaddiert wird.

Für Datenaktualisierungen werden die Statistiken zum Zählen von Aktualisierungen und Empfängern für lesende Methoden (getter) gespiegelt angesprochen: Eine Datenaktualisierung am Gerät d inkrementiert dort in localStats die Anzahl der Aktionen und der Empfänger aus der Teilmenge A_d, an die Anwendungen

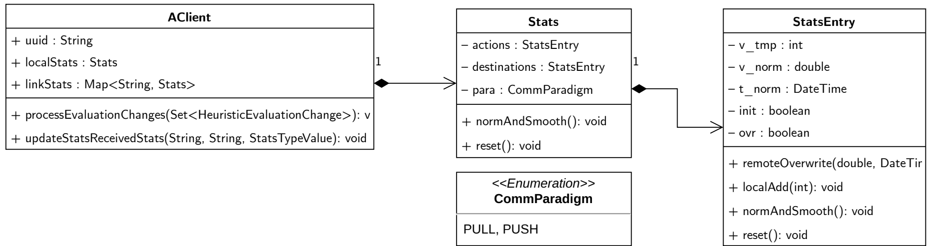


Abbildung 4.3: UML-Klassendiagramm, Zählstatistiken

an die gepushed wird. Bei Empfang einer Datenaktualisierung auf der Anwendungsseite a wird dort in der linkStats für die Verbindung mit dem sendenden Gerät die Statistik für die Frequenz um eins erhöht und aus den Metadaten der Datenaktualisierung die Anzahl der Empfänger des aktuellen Aufrufs der korrespondierenden Statistik hinzugefügt.

4.5.2 Statistiken

In jedem Klienten existierende Datenstrukturen halten i) die lokalen Zählstände und ii) die eingehenden Anfragen bzw. Aktualisierungen der Kommunikationspartner vor. Die im vorherigen Kapitel vorgestellte Programmierabstraktion soll eine adaptive Umschaltung zwischen pull- und push-basierter Kommunikation ermöglichen, um bei Integration in eine Anwendung vergleichbar eines lokalen Objektes die entfernten idempotenten Methodenaufrufe zu optimieren, dass die Netzlast reduziert wird. Die Granularität der Adaptivität kann auf Typ- oder Methodenebene der Schnittstelle konfiguriert werden, erfordert dann eine unterschiedliche Adressierung und Verarbeitung der Datenstrukturen der Zählstatistiken.

Abbildung 4.3 zeigt das UML-Klassendiagramm mit den notwendigen Klassen. Ein abstrakter AClient in konkreter Ausprägung eines Datenproduzenten oder -konsumenten (im Motivationsbeispiel: Gerät sowie dynamische Menge in einer Anwendung), instanziiert für die lokalen Zählungen eine Stats sowie eine Kollektion für die Verbindungen zu den entfernten Klienten entsprechend der Mengenbildung, welche über einen eindeutigen Identifikator als Schlüssel ansprechbar sind. Für eine dynamische Menge, d.h. Datenkonsument, zählt die lokale Statistik localStats die Anfrageraten (pull) von der Anwendung, die Kollektion mit Verbindungs-Statistiken linkStats jeweils die eingehenden Aktualisierungsraten (push) für jeden Datenproduzenten, d.h. Gerät, welches Mitglied der dynamischen Menge ist. Für Stellvertreter von Geräten stehen localStats und linkStats gespiegelt für eigene Aktualisierungsraten vom Gerät (push) und den jeweiligen Anfrageraten von dynamischen Mengen (pull), welches dieses Gerät als Mitglied aufgenommen haben. Jede Stats Instanz hält zwei StatsEntry-Attribute, um die Aktionen (actions) und die Anzahl der damit adressierten Empfänger

(destinations) aus der Menge der Mitglieder zu zählen, was später für die anteiligen Kosten einer Verbindung bei Anwendung von Multicast von Heuristiken verrechnet wird. Dem Feld para ist entweder der Aufzählungswert PULL oder PUSH zugewiesen. Je nach angewendeter Heuristik, Art des Klienten und umgebener Datenstruktur hält para den aktuellen Zustand des Kommunikationsparadigmas für eine Verbindung oder einen Klienten vor. Beispielsweise hält bei einem Geräte-Stellvertreter für die lokalen Aktualisierungsraten und angewendeter Geräte-basierter Heuristik h_{dev} die Kommunikationsart mit allen dynamischen Mengen vor, während bei angewendeter hybrider Heuristik h_{hyb} die Kommunikation verbindungs-individuell auf Geräte-Seite bestimmt in jeweiligen Verbindungsstatistiken zu einer dynamischen Menge vorgehalten wird.

Des Weiteren werden Normierungs- und Glättungsoperationen an jedem Klienten durchgeführt. Wären die Zeitpunkte von Evaluierungen systemweit an allen Klienten synchronisiert und Ankunftszeiten der Nachrichten mit dem Austausch von Statistiken garantiert, bestünde keine Notwendigkeit einer Normierung. Jedoch können Klienten mit unterschiedlichen Intervallen initial konfiguriert sein, wann eine Evaluierung stattfinden soll. Für die Verrechenbarkeit mit Statistiken anderer Klienten bedarf es daher eine Wertenormierung auf einen Zeitintervall, der zusammen mit dem Statistikwert über das Netzwerk kommuniziert wird. Des Weiteren können Heuristiken für die Reduktion von Kontrollnachrichten einen Sende-Schwellwert konfigurieren, ob nach Evaluierung überhaupt eine Statistikaktualisierung gesendet werden soll oder nicht. Auch die Evaluierungszeitpunkte können statisch individuell konfiguriert werden bzw. dynamisch entsprechend der Frequenz der Aktionen ausgelöst werden. Ist der Betrag zwischen dem aktuellen und dem zuletzt gesendeten Wert kleiner als der Schwellwert, wird die Statistik nicht gesendet.

Algorithmus 2 zeigt mit Pseudocode die elementare Prozedur der Werteevaluierung auf. Für lokal ausgelöste Ereignisse (Ereignis und Empfängeranzahl) wie a) get() an einer Anwendung a und b) set() an einem Gerät d sowie c) an einem Gerät gezählte ankommende Anfragen (pull) und d) an einer Anwendung gezählten eingehende Aktualisierungen (push) werden über die Prozedur (add()) gezählt und die Statistik als nicht überschrieben (ovr) markiert. Im Fall eines Statistikaustauschs entsprechend der Logik der Heuristik werden bereits normierte Werte über das Netzwerk gesendet und dürfen beim Empfänger nicht erneut normiert werden. Dazu wird die Prozedur overwrite() angesprochen, die vorhanden Statistikwerte für die vorgenannten Fälle c) oder d) überschreibt und den Normierungszeitpunkt am Sender setzt. Der Normierungszeitpunkt ist im Falle eines Wechsels des Kommunikationsparadigmas notwendig, wenn zukünftig lokale Werte gezählt werden, für deren Normierung der Zeitabstand zur letzten Normierung eingeht. Die Prozedur NormAndSmooth setzt die Normierung und Glättung um. Nur für lokal gezählte Werte wird normalisiert. Bei Folgeaufrufen werden die Werte exponentiell geglättet unter Hinzunahme des Vorgängerwertes. Abschließend wird der Zähler v_{tmp} für die Werte des kommenden Betrachtungszeitraumes zurückgesetzt und die letzte Normierungszeit auf die Aktuelle gesetzt.

Algorithm 2 Werteverarbeitung in der Statistik

```
 1: procedure RESET
 2:     v_tmp, v_norm ← 0                                  ▷ Werte
 3:     init ← true, ovr ← false                          ▷ initialisiert, überschrieben
 4:     t_norm ← t_now                                     ▷ letzte Normalisierung
 5: end procedure
 6: procedure ADD(v)
 7:     v_tmp +← v
 8:     ovr ← false
 9: end procedure
10: procedure OVERWRITE(t,v)
11:     t_norm ← t
12:     v_norm ← v, v_tmp ← 0
13:     init ← false, ovr ← true
14: end procedure
15: procedure NORMANDSMOOTH
16:     if not ovr then                                    ▷ nur bei lokaler Zählung
17:         v_calc ← v_tmp/(t_now − t_norm)                ▷ Normalisierung
18:         if init then                                   ▷ Erstaufruf
19:             v_norm ← v_calc
20:             init ← false
21:         else                                           ▷ Wiederholungsaufruf
22:             v_norm ← α * v_calc + (1 − α) * v_norm     ▷ Exponentielles Glätten
23:         end if
24:         v_tmp ← 0                                      ▷ Zurücksetzen lokaler Zähler
25:         t_norm ← t_now                                 ▷ Aktualisierung Normalisierungzeit
26:     end if
27: end procedure
```

4.5.3 Evaluierung

Die Auswertung der Statistiken und Entscheidung durch die eingesetzte Heuristik wird durch einen eigenständigen Thread wiederholt gestartet; je nach Konfiguration in definierten Intervallen und/oder bei Aktualisierung der Statistiken nach einer konfigurierten Anzahl von Aktualisierungen. Abbildung 4.4 zeigt ein Sequenzdiagramm mit den beteiligten Objekten für die Auswertung der linkbasierten Heuristik h_{lnk}/ h^*_{lnk} auf der Geräteseite d. Das Device, geerbt von AAdaptiveClient, ist aus der Programmierabstraktion der dynamischen Mengen in blau (Kapitel 3) und registriert sich beim ClientEvaluator, welcher Auswertungen überwacht und initialisiert. Mit der Referenz vom Klienten wird über die abstrakte Klasse AHeuristics die evaluateDevice()-Methode der LinkHeuristics aufgerufen und dort für alle Anwendungen $a \in A_d$ durch Aufruf von evaluateLink(a,d) die kostengünstigste Kommunikationsart unter Respektierung eines Schwellwertes ermittelt, vgl. Abschnitt 4.5.4. Die Rückgabe enthält eine

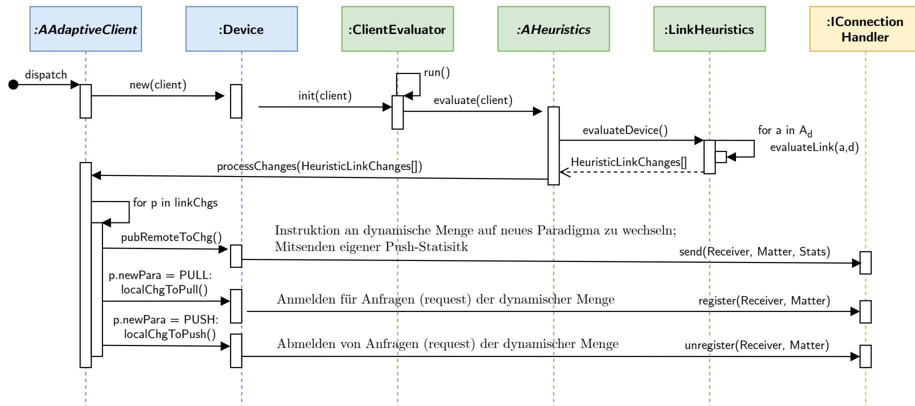


Abbildung 4.4: Sequenzdiagramm, Evaluierung der verbindungsbasierten Heuristik auf Geräteseite

Menge mit Änderungseinträgen, wo sich das Kommunikationsparadigma ändert. Bei der linkbasierten Heuristik und Auswertung auf Geräteseite sind dies nur die Kandidaten, welche von Pull auf Push umgeschaltet werden sollen. Die Datenstruktur enthält Adressierungen von Sender (hier Gerät) und Empfänger (hier Anwendung), Ziel-Kommunikationsparadigma für die Verbindung (hier Push), sowie die lokalen Statistiken des Senders (hier Push-Statistiken vom Gerät) zum Überschreiben auf Anwendungsseite. Die Änderungsmenge wird mit Aufruf von processChanges() am Klienten übergeben und dort verarbeitet. An dem Beispiel der Geräteseite d für die linkbasierte Heuristik h_{lnk} werden die Änderungseinträge verarbeitet und mit pubRemoteToChg() jeweils die Anwendung a über den ConnectionHandler aus dem Netzwerktransport (gelb) instruiert auf Push-Kommunikation zu wechseln. Mit der Instruktion werden die Push-Statistiken des Gerätes übermittelt. Des Weiteren wird lokal die Geräteseite d für die Push-Kommunikation über die Verbindung zu Anwendung a angepasst. Dazu wird über den ConnectionHandler die Abmeldung von Anfragenachrichten von der Anwendung a gesendet, da dies zuvor angewiesen wurde sich für Datenaktualisierungen zu registrieren.

4.5.4 Algorithmen

Die in Abschnitten 4.4.1–4.4.3 vorgestellten Heuristiken werden algorithmisch erweitert, um die zuvor vorgestellten Statistiken mit Normierung und Glättung der Zählwerte zu berücksichtigen. In den folgenden Unterabschnitten werden die Evaluierungsprozeduren der drei Heuristiken ausschnittsweise in Pseudocode dargestellt, um die Funktionsweise aufzuzeigen. Tabelle 4.1 stellt für die Heuristiken h_{lnk} und h_{hyb} die anwendbaren Pull- und Push-Kosten dar unter Berücksichtigung der Adressierungsart der Empfänger sowie welcher Klienttyp in welchem Kommunikationsparadigma für Evaluierung zuständig ist.

Algorithm 3 Link-basierte Heuristik h_{lnk}, Evaluation auf der Anwendungsseite

1: S_{local} ▷ Anfragestatistik (lokaler Pull von Anwendung)
2: $S_{link}[]$ ▷ Aktualisierungsstatistiken (Push von Gerät je Verbindung)
3: **function** EVALUATEAPP
4: $R \leftarrow \emptyset$ ▷ Paradigmenwechsel
5: S_{local}.normalizeAndSmooth() ▷ Normalisierung und Glättung
6: **for** s in S_{link} **do**
7: **if** $s.p ==$ `PUSH` **then** ▷ Anwendung evaluiert nur wenn Link im Push
8: s.normalizeAndSmooth() ▷ Normalisierung und Glättung
9: $p_{new} \leftarrow evalLink(s, S_{local}, s.p, \texttt{PULL})$ ▷ Eval. der Verbindung
10: **if** $p_{new} ==$ `PULL` **then** ▷ Paradigmenwechsel
11: $R \overset{+}{\leftarrow} (t_{now}, ID_{dev}, p_{new}, S_{local})$ ▷ Eintrag zu Änderungsmenge
12: **end if**
13: **end if**
14: **end for**
15: **return** R
16: **end function**

Link Die linkbasierte Heuristik h_{lnk} betrachtet nur die Kosten einer Verbindung zwischen Anwendung und Gerät (a, d). Ist diese Verbindung im Pull, ist das Gerät d für die Evaluierung der Kosten für die Verbindung zuständig; im Push evaluiert die Anwendungsseite a.

Algorithmus 3 zeigt die Evaluierung auf Anwendungsseite, die gespiegelt auf Geräteseite vergleichbar anzuwenden ist. S_{local} hält die Statistiken zu lokalen Pull-Frequenzen von der Anwendung r_a sowie die Anzahl der adressierten Nachrichtenempfänger. Die Menge $S_{link}[]$ hält für jede Verbindung zu einem Gerät die Push-Statistiken mit dessen Aktualisierungen u_d, dass Mitglied der dynamischen Menge ist. Zu Beginn der Evaluierung werden die lokalen Anfrage- und Adressierungszähler (Pull) normiert und geglättet, wie in Abschnitt 4.5.2 beschrieben. Anschließend werden alle Verbindungsstatistiken überprüft, ob die Verbindung im Push-Paradigma vom Gerät kommuniziert. Nur dann ist die Anwendungsseite für die Evaluierung verantwortlich und die Push-Statistiken liegen durch Statusaktualisierungen vom Gerät bei der Anwendung vor. Push-Aktionen und -Empfängerzähler werden ebenfalls normiert und geglättet. Anschließend wird die an beiden KlTypen anwendbare Evaluierungsmethode evalLink() mit Pull- und Push-Statistiken aufgerufen, dem aktuellen Kommunikationsparadigma sowie welche der beiden Statistiken als lokal zu betrachten ist. Die Methode gibt entweder `PULL` oder `PUSH` zurück, welches die günstigeren Kosten für den Link ergibt. Im Falle eines Paradigmenwechsels zu `PUSH` wird ein Eintrag mit aktuellem Zeitstempel, Geräte-ID, neuem Paradigma für den Link sowie einem Extrakt der lokalen Pull-Statistik der Änderungsmenge hinzugefügt. Dieser Eintrag wird nachfolgend als Kontrollnachricht an das Gerät gesendet, so dass in der Pull-Statistik für die Verbindung auch das Paradigma aktualisiert und die nor-

mierten Statistikwerte sowie der zuletzt angewendete Zeitpunkt überschrieben werden kann.

Auf der Geräte-Seite wird der vorherige Algorithmus 3 gespiegelt angewendet: S_{local} hält die Statistiken zu lokalen Set-Aktualisierungen und die jeweils aufaddierte Anzahl der Empfänger. Die weitere Evaluierung findet nur statt, wenn die Verbindung im `PULL` kommuniziert. Nach Evaluierung wird bei Wechsel zu `PUSH` ein Eintrag in die Änderungsmenge erstellt, welcher in eine Kontrollnachricht an die Anwendung weitergeführt wird.

Der Algorithmus 4 zeigt die Kostenbildung und -vergleiche für eine Verbindung anhand der Pull- und Push-Statistiken S_{pull} und S_{push} auf. Je nach Aufruf der Funktion auf Anwendungs- oder Geräteseite werden die Statistiken als *local* oder *link* im Aufruf zugewiesen. Exemplarisch ist folgend die Bildung der Pull-Kosten aufgezeigt; die Push-Kosten werden gespiegelt gebildet. Ist Unicast als Nachrichtenadressierung in der Heuristik konfiguriert, ergeben sich die Kosten aus dem Produkt der normierten, geglätteten Pull-Aktionen $S_{pull}.actions$ und den statischen Kosten für einen Unicast-Request. Wird Multicast verwendet, werden zunächst aus der Division der aufaddierten Anzahl der Nachrichtenempfänger und der Anfragen die durchschnittlichen Empfänger der Multicast-Nachrichten gebildet auf Basis der normierten und geglätteten Werte. Werden über den Parameter P_{local} die Pull-Statistiken als lokale Statistikart definiert, wurde die Funktion auf Anwendungsseite a aufgerufen und der zu evaluierende Link ist im `PUSH`-Modus. Die durchschnittlichen Empfängeranzahl wird in diesem Fall zuvor um eins inkrementiert, d.h. die Adressierung würde den eigenen Link mit für ein Multicast-Pull betrachten, der jedoch aktuell im Push kommuniziert. Die Empfängeranzahl wird als Parameter an die konfigurierte Kostenfunktion übergeben, welche die anteiligen Kosten der Verbindung (a, d) für einen Multicast-Pull zurückgibt. Diese werden zur Berechnung der Pull-Kosten C_{pull} für die Verbindung mit der Anzahl der Aktionen $S_{pull}.actions$ im Betrachtungszeitraum multipliziert. Die Push-Kosten werden in der Funktion vergleichbar aus dem zweiten Parameter gebildet. Anhand der beiden erreichten Kosten für die Verbindung wird dynamisch ein Schwellwert aus der Konfiguration der Heuristik errechnet. Ausgehend von dem übergebenen Parameter P_{before} für das aktuelle Kommunikationsparadigma werden diese Kosten zuzüglich des Schwellwertes mit den gegensätzlichen Kosten verglichen. Die Funktion gibt das Paradigma der preiswerteren Kostenart für die Verbindung (a, d) an den Aufrufer zurück.

Gerät Algorithmus 5 zeigt im Pseudocode die Evaluierung der geräte-absoluten Heuristik h_{dev} auf Geräteseite d auf. Unabhängig von den angewendeten Paradigmen je Verbindung wird bei dieser Heuristik nur auf der Geräteseite für alle Verbindungen A_d ausgewertet. Das aktuell angewendete Kommunikationsparadigma wird in der lokalen Statistik vorgehalten. Sind die Verbindungen im Push-Modus, senden Anwendungen entsprechend der Konfiguration wiederkehrend Pull-Statistiken, damit beide Statistikarten zur Errechnung und Vergleich der Kosten am Gerät vorliegen. Die Kosten werden für alle Verbindungen von

Algorithm 4 Link-basierte Heuristik, Evaluation einer Verbindung

1: **function** EVALUATELINK(S_{pull}, S_{push}, P_{before}, P_{local})
2: $C_{pull} \leftarrow S_{pull}.actions * costs.UCPullLink()$ ▷ Errechne Pull-Kosten, Unicast
3: **if** $H.pullCast = \text{MC}$ **then** ▷ Heuristik fragt über Multicast an
4: $avgPullDests \leftarrow S_{pull}.destinations/S_{pull}.actions$
5: **if** p_{local} = `PULL` **then**
6: $avgPullDests \overset{+}{\leftarrow} 1$ ▷ Inkr. für mögliches Pull der Verbindung
7: **end if**
8: $C_{pull} \leftarrow S_{pull}.actions * costs.MCPullLink(avgPullDests)$
9: **end if**
10:
11: $C_{push} \leftarrow \ldots$ ▷ Push-Kosten der Verbindung äquivalent zu Pull
12:
13: $T \leftarrow calcTreshold(C_{pull}, C_{push})$ ▷ Schwellwert basierend auf Kosten
14:
15: **if** (P_{before} = `PULL` and $C_{push} + T < C_{pull}$) or (P_{before} = `PUSH` and $C_{pull} + T > C_{push}$) **then**
16: **return** `PUSH`;
17: **end if**
18: **return** `PULL`;
19: **end function**

dem Gerät d zu den registrierten Anwendungen A_d betrachtet und gleich entschieden.

Für die Pullkosten wird über alle Verbindungen iteriert und Teilergebnisse aufaddiert. Ist die Verbindung im Pull-Modus, wurden die Statistiken der Verbindung durch Anfragen auf Geräteseite gezählt und werden vorab normalisiert und exponentiell geglättet. Anschließend wird die durchschnittliche Empfängeranzahl der Pulls aus der aufaddierten Empfängeranzahl geteilt durch die aufaddierten Aktionen gebildet. Wird Multicast verwendet ist die Empfängeranzahl größer gleich der Aktionen, bei Unicast sind beide Werte gleich und lösen zu maximal eins auf. Sind die Verbindungen im Push, werden jeweils die durchschnittlichen Empfänger um eins erhöht, da mögliche Anfragekosten abgeschätzt werden sollen, würden die Verbindungen im `PULL` das Gerät anfragen. Vor Aufaddieren der Teilergebnisse zu den Gesamt-Pullkosten für das gesamte Gerät wird die Anfrageanzahl multipliziert mit den Anfragekosten für die Verbindung (a, d), welche aus der konfigurierten Kostenfunktion unter Berücksichtigung der ermittelten Empfängerzahl gebildet wird.

Die Push-Kosten für das gesamte Gerät werden ebenfalls durch Faktorisierung des lokal vorliegenden, normalisierten und exponentiell geglätteten Zählers sowie der Kosten eines Multicast Push an alle registrierten Anwendungen gebildet.

Algorithm 5 Evaluierung der Gerät-absolute Heuristik h_{dev}

1: $R \leftarrow \emptyset$ ▷ Paradigmenwechsel
2: S_{local} ▷ Aktualisierungen (lokaler Push von Gerät)
3: $S_{link}[]$ ▷ Anfragen (je Verbindung Pull von Anwendungen)
4: **for** s in S_{link} **do** ▷ Pull-Kosten: Aufaddieren aller Verbindungen
5: **if** $s.para ==$ `PULL` **then**
6: s.normalizeAndSmooth()
7: **end if**
8: $avgPullDests \leftarrow s.destinations/s.actions$
9: **if** $S_{local}.para =$ `PUSH` **then** ▷ Gerät pushed derzeit an alle
10: $avgPullDests \overset{+}{\leftarrow} 1$ ▷ Inkrementieren für mögliches Pull der Verbindung
11: **end if**
12: $C_{pull} \overset{+}{\leftarrow} s.actions * costs.MCPullLink(avgPullDests)$
13: **end for**
14: ▷ Multicast Push-Kosten Geräte-global an alle Anwendungen
15: $C_{push} \leftarrow S_{local}.actions * costs.MCPushLink(|S_{link}|)$
16: $T \leftarrow calcTreshold(C_{pull}, C_{push})$ ▷ Schwellwertbildung
17: ▷ Paradigmenwechsel entscheiden und bereitstellen
18: **if** $P_{before} =$ `PULL` and $C_{push} + T < C_{pull}$ **then** ▷ Pull → Push
19: **for** s in S_{link} **do**
20: $s.para \leftarrow$ `PUSH`
21: $R \overset{+}{\leftarrow} (t_{now}, ID_{dev}, \texttt{PUSH}, s)$
22: **end for**
23: **else if** $P_{before} =$ `PUSH` and $C_{push} \geq T + C_{pull}$ **then** ▷ Push → Pull
24: **for** s in S_{link} **do**
25: $s.para \leftarrow$ `PULL`
26: $R \overset{+}{\leftarrow} (t_{now}, ID_{dev}, \texttt{PULL}, s)$
27: **end for**
28: **end if**
29: **return** R

Nach Bildung eines Schwellwertes wird unter Berücksichtigung dessen entschieden, welche Kosten geringer sind und eine Kollektion den Angaben zu Paradigmenänderungen für Verbindungen $(a_i, d)|a_i \in A_d$ gebildet und an den Aufrufer zurückgegeben, vgl. Algorithmus 3 zur linkbasierten Heuristik h_{lnk}.

Hybrid Die Umsetzung der hybrid-basierten Heuristik h_{hyb} folgt den drei Stufen des vorstellten Algorithmus zur Partitionierung der Verbindungen von registrierten Anwendungen zu einem Gerät A_d in disjunkte Teilmengen zur Anfrage und Aktualisierung. Die Auswertung findet immer auf Geräteseite statt und konfiguriert so dass hier verbindungsindividuell das Kommunikationsparadigma nach der Partition mit den geringsten Gesamtkosten.

Zuerst werden die Pullkosten anhand der Statistiken je Verbindung berechnet. Ist eine Verbindung im Pull-Modus, werden die Statistiken normiert und exponentiell geglättet; im Push-Modus jedoch nicht, da normierte Pull-Statistiken für die Verbindung über Kontrollnachrichten vom Gerät wiederholt gesendet und überschrieben werden. Aus den Statistiken werden die durchschnittlichen Empfänger eines Multicast-Pull errechnet. Ist die zu betrachtende Verbindung im Push-Modus, wird die um eins erhöht, um eine mögliche Umschaltung der zu betrachtenden Verbindung nach Pull zu berücksichtigen. Die für den Link anteiligen Pull-Kosten auch hier durch Multiplikation der Aktionen und dem Ergebnis der Kostenfunktion für Multicast-Pull berechnet. Die errechneten Pullkosten je Verbindung zu registrierten Anwendungen wird absteigend sortiert. Die lokalen Pushkosten an alle registrierten Anwendungen wird durch Normalisierung, exponentieller Glättung und Anwendung der Kostenfunktion wie in der gerätebasierten Umsetzung zuvor beschrieben.

In der zweiten Phase wird beginnend von der Konfiguration in der sich alle Verbindungen im Push-Modus befinden bis hin zu jeder wo alle Verbindungen im Pull-Modus kommunizieren die Kostenzusammensetzungen durch abnehmende Push- und zunehmende Pull-Teilnehmer bestimmt. Dabei werden Kosten je Iteration neu bestimmt: Der Kostenfunktion für einen Multicast-Push wird eine dekrementierende Empfängeranzahl übergeben. Die Pullkosten, mit null initialisiert, werden die Kosten der nächstteureren Pull-Verbindung aufaddiert. Ist die Summe beider Kosten in der Partition als eine vorher ermittelte, wird die Partition als Änderungskandidat bestimmt.

In der letzten dritten Phase wird hier ebenso ein Schwellwert gebildet, jedoch aus den Kosten der zuletzt kommunizierten und der aktuellen Änderungskonfiguration. Sind die neu errechneten Minimalkosten unter den Schwellwert zu den gespeicherten Minimalkosten der letzten Änderung, findet eine Neuaufteilung der Verbindungen A_d statt, deren Statistiken werden ggf. mit dem neuen Kommunikationsparadigma überschrieben und bei Änderung ein Eintrag in die Änderungsmenge erstellt, wie zuvor bei der link-basierten Heuristik beschreiben, erstellt und diese dem Aufrufer zurückgegeben. Die Minimalkosten der aktuellen Evaluierung überschreiben jene der vorherigen.

4.6 Evaluation

In diesem Abschnitt wird die Leistung des adaptiven Ansatzes bewertet. Die Bewertung basiert auf diskreten Ereignissimulationen und vergleicht die drei Heuristiken h_{lnk}, h_{dev} und h_{hyb} miteinander und gegenüber statischen Kommunikationsschemata, die entweder Unicast Pull ($Pull_{UC}$) oder Unicast Push ($Push_{UC}$) verwenden. h_{lnk} wechselt für jede Verbindung (d.h. für jedes Paar aus Anwendung a und Gerät d) zwischen Unicast Pull und Unicast Push, h_{dev} wechselt für jedes Gerät zwischen vollständigem Unicast Pull und vollständigem Multicast Push $Push_{MC}$ und h_{hyb} verwendet für jedes Gerät sowohl partiellen

Unicast Pull als auch partiellen Multicast Push. Darüber wird auch jede der drei Heuristiken in einer Variante vorgestellt, die nur Multicast-Kommunikation verwendet (h^*_{lnk}, h^*_{dev} bzw. h^*_{hyb}).

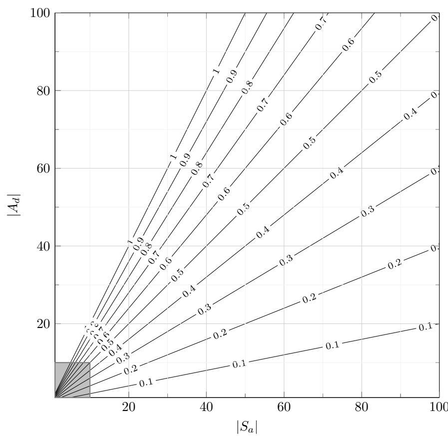
Von den in Abschnitt 4.2 vorgestellten Interaktionsmustern ergeben sich die folgenden Gleichungen für Pull- und Push-Kosten, wie sie den Heuristiken als Berechnungsgrundlage dienen, um die kostengünstigere der beiden Kommunikationsarten auf Basis von Interaktionen und dem angewendeten Kostenmodell zu bestimmen. Die Gleichung 4.15 zeigt die Gegenüberstellung von Unicast Pull und Unicast Push, wie sie von der link-basierten Heuristik h_{lnk} angewendet wird. Die Gleichung 4.16 zeigt den Vergleich von Multicast Pull und Multicast Push Kosten, welche von der den geräte- und hybriden Heuristiken h^*_{dev} und h^*_{hyb} in der erweiterten Verwendung mit Multicast Pull angewendet werden. Durch Gegenüberstellung der Unicast Pull und Multicast Push Kosten ergeben sich die Bewertungen für h_{dev} und h_{hyb}.

$$C_{UC}^{Pull} = C_{UC}^{Push} \Leftrightarrow \quad r_a \cdot 2\,|S_a| = u_d \cdot |A_d| \tag{4.15}$$

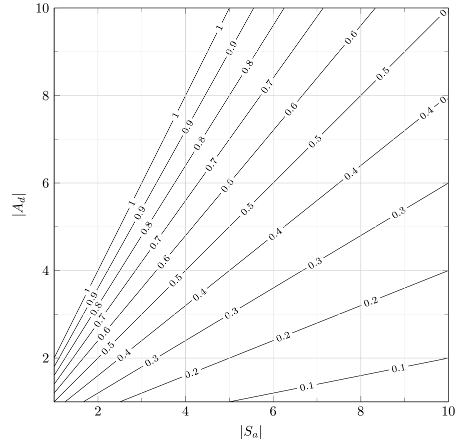
$$C_{MC}^{Pull} = C_{MC}^{Push} \Leftrightarrow \quad r_a \cdot (f(|S_a|) + |S_a|) = u_d \cdot f(|A_d|) \tag{4.16}$$

Um die Kostengleichheit für die Heuristik abzuschätzen, werden die 4.15 und 4.16 mit Werten belegt und umgestellt, um Anfrageraten r_a entsprechend der Konfiguration zu erhalten. Für die Darstellung der Werte in den Abbildungen 4.5 wurden die folgenden Variablen mit den Werten belegt: Die Aktualisierungsraten u_d wurde auf 1.0 gesetzt und als Kostenfunktion für Multicast die Quadratwurzel $f(n) = n^{0.5}$ gewählt, wobei n die Anzahl der zu adressierenden Nachrichtenempfänger ist.

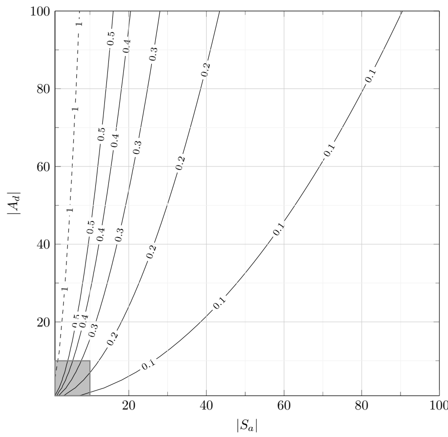
Die Abbildungen 4.5a und 4.5b zeigen die Anwendung der Formel 4.15 für reine Unicast-Abschätzungen sowie die Abbildungen 4.5c und 4.5d der Formel 4.16 für Multicast im Push und Pull. Die Abszissenachse zeigt die aufsteigende Größe der dynamischen Menge $|S_a|$, d.h. Anzahl der gebundenen Geräte. Die Ordinatenachse weist gespiegelt die Mengenbildung aus Gerätesicht aus mit aufsteigender Größe $|A_d|$ die assoziierten Anwendungen. Während die beiden Abbildungen 4.5a und 4.5c einen Wertebereich $[1, 100]$ aufzeigen, stellen die Abbildungen 4.5b und 4.5d einen Zoom der Wertebereichs $[1, 10]$ dar, welcher in den linken Abbildungen durch ein grau gefärbtes Rechteck dargestellt ist. Durch die Gegenüberstellung der beiden Größen $|S_a|$ an der x-Achse und $|A_d|$ an der y-Achse lassen sich die Anfrageraten r_a ablesen, die eine Kostengleichheit bei gesetzter Aktualisierungsrate und Kostenfunktion ergibt. Die Abbildungen zeigen Datenreihen zu Anfrageraten r_a zwischen [0,1, 1] alle 0,1 Schritte auf, Abbildung 4.5c lässt die Werte [0,6, 0,9] aus, um die Übersichtlichkeit zu ermöglichen. Für Unicast Push und Pull ist der Verlauf linear. Für $|S_a| = |A_d|$ lässt sich eine Anfragerate r_a von 0,5 aus Abbildung 4.5a ablesen. Bei beiderseitig gleichgroßen Mengen und der Verwendung von Multicast ergibt sich aus Abbildung 4.5c für $|S_a| = 20$ bspw. eine Anfragerate r_a von unter 0,2.



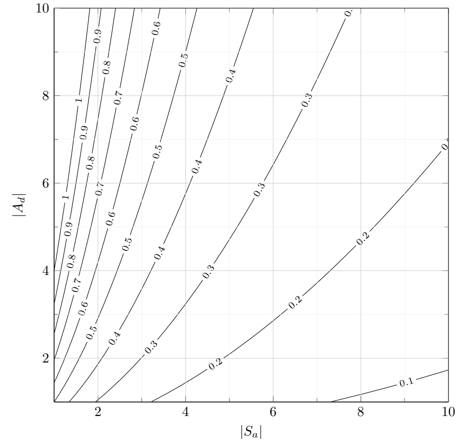
(a) $C_{UC}^{Pull} = C_{UC}^{Push}, |S_a| = |A_d| = [1, 100]$



(b) $C_{UC}^{Pull} = C_{UC}^{Push}, |S_a| = |A_d| = [1, 10]$



(c) $C_{MC}^{Pull} = C_{MC}^{Push}, |S_a| = |A_d| = [1, 100]$



(d) $C_{MC}^{Pull} = C_{MC}^{Push}, |S_a| = |A_d| = [1, 10]$

Abbildung 4.5: λ_{set} für Kostengleichheit Pull/Push, $\lambda_{get} = 1.0, f_{MC} = n^{0.5}$

Vor Durchführung und Präsentation der Simulationsergebnisse lässt sich anhand der Abbildungen 4.5 schlussfolgern, dass im Fall von Unicast Pull und Push das Spektrum an Adaptivität breit über das Verhältnis zwischen Größen dynamischer Mengen S_a und Größe gebundener Anwendungen einem Gerät A_d erteilt ist. Ist S_a größer, bspw. 100, und A_d kleiner, bspw. 20, ist die Kostengleichheit und eine mögliche Umschaltung bei kleiner Anfragerate von 0,1 möglich. Im gespiegelten Fall mit kleinerem S_a und größerem A_d ist eine bedeutend höhere Anfragerate $r_a > 1,0$ notwendig, um Kostengleichheit für Unicast Pull und Unicast Push zu erreichen. Bei der Verwendung von Multicast Pull und Multicast Push sind die Kurven nicht linear bedingt durch die Verwendung der eingesetzten Kostenfunktion. Bei größeren Kardinalitäten und verhältnismäßig größeren dynamischen Mengen S_a gegenüber Geräteanwendungen A_d zeigt einen stetig größer werdenden Bereich, wo bereits verhältnismäßig geringe Anfragerate r_a gegenüber der gesetzten Aktualisierungsrate $r_a = 1,0$ in günstigere Push-Kosten resultiert. In einem Ensemble mit dynamischen Mengen, die an eine kleinere Anzahl von Geräten kommunizieren (S_a), sowie Geräten, die an eine höhere Anzahl an Anwendungen aktualisieren (A_d), entstehen für variierende Anfrageraten r_a die meiste Kostengleichheit und damit die Notwendigkeit von adaptiven Kommunikationsmechanismen zur Umschaltung. Die berechneten Kurven dienen als Ausgangswert, da Anfrage- und Aktualisierungsraten über die Zeit nicht konstant sind sowie Mengenmitglieder zur Laufzeit nicht statisch konfiguriert sind, d.h. unterschiedliche Kardinalitäten aufweisen.

Die Simulationen wurden in Java 1.8 implementiert. Jeder Simulationslauf wird in einem einzelnen Thread in einer separaten Java VM ausgeführt und umfasst insgesamt 150.000 Simulationsschritte, wobei ein Schritt eine Millisekunde in Echtzeit darstellt. Nach einer Aufwärmphase von 50.000 ms beträgt die eigentliche Messzeit 100.000 ms. Alle 1.000 ms werden Auswertungsroutinen ausgeführt, um bei Bedarf Strategiewechsel auszulösen. Darüber hinaus werden alle 10.000 ms statistische Werte mit einem Glättungsfaktor α geglättet, um den Einfluss älterer Interaktionen zu verringern, wobei $0 < \alpha \leq 1$. Beispielsweise bedeutet $\alpha = 0{,}9$, dass der neue geglättete Wert abgeleitet wird, indem dem neuen gemessenen Wert 90 % Gewicht und dem alten geglätteten Wert 10 % Gewicht gegeben werden. Alle Experimente wurden 20-mal ausgeführt. Jeder Simulationslauf wird mit einem einzigartigen, zufällig gewähltem Ausgangswert (engl. Seed) konfiguriert, sodass unterschiedliche Setups mit genau derselben Interaktionssequenz simuliert werden können.

Die Anzahl der initialisierten Anwendungen $|A|$ und Geräte $|D|$ wird für alle Experimente auf jeweils 500 festgelegt. Die Größe der dynamischen Mengen, d.h. die Anzahl der an eine Anwendung a gebundenen Geräte, wird auf $|S_a| = 20$ festgelegt und die an ein Set gebundenen Geräte werden mit gleichmäßiger Wahrscheinlichkeit zufällig ausgewählt.

Die Heuristiken verwenden die geschätzten Kommunikationskosten, um von einem Interaktionsmuster zu einem anderen zu wechseln, wenn dies eine Kostenreduzierung verspricht. Diese Kosten sind jedoch nur eine Schätzung, da sie nur

die Anzahl der Anwendungen und Geräte sowie Getter- und Setter-Raten berücksichtigen. Die tatsächlichen Kommunikationskosten hängen zusätzlich von der Netzwerktopologie und davon ab, wo sich Anwendungen und Geräte im Netzwerk befinden. Anwendungen und Geräte werden mit gleichmäßiger Wahrscheinlichkeit zufällig auf Knoten platziert. Um eine realistischere Sicht auf die Wirksamkeit der Heuristik zu erhalten, wird *BRITE (Boston university Representative Internet Topology gEnerator)* [98] verwendet, um eine internetähnliche Topologie mit 1.000 Knoten zu generieren, die durch 1.225 Links verbunden sind. Basierend auf dieser Topologie werden die tatsächlichen Kommunikationskosten ermittelt.

Für jedes Knotenpaar (V_{src}, V_{dst}) werden die kürzesten Pfade durch eine Breitensuche ermittelt (engl. breadth-first search, BFS). Bei der Unicast-Kommunikation zählt jede Kante entlang eines solchen Pfades bei der Ermittlung der Kommunikationskosten als eine Nachricht. Die Kosten entsprechen also der Länge des Pfades zwischen dem Quell- und dem Zielknoten. Um die Kosten für das Senden einer Nachricht an eine Anzahl von Empfängern mittels Unicast zu ermitteln, werden die Kosten addiert. Für die Multicast-Kommunikation wird ein Multicast-Verteilungsbaum bestimmt, der den Sender und alle Empfänger umfasst, indem alle kürzesten Pfade vom Sender zu den Empfängern verbunden werden. Die Multicast-Kosten entsprechen dann der Anzahl der Links im Baum. Dies bedeutet, dass die Kosten für das Multicast einer Nachricht höchstens so hoch sein können wie die Kosten für das Senden einer Unicast-Nachricht an dieselbe Anzahl von Empfängern. Dies ist der Fall, wenn die kürzesten Pfade zu den Empfängern nicht über eine einzige Verbindung verfügen. Der erstellte Multicast-Baum ist nicht optimal. Um den optimalen Multicast-Baum zu berechnen, müsste ein NP-vollständiges Steinerbaum-Problem gelöst werden.

4.6.1 Gleiche Anfrageraten

Die in diesem Unterabschnitt beschriebenen grundlegenden Experimente weisen allen Anwendungen dieselbe λ_{get}-Rate und allen Geräten dieselbe λ_{set}-Rate zu. Der Bereich für beide Raten beträgt $\lambda \in [0,1\text{s}^{-1}, 10\text{s}^{-1}$. Da die erwartete Zwischenankunftszeit bei Exponentialverteilungen $1/\lambda$ beträgt, wird ein Pull einer bestimmten Anwendung oder ein Push eines bestimmten Geräts im Durchschnitt zwischen 0,1s = 100 ms und 10 s = 10.000 ms erwartet. Der Schwellwert T, um teure Kontrollnachrichten ohne signifikante Nachrichteneinsparung sowie das Oszillieren der Nachrichten zu vermeiden, wird auf 5 % der kleineren Kostenart gesetzt, $min(C_{pull}, C_{push})$.

Die drei Abbildungen 4.6 und 4.7 untersuchen die Kommunikationskosten für unterschiedliche Anforderungsraten λ_{get} (Abszissenachse) und Aktualisierungsraten λ_{set} (Ordinatenachse) in einem 3D-Diagramm. Die Kommunikationskosten werden auf der Applikatenachse angezeigt und umfassen sowohl Push- als auch Pull-Nachrichten, aber auch Kontrollnachrichten, die den durch unsere Heuristik verursachten Overhead darstellen. Die Farbe der Punkte zeigt den relativen

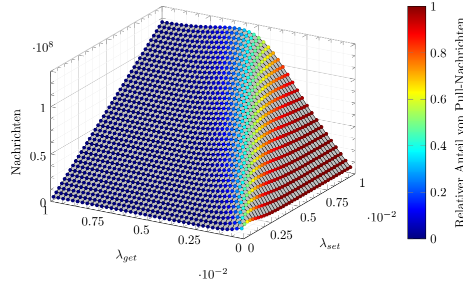


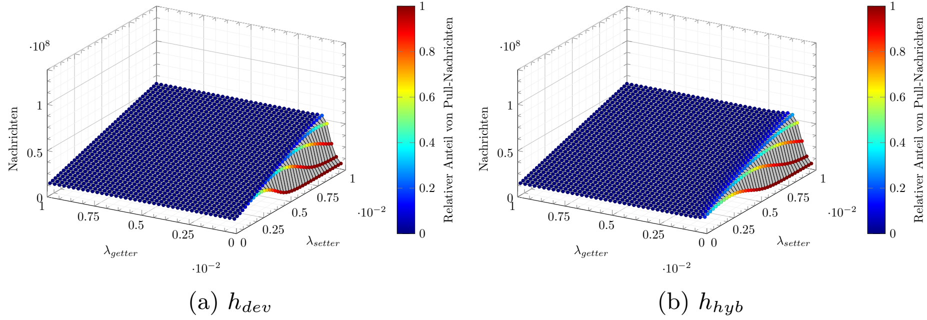
Abbildung 4.6: Gleich Get/Set-Raten h_{lnk}

Anteil der Interaktionen im Pull-Modus an: Rot bedeutet, dass die meisten Verbindungen (a_i, d_j) im Pull-Modus verwendet werden, während blau die meisten den Push-Modus.

Abbildung 4.6 wertet die linkbasierte Heuristik h_{lnk} aus, die zwischen Unicast-Pull und Unicast-Push unabhängig für jeden Link (d.h. jedes Paar aus einer Anwendung a und einem Gerät d) umschaltet. Das Diagramm hat einen abgerundeten Rand, der durch farbige Punkte in den Farben zwischen den Extremen angezeigt wird, wo die Kosten für Unicast-Pull und Unicast-Push ähnlich sind und beide Schemata im System angewendet werden. In den Bereichen mit Mischkommunikation ist eine leichte Ausbeulung der Gesamtnachrichtenanzahl gegenüber den Ebenen mit Kommunikation ausschließlich in einem Kommunikationsparadigma. Dies ist in den entstehenden und in Grenzsituationen häufiger auftretenden Kontrollnachrichten begründet, welche die Gegenseite eine Verbindung (a_i, d_j) zum Umschalten des Kommunikationsparadigmas instruiert. Durch Extrapolieren der roten Ebene und der blauen Ebene können die Kosteneinsparungen im Vergleich zu einem reinen Pull- bzw. einem reinen Push-Ansatz mit ausschließlich Unicast-Kommunikation geschätzt werden.

Abbildung 4.7a zeigt die gerätebasierte Heuristik h_{dev}, die entweder vollständig zwischen Unicast-Pull oder Multicast-Push für alle Links eines Geräts (a_i, d) umschaltet. Die Heuristik reduziert die Anzahl der Nachrichten für die Mehrheit der Punkte erheblich. Der Grund dafür ist, dass eine Multicast-Push-Nachricht im Vergleich zu mehreren Unicast-Push-Nachrichten meistens Kosten spart, da sie nicht mehrere Male über einen Link läuft und die Replikation der Nachrichten erst an späterer Stelle im Nachrichtenverlauf am letzten gemeinsamen Netzwerk-Knoten stattfindet. Die Abbildung zeigt auch, dass Multicast-Push im Vergleich zur linkbasierten Heuristik bereits bei niedrigeren Anfrage- und höheren Aktualisierungsraten angewendet wird. Dies ist deutlich sichtbar, wenn man den Verlauf des Randes in beiden Abbildungen miteinander vergleicht.

Abbildung 4.7b evaluiert die hybride Heuristik h_{hyb}, die teilweises Unicast-Pull und teilweises Multicast-Push gleichzeitig verwendet, d.h. nur eine Teilmenge der mit einem Gerät kommunizierenden Anwendungen empfängt eine Multicast-Push-Nachricht (vgl. Teilmenge U_d in Abschnitt 4.4.3), während die



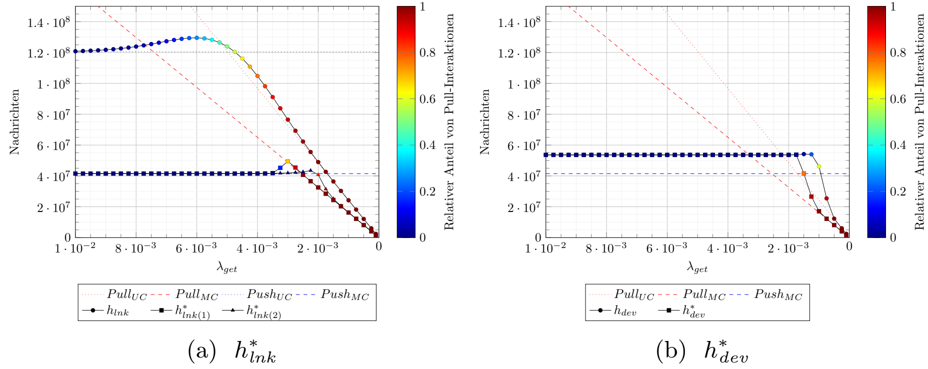
(a) h_{dev} (b) h_{hyb}

Abbildung 4.7: Statische Get/Set-Raten, $C_{UC}^{Pull}, C_{MC}^{Push}$

übrigen Anwendungen stattdessen Unicast-Pull (vgl. R_d) anwenden. Für die meisten Bereiche des Diagramms sind die Kommunikationskosten ähnlich denen von Abbildung 4.7a. Es gibt jedoch auch einen Bereich mit geringfügig höheren Kosten, der durch einen Buckel gekennzeichnet ist. Dies ist auf stochastische Effekte zurückzuführen, die bei der hybriden Heuristik stärker zum Tragen kommen, da die hybride Heuristik für einen einzelnen mit dem Gerät verbundenen Link (a_i, d) entscheidet, ob dieser sich im Push- oder Pull-Modus befindet, während die gerätebasierte Heuristik h_{dev} alle Links nur entweder im Push- oder im Pull-Modus verwenden kann.

Auf dem Vorherigen aufbauend werden nun die Ergebnisse für die Varianten der Heuristiken vorgestellt, die ausschließlich Multicast-Kommunikation verwenden (vgl. Abschnitt 4.4.4). Zur genaueren Darstellung werden die veränderten Heuristiken nicht in 3D-Diagrammen mit den Wertebereichen in beiden Raten dargestellt, sondern nur die Scheibe mit $\lambda_{set} = 0.1$ (der größte Wert an der y-Achse).

Abbildung 4.8a zeigt erwartungsgemäß, dass die größte Leistungsänderung bei der linkbasierten Heuristik auftritt, die zuvor Unicast-Kommunikation sowohl für Push als auch für Pull verwendete und nun Multicast-Kommunikation für beide Schemata verwendet. Es ist offensichtlich, dass die Anwendung von Multicast-Kommunikation die Kommunikationskosten sowohl für den Bereich, in dem Push angewendet wird, als auch für den Bereich, in dem Pull angewendet wird, erheblich reduziert. Es ist auch ersichtlich, dass Pull in einem viel kleineren Bereich angewendet wird als zuvor. Dies liegt daran, dass Push mehr von Multicast-Kommunikation profitiert als Pull, da im letzteren Fall die Antwort in jedem Fall mit Unicast zurückgegeben werden muss. $h^*_{lnk(1)}$ zeigt die Gesamtnachrichtenanzahl der Simulation mit initial konfiguriertem Kommunikationsparadigma Pull; $h^*_{lnk(2)}$ mit Push. Durch die linkweise Kostenvergleiche werden zwar nur anteilige Multicast-Kosten eingerechnet, jedoch besteht keine mengen- bzw. geräteweite Kostenbetrachtung, so dass h^*_{lnk} verzögert um den Punkt der Kostengleichheit umschaltet, je nach bestehendem Kommunikationsparadigma. Bei der anderen Heuristik h_{dev} in Abbildung 4.8b sind die Einsparungen weniger ausgeprägt, da



(a) h^*_{lnk} (b) h^*_{dev}

Abbildung 4.8: Heuristiken mit C^{Pull}_{MC}, $\lambda_{set} = 0.1$

sie bereits Multicast-Push angewendet haben und daher nur Unicast-Pull durch Multicast-Pull ersetzt wurde. In der Simulationskonfiguration mit periodischem Austausch der Statistiken im Push-Paradigma von dynamischer Menge zum Gerät entsteht ein konstanter Overhead zur Nachrichtenanzahl für Datenaktualisierungen wie im reinen Multicast Push $Push_{MC}$. In den Abbildungen sind vor allem zwei Dinge zu erkennen: Erstens wird Pull in einem etwas größeren Bereich angewendet als zuvor und zweitens sind die Kosten in einem großen Teil des Bereichs, in dem Pull angewendet wird, niedriger als zuvor. Es gibt jedoch auch einen kleinen Bereich, in dem die Kommunikationskosten höher geworden sind.

In der Abbildung 4.7 erscheinen die Ergebnisse der beiden Varianten der Hybridheuristiken h_{hyb} und h^*_{hyb} auf den ersten Blick ähnlich zur jeweiligen gerätebasierten Heuristik h_{dev} und h^*_{dev} und weniger vorteilhaft für die hybride Variante. Es wurden für alle Anwendungen dieselbe Getter-Rate und für alle Geräte dieselbe Setter-Rate verwendet, was der Worst-Case für die hybride Heuristik ist. Damit die hybride Heuristik tatsächlich besser abschneidet als die gerätebasierte, ist es notwendig, dass die Getter-Raten der Anwendungen und/oder die Setter-Raten der Geräte variieren. Nur in diesem Fall wird sich die höhere Flexibilität voraussichtlich auszahlen. Daher werden folgend die gespreizten Getter-Raten betrachtet. Für gespreizte Setter-Raten können ähnliche Ergebnisse erzielt werden.

4.6.2 Gespreizte Anfrageraten

Bisher hatten alle Anwendungen die gleichen Getter-Raten zugewiesen. Dies erschwerte die Auswahl der richtigen Strategie, insbesondere für die Hybridheuristik. Daher werden folgend Auswirkungen der gestreuten Getter-Raten untersucht. Um die Streuung zu modellieren, wird eine minimale Getter-Rate λ^{min}_{get}, eine maximale Getter-Rate λ^{max}_{get} und einen Streukoeffizienten i definiert, wobei

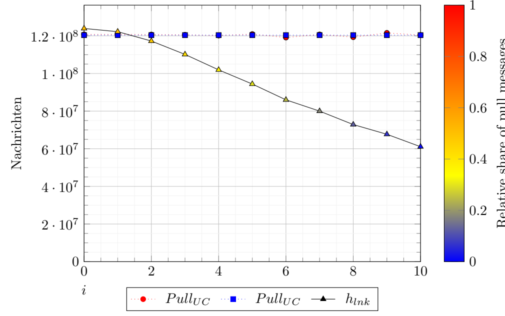


Abbildung 4.9: Gespreizte Get-Raten, h_{lnk}, $E(\lambda_{get}) = 0.005$

$0 \leq i \leq 10$. Die Grundidee ist, dass für $i = 0$ alle Anwendungen die gleiche Getter-Rate erhalten. Für $i \geq 1$ wird jeder Anwendung eine von zwei Getter-Raten zugewiesen, entweder eine niedrigere Getter-Rate λ^l_{get} oder eine höhere Getter-Rate λ^u_{get}. Wir verwenden $\lambda^l_{get} = E(\lambda_{get}) - i \cdot \Delta$ und $\lambda^u_{get} = E(\lambda_{get}) + i \cdot \Delta$, sodass der Unterschied zwischen den beiden Getter-Raten mit zunehmendem i wächst und dass für $i = 10$ beide Raten jeweils λ^{min}_{get} und λ^{max}_{get} entsprechen.

Wenn $E(\lambda_{get})$ in der Mitte des Intervalls $[\lambda^{min}_{get}, \lambda^{max}_{get})]$ liegt, wird $\Delta = (\lambda^{max}_{get} - \lambda^{min}_{get})/20$ verwendet und die beiden Getter-Raten werden mit gleicher Wahrscheinlichkeit von $0,5$ gewählt. Dadurch wird sichergestellt, dass die für alle Anwendungen berechnete, erwartete Getter-Rate $E(\lambda_{get})$ beim gleichen Wert bleibt, wenn i von 0 bis 10 variiert wird. Wenn $E(\lambda_{get})$ jedoch nicht in der Mitte des Intervalls $[\lambda^{min}_{get}, \lambda^{max}_{get})]$ liegt, werden zwei unterschiedliche Delta-Werte verwendet. Dann ist $\lambda^l_{get} = E(\lambda_{get}) - i \cdot \Delta_l$ und $\lambda^u_{get} = E(\lambda_{get}) + i \cdot \Delta_u$, wobei $\Delta_l = (E(\lambda_{get}) - \lambda^{min}_{get})/10$ und $\Delta_u = (\lambda^{max}_{get} - E(\lambda_{get}))/10$. Außerdem müssen beide Getter-Ratenwerte mit unterschiedlichen Wahrscheinlichkeiten abgetastet werden, um sicherzustellen, dass $E(\lambda_{get})$ beim gleichen Wert bleibt. Die Wahrscheinlichkeit p_l, die niedrigere Getter-Rate λ^l_{get} zu wählen, ist $p_l = \frac{\Delta_u}{\Delta_l + \Delta_u}$, während die Wahrscheinlichkeit p_u, die höhere Getter-Rate λ^u_{get} zu wählen, ist $p_u = \frac{\Delta_l}{\Delta_l + \Delta_u} = 1 - p_l$.

Link-basierte Heuristik

Abbildung 4.9 zeigt die Ergebnisse der gespreizten Getter-Raten für die link-basierte Heuristik h_{lnk}. Die Aktualisierungsraten von den Geräten wurde auf $\lambda_{set} = 1,0$ konfiguriert. Der Erwartungswert für die Anfrageraten $E(\lambda_{get})$ wurde auf 0,5 gesetzt, was nach Formel 4.15 die Kostengleichheit ergibt. Das Intervall zum Aufspreizen der Get-Raten wurde auf den bisher betrachteten Wertebereich $[0.001, 0.1]$ aufgespannt.

Es ist ersichtlich, dass die linkbasierte Heuristik h_{lnk} im Vergleich zu reinem Unicast Push ($Push_{UC}$) und reinem Unicast Pull ($Pull_{UC}$) umso mehr Kom-

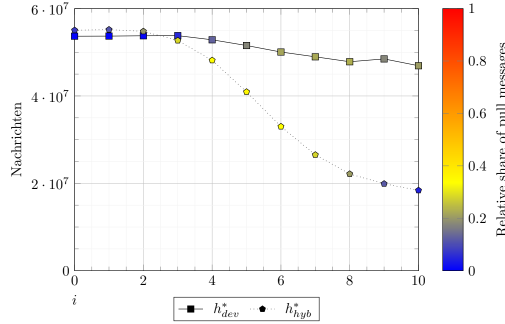


Abbildung 4.10: Gespreizte Get-Raten, h^*_{dev} & h^*_{hyb}, $E(\lambda_{get}) = 0.00182$

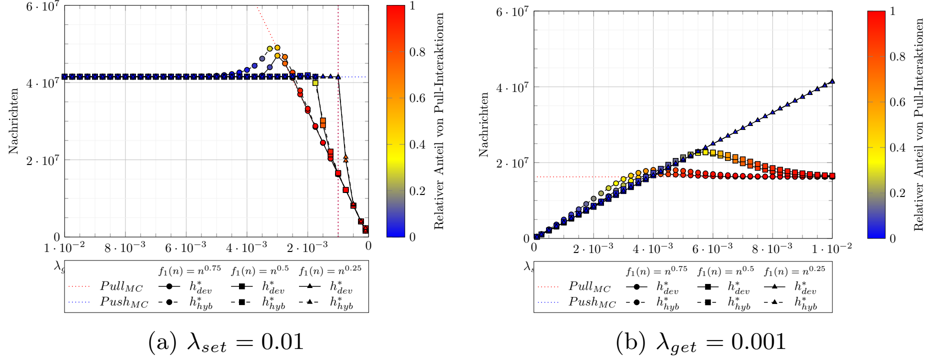
munikationskosten spart, je höher die Spanne zwischen den beiden Getter-Raten wird. Im Gegensatz dazu und wie erwartet ändern sich die Kosten von reinem Unicast Push und reinem Unicast Pull nicht, wenn der Spreizungskoeffizient i variiert, da sich λ_{set} und $E(\lambda_{get})$ nicht ändern. Schließlich sind die Kosten von h_{lnk} für kleinere Werte von i und $\lambda_{set,2}$ oder $\lambda_{set,3}$ etwas höher als die Kosten von reinem Unicast Push. Dies liegt daran, dass einige Links aufgrund stochastischer Effekte auf Unicast Pull umgestellt wurden, obwohl Unicast Push geringere Kosten verursachen würde.

Gerätebasierte und hybride Heuristik

Abbildung 4.10 zeigt die Ergebnisse der Auswertung der gespreizten Getter-Raten für die gerätebasierte Heuristik h^*_{dev} und die hybride Heuristik h^*_{hyb} mit Multicast-Pull ($Pull_{MC}$). Die Leistung der Hybridheuristik umso besser (im Vergleich zur gerätebasierten Heuristik), je höher der Spreizungskoeffizient i wird. Dies zeigt, dass die Hybridheuristiken h_{hyb} und h^*_{hyb} in der Lage sind, die Spreizung der Getter-Raten auszunutzen und die Partition auszuwählen, wo die Geräteseite für die verbunden Anwendungen A_d die disjunkten Teilmengen R_d und U_d bildet und jeweils die Kommunikation für die Verbindung anwendet, die der Partition mit den geringsten Gesamtkosten ausweist. Bei zunehmendem i verringern sich die Kosten nur geringfügig. Dieser Effekt ist jedoch auf stochastische Schwankungen zurückzuführen.

4.6.3 Kostenfunktionen

Die vorgestellten Heuristiken basieren auf einer Kostenfunktion zur Schätzung der Kommunikationskosten von Multicast- und Unicast-Kommunikation. Die Schätzung basiert ausschließlich auf der Anzahl der Empfänger und beinhaltet nur grobe Kenntnisse über die Netzwerktopologie und den Standort von Anwendungen und Geräten. Daher wird vermutet, dass die Schätzung je nach der tatsächlichen Topologie und der angewandten Kostenfunktion besser oder



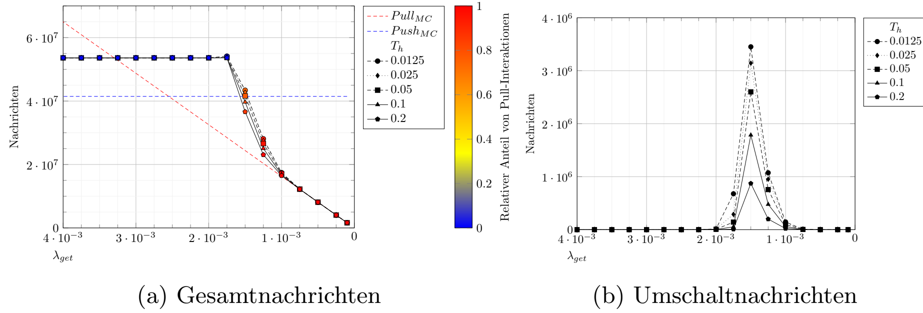
(a) $\lambda_{set} = 0.01$ (b) $\lambda_{get} = 0.001$

Abbildung 4.11: Multicast-Kostenfunktionen

schlechter ausfallen kann. Es werden daher die Auswirkungen unterschiedlicher Multicast-Kostenfunktionen auf die Wirksamkeit der gerätebasierten und der hybriden Heuristiken untersucht, die Multicast-Push anwenden. Als Kostenfunktionen wurde entscheiden $f_1(n) = n^{0,75}$, $f_2(n) = n^{0,5}$ und $f_3(n) = n^{0,25}$ zu verwenden, wobei n die Anzahl der Empfänger ist.

Um die Auswirkungen verschiedener Multicast-Kostenfunktionen zu untersuchen, werden Ergebnisse für zwei Scheiben präsentiert (eine, wo die Getter-Rate variiert, und eine, wo die Setter-Rate variiert). Beide Scheiben wurden so gewählt, dass sie eine detaillierte Untersuchung des Bereichs ermöglichen, in dem der Wechsel von Push zu Pull und umgekehrt erfolgen sollte. Die Ergebnisse sind in den Abbildungen 4.11 dargestellt, die als nächstes besprochen werden. Zur besseren Vergleichbarkeit wurden die x-Achsen in den Diagrammen, die unterschiedliche Getter-Raten untersuchen (Abbildung 4.11a) so gespiegelt, dass die 0 auf der rechten Seite der x-Achse liegt.

Die Abbildungen 4.11 zeigen die Datenaustauschkommunikation für die Heuristiken h^*_{dev} und h^*_{hyb}, die ausschließlich Multicast-Kommunikation für die drei betrachteten Multicast-Kostenfunktionen anwenden. Zum Vergleich der Kostenfunktionen werden hier nicht die Gesamtnachrichtenanzahl mit Datenaustausch- und Kontrollnachrichten der Heuristiken dargestellt, sondern ausschließlich erste Transportnachrichten für den Datenaustausch, um die Wirkung der Wahl der Kostenfunktion in Bezug auf reine Push- oder Pull-Kommunikation zu beschreiben, welche keine Kontrollnachrichten erzeugt. In Abbildung 4.11a wird die Getter-Rate variiert und eine feste Setter-Rate von $\lambda_{set} = 0,01$ verwendet. Was in der Abbildung zu sehen ist, ist, dass die Kostenfunktion $f_1 = n^{0,75}$ viel zu spät von Pull auf Push umschaltet, wenn die Getter-Rate steigt, was zu hohen Kommunikationskosten in dem Bereich führt, in dem λ_{get} zwischen $2,5 \cdot 10^{-3}$ und $5 \cdot 10{-}3$ liegt. Allerdings führt f_1 auch zu den niedrigsten Kosten, wenn die Getter-Rate unter $2,5 \cdot 10^{-3}$ liegt. Die anderen beiden Kostenfunktionen $f_2 = n^{0,5}$ und $f_3 = n^{0,25}$ wechseln früh genug zum Push, verursachen aber auch höhere Kosten in dem Bereich, in dem die Pull-Kommunikation dominiert. Der



(a) Gesamtnachrichten (b) Umschaltnachrichten

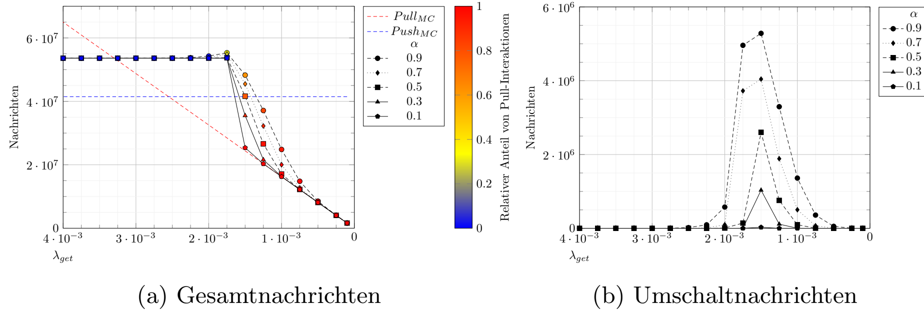
Abbildung 4.12: Schwellwerte Paradigmenwechsel

beste Kompromiss scheint f_2 zu sein. In Abbildung 4.11b wird die Setter-Rate variiert und eine feste Getter-Rate von $\lambda_{set} = 10^{-3}$ verwendet. In der Abbildung ist zu sehen, dass $f_1 = n^{0,25}$ nicht von Push auf Pull umschaltet, wenn die Setter-Rate λ_{set} erhöht wird. $f_3 = n^{0.75}$ wechselt früh genug von Push auf Pull, verursacht aber höhere Kosten in dem Bereich, in dem die Pull-Kommunikation dominiert. Auch hier scheint $f_2 = n^{0.5}$ der beste Kompromiss zu sein, sodass f_2 unter Berücksichtigung beider Werte insgesamt die beste Wahl ist.

4.6.4 Konfigurationen

Die vorgestellten Heuristiken werden mit unterschiedlichen Parametern konfiguriert, welche sich direkt auf das Umschaltverhalten haben und damit die Adaptivität mitbestimmen. Die Varianz einzelner Parameter wird im Folgenden an der gerätebasierten Heuristik h^*_{dev} mit Multicast-Pull vorgestellt. Dieses weist bei statischen Raten über den Simulationszeitraum vergleichbare Ergebnisse wie die hybrid-basierte Heuristik auf. Die Darstellungen bilden auch hier die hintere „Scheibe“ aus den 3D-Diagrammen bei der Erstevaluation der Heuristiken ab mit konstanten Aktualisierungsraten $\lambda_{set} = 10^{-2}$.

Die Abbildungen 4.12 zeigen das Verhalten von h^*_{dev} bei unterschiedlichem Schwellwert T_h, welche als minimale Differenz zwischen Pull- und Pushkosten gesetzt wird, um einen Paradigmenwechsel auszulösen. Die mittlere Datenreihe mit den Quadrat-Datenpunkten zeigt mit $T_h = 0,05$ die Konfiguration, welche in den vorherigen Simulationen gesetzt war: 5% von der kleineren der zu vergleichenden Kostenarten. Die Abszissenachsen stellen für die Lesbarkeit nur den Ausschnitt des Wertebereichs $\lambda_{set} = [0,25 \cdot 10^{-3}, 4 \cdot 10^{-3}]$ dar. Abbildung 4.12a zeigt die Gesamtnachrichtenanzahl inkl. Steuerungsnachrichten der Heuristik. Dies zeigt sich im konstanten Abstand der Heuristik zur Datenreihe für reines Push für $\lambda_{get} > 1,5 \cdot 10^{-3}$, wo regelmäßig Statistiken von Anwendungen an die Geräte gesendet werden, die im nur Push kommunizieren. Die Datenreihen mit verändertem T_h liegen sehr dicht beieinander und weisen kaum unterschiedliches Umschaltverhalten auf. Die Varianz entsteht durch die stochas-

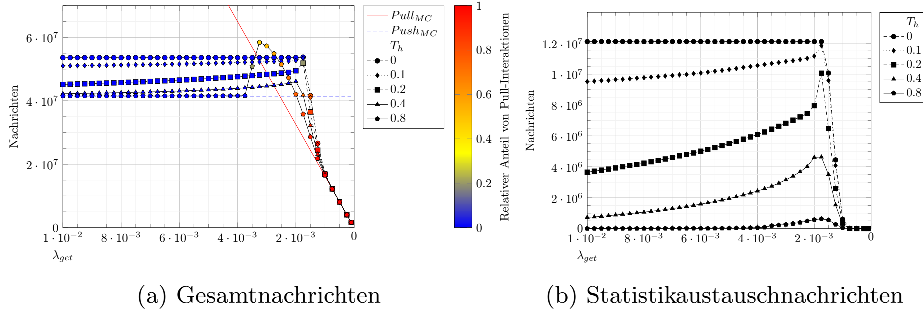


(a) Gesamtnachrichten (b) Umschaltnachrichten

Abbildung 4.13: Glättungsfaktor

tischen Effekte bei der Ereignisgenerierung. Abbildung 4.12b stellt nur Anzahl der Umschaltnachrichten dar, welche von Geräten an Anwendungen gesendet werden, um einen Paradigmenwechsel einzuleiten, insofern eine Kostenersparnis unter Berücksichtigung des Schwellwertes T_h entsteht. Die Ordinatenachse der Darstellung stellt einen angepasste Wertebereich dar. Die höchste Anzahl von Umschaltungen – hier durch das Aufaddieren der relevanten Nachrichten – besteht bei $\lambda_{set} = 1,5 \cdot 10^{-3}$, was einer der angrenzenden Messpunkte an der errechneten Kostengleichheit bei $\approx 1,8 \cdot 10^{-3}$ ist. Erwartungsgemäß resultiert ein geringer konfigurierter Schwellwert T_h in höherer Umschaltvorgänge in näheren Bereichen der Kostengleichheit von C_{MC}^{Pull} und C_{MC}^{Push}, was die Heuristik sensitiver macht.

Der Glättungsfaktor α bestimmt für eine Heuristik, welche Gewichtung Messwerte aus der zuletzt abgeschlossenen Evaluierung im Verhältnis zur aktuellen bekommen. Mit den gleichen Grundeinstellungen zu den vorherigen Betrachtungen des Schwellwertes T wird der Glättungsfaktor α untersucht und die Ergebnisse in den Abbildungen 4.13 dargestellt. Die Konfiguration für die vorherigen Simulationen stellt auch hier die mittlere Datenreihe mit $\alpha = 0,5$ dar. Die Gesamtnachrichtenanzahl in Abbildung 4.13a zeigt eine stärke Auswirkung unterschiedlicher Konfigurationen. Mit einem kleinen $\alpha = 0,1$ wird Messwerten aus der vergangenen Glättung sehr hohes Gewicht gegeben. Mit statischen Ereignisdistributionen durch Exponentialverteilung über die gesamte Simulationszeit werden in Bereichen mit Nähe zur Kostengleichheit Push/Pull die stochastischen Schwankungen aus dem aktuellen Evaluierungszeitraum durch stabilere Werteübernahmen ausgeglichen. Die Messpunkte der gerätebasierten Heuristik folgen mit aufsteigenden Anfrageraten längst möglich der reinen Pull-Gerade bis $\lambda_{get} = 1,5 \cdot 10^{-3}$ bevor eine sofortige Umschaltung zu Push am nächsten Messpunkt abzulesen ist. Bei einem großen $\alpha = 0,9$ ist der gegenteilige Effekt feststellbar: Bereits bei kleineren Anfrageraten werden Mischformen von Push und Pull durch häufigere Umschaltungen erzeugt, was sich in Abbildung 4.13a an der höchsten Anzahl für Umschaltnachrichten bestätigt. Um den Messpunkt $\lambda_{get} = 1,75 \cdot 10^{-3}$ übersteigt die Gesamtnachrichtenanzahl der Heuristik sogar die jene für reine Push-Kommunikation geringfügig. Der Glättungsfaktor α

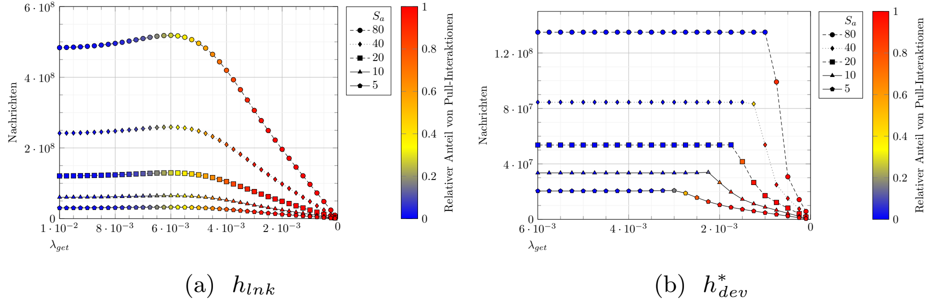


(a) Gesamtnachrichten (b) Statistikaustauschnachrichten

Abbildung 4.14: Schwellwert Statistikaustausch

trägt auch zur Sensitivität bzw. Trägheit von Heuristiken bei. So kann im Umfeld der Kostengleichheit bei Ereignisraten mit wenig Schwankungsbreite und einem großen α ein ungewünschter Overhead entstehen, jedoch gegenteilig bei hohen Varianzen mit gleichem α eine höhere Reaktivität der Heuristik erzeugt werden.

In den Geräte- und hybriden Heuristiken entscheidet die Geräteseite für den Paradigmenwechsel von Pull nach Push und auch umgekehrt für alle gebundenen Anwendungen. Kommunizieren die Verbindungen im Push, so werden wiederholt aktualisierte Pull-Statistiken von Anwendungen an ein Gerät übertragen, damit dieses Kostenvergleiche auf Basis aktuellerer Werte durchführen kann. In diesem Zustand werden in der initialen Konfiguration die Statistiken nach einem Evaluierungsschritt an einer Anwendung zum Gerät gesendet unabhängig von Abweichungen zur vorherigen Auswertung. Dafür wird der Schwellwert zum Senden einer Statistik bestimmt, der auf $T_s = 0,0$ konfiguriert ist. Die Abbildungen 4.14 untersuchen das Verhalten der gerätebasierten Heuristik mit Multicast Pull h^*_{dev} bei Anheben dieses Konfigurationsparameters. Abbildung 4.14a zeigt die Gesamtnachrichtenanzahl für verschiedenen T_s auf dem gesamten Wertebereich $\lambda_{get} = [0, 10^{-2}]$. Ein Anheben dieses Wertes, der prozentual interpretiert auf die Pull-Kosten des letzten Statistikaustausches angewendet wird, zeigt zum einen, dass der Overhead im Bereich der Kostengleichheit geringer wird, jedoch bei ansteigenden Anfrageraten auch später umgeschaltet wird. Im Fall $T_s = 0,8$ ist der Schwellwert 4/5 der zuletzt kommunizierten Statistik was in ein sehr spätes Umschalten nach Push mündet und einen Overhead erzeugt, wo bei den Anfrageraten bereits eine Kommunikation im Push kostengünstiger wäre. Des weiteren ist Abbildung 4.14b zu entnehmen, dass ein höhere Sendeschwellwert T_s bei Anfrage- und Aktualisierungsraten entfernt von dem Bereich der Kostengleichheit die Anzahl der Nachrichten für Statistikaustausch schneller absinken lässt – bei über die Simulationsdauer konstant beabsichtigten Ereignisterminierungen mit Poisson-Prozess Die Datenreihe mit $T_s = 0.2$ hat im Bereich der Kostengleichheit beim Messpunkt $\lambda_{get} = 1,75 \cdot 10^{-3}$ die Spitze an Nachrichten zum Statistikaustausch, die fast die Anzahl erreicht, als wäre kein Schwellwert konfiguriert ($T_s = 0,0$), sackt dann jedoch signifikant auf 1/3 der Nachrichten



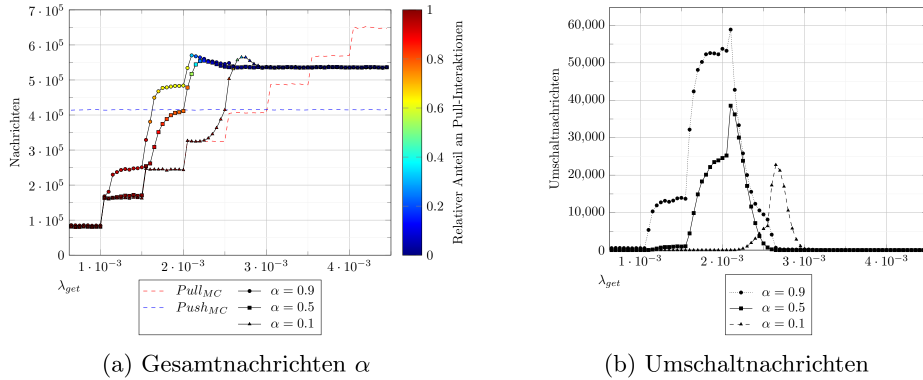
(a) h_{lnk} (b) h^*_{dev}

Abbildung 4.15: Umschaltverschiebung bei unterschiedlichen Mengengrößen

bei der maximal dargestellten Anfragerate mit $\lambda_{get} = \cdot 10^{-2}$ ab. Die Datenreihe mit $T_s = 0,8$ erzeugt kaum Umschaltnachrichten im Bereich der Kostengleichheit und sinkt beim Verlassen dessen am schnellsten auf null ab. Die Wahl des Schwellwertes $T_s > 0$ kann zu einer Reduzierung der Nachrichtenanzahl dieses Kontrolltyps einer Heuristik beitragen. Ein zu hoch gewählter Parameter kann jedoch im Umfeld der Kostengleichheit zu suboptimalen Umschaltungen und einer höhere Nachrichtenanzahl führen kann.

4.6.5 Mengengrößen

Zu Beginn dieses Abschnitts wurde die Kostengleichheit für unterschiedliche Größen dynamischer Mengen bzw. an ein Gerät gebundene Anwendungen mathematisch motiviert (Abbildungen 4.5). Die Abbildungen 4.15 zeigen die Simulationsergebnisse für unterschiedliche Kardinalitäten dynamischer Mengen $|S_a| = \{5, 10, 20, 40, 80\}$, die in dem Simulations-Setup mit der gleichen Größe von zu einem Gerät assoziierten Anwendungen A_d einhergeht. Abbildung 4.15a zeigt die verbindungsorientierte Heuristik h_{lnk} mit Unicast Pull und Unicast Push. Entsprechend des linearen Kostenverhältnis von Pull zu Push schaltet die Heuristik erwartungsgemäß im Bereich $\lambda_{get} = 5 \cdot 10^{-3}$ um bei einer konstant konfigurierten Aktualisierungsrate $\lambda_{set} = 10^{-2}$; erkennbar durch die orangegefärbten Datenpunkte. Bei größeren Mengen ist der Overhead im Umschaltbereich deutlich ausgeprägter, was durch die höhere Anzahl an Verbindungen zwischen Anwendungen und Geräten bei gleichbleibend platzierter Klientenanzahl sowie den stochastischen Effekten der Ereignisdistribution zurückzuführen ist. Abbildung 4.15b stellt die Anwendung der geräte-basierten Heuristik h^*_{dev} mit Multicast Pull dar. Je größer die Menge ist, desto geringer ist das Verhältnis von Anfrage- zu Aktualisierungsraten, und umgekehrt. Während bei einer größeren Menge $|S_a| = 80$ die Umschaltung zwischen Pull und Push bereits bei $\lambda_{get} = 10^{-3}$ stattfindet, schaltet die Heuristik bei einer kleineren Menge mit $|S_a| = 5$ erst bei $\lambda_{get} = 2,75 \cdot 10^{-3}$ um, was sehr nah den Werten aus Abbildungen 4.5c und 4.5c für die Kardinalitäten ist. Zu berücksichtigen sind in der Simulation stochastische Effekte der Ereignisausspielung sowie die Darstellung



(a) Gesamtnachrichten α (b) Umschaltnachrichten

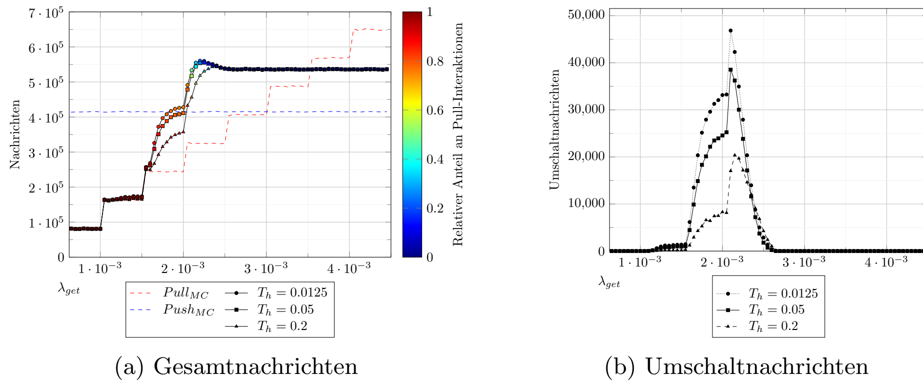
Abbildung 4.16: Evaluierung des Schwellwertes α im zeitlichen Verlauf

der Gesamtnachrichtenanzahl inklusive Kontrollverkehr, der im mathematischen Modell nicht mit einfließt.

4.6.6 Zeitliche Änderungen der Ereignisraten

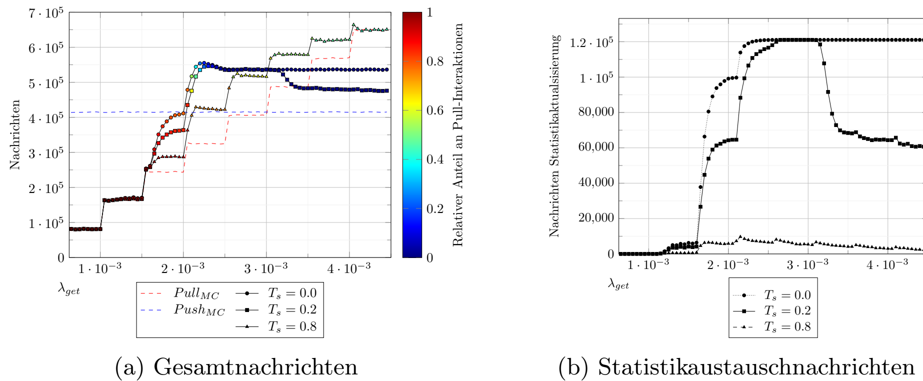
Bisher wurden die Heuristiken mit konstanten Ereignisraten und Konfigurationen über die Messzeit untersucht. Folgend soll die Veränderung der Anfragerate λ_{get} die Wirkungsweise im zeitlichen Verlauf aufzeigen. Die Konfiguration der Heuristiken mit Mengengrößen und konstanter Aktualisierungsrate λ_{set} wurde aus den vorherigen Simulationen übernommen. Allen Anwendungen sowie Geräten wurden jeweils die gleichen Anfrage- bzw. Ereignisraten zugewiesen. Die gerätebasierte Heuristik h^*_{dev} mit Multicast Pull und Multicast Push findet Anwendung. Die Abszissenachse in den Abbildungen 4.16–4.18 zeigen den zeitlichen Verlauf der Experimente. Nach der Aufwärmphase wurde alle 10.000 Simulationsschritte die Anfragerate λ_{get} um $5 \cdot 10^{-4}$ erhöht, um einen langsamen Werteanstieg über den Punkt der Kostengleichheit von Pull und Push aufzuzeigen. Zur besseren Darstellung wurde die Beschriftung der x-Achse vom zeitlichen Fortschritt auf die aktuelle Anfragerate λ_{get} überschrieben. Die Messkurven der Heuristiken stellen die gesamten Nachrichten dar, d.h. inklusive Kontrollnachrichten, was die höhere Anzahl in kompletter Umschaltung auf ein Kommunikationsparadigma gegenüber den Vergleichsmessungen in reinem Push und Pull ergibt.

Die beiden Abbildungen 4.16 zeigen die Auswirkungen eines unterschiedlich konfigurierten Glättungsfaktors α. Die linke Abbildung 4.16a zeigt die Gesamtnachrichten im System, während die rechte Abbildung 4.16b ausschließlich die Kontrollnachrichten der Heuristiken darstellt. Es ist zu erkennen, dass ein hoher Glättungsfaktor ($\alpha = 0.9$) Werten jüngerer Evaluierungszeiträume mehr Gewicht gibt und Umschaltungen früher stattfinden. Die hohe Reaktivität resul-



(a) Gesamtnachrichten (b) Umschaltnachrichten

Abbildung 4.17: Evaluierung des Schwellwertes T_h im zeitlichen Verlauf



(a) Gesamtnachrichten (b) Statistikaustauschnachrichten

Abbildung 4.18: Evaluierung des Schwellwertes T_s im zeitlichen Verlauf

tiert in eine höhere Anzahl von Kontrollnachrichten in Wertebereichen um den Punkt der Kostengleichheit.

Die beiden Abbildungen 4.17 zeigen in gleicher Weise die unterschiedliche Konfiguration des Schwellwertes T_h, der steuert bei welchem Abstand zwischen den beiden errechneten Kosten für Pull und Push während einer Evaluierung auf die günstigere umzuschalten ist. Die linke Abbildung 4.17a lässt im zeitlichen Unterschied kaum Auswirkungen mit unterschiedlichen Konfigurationen auf die Reaktivität der Umschaltung erkennen. Die rechte Abbildung 4.17b zeigt einen höheren Kontrollverkehr im Umschaltbereich bei kleiner konfiguriertem Schwellwert, was die Gesamtnachrichtenanzahl geringfügig ansteigen lässt.

Die beiden Abbildungen 4.18 stellen die Variation des Schwellwertes T_s im zeitlichen Verlauf dar, der Einfluss darauf hat, bei welcher Veränderung der Pull-Statistiken gegenüber der vorherigen Statistikaktualisierung an ein Gerät eine neue Aktualisierung gesendet wird. Die rechte Abbildung 4.18b zeigt große Unterschiede der Konfiguration. Der initial angewendete Parameter $T_s = 0$ zeigt die

konstante periodische Übertragung der Statistiken, wenn die Heuristik im Verlauf nach Push umgeschaltet hat. Auch ein erhöhter Schwellwert $T_s = 0.2$ scheint vorteilhaft, da die Abbildung 4.18a eine zeitlich vergleichbare Umschaltung zeigt und Kontrollnachrichten in der entstehenden reinen Push-Kommunikation reduziert werden. Ein hoch konfigurierter Schwellwert $T_s = 0.8$ erzeugt jedoch ein unerwünschtes Verhalten. Es werden kaum Nachrichten für den Statistikaustausch an das Gerät generiert, das die Trägheit in der Evaluierung aufzeigt und sehr verzögert in Richtung Push umschaltet, was in bedeutend höhere Gesamtnachrichten resultiert.

4.7 Verwandte Arbeiten

Im Bereich Internet der Dinge (IoT) konkurrieren mehrere prominente Protokolle um die Gunst der Entwickler [188], beispielsweise MQTT [9], CoAP [150] und XMPP [141], um nur einige zu nennen. Trotz der Unterschiede und Details erfüllen sie alle den Bedarf der Entwickler an flexibler Interaktion, indem sie vielseitige Funktionen zur Gruppenkommunikation bieten. Entweder sind dies wichtige, dem Protokoll inhärente Funktionen (z.B. Publizieren/Abonnieren in MQTT) oder wurden durch Erweiterungen hinzugefügt. Im Fall von CoAP wird die Gruppenkommunikation und Multicast-Unterstützung durch das Beispiel einer einzigen Nachricht motiviert, mit der alle Lichter in einem bestimmten Raum sofort eingeschaltet werden können [133]. Tatsächlich passt dies perfekt zu unserem Konzept dynamischer Mengen. Allerdings ist keines der Protokolle von Natur aus adaptiv, d.h. in der Lage, zur Laufzeit automatisch das Kommunikationsparadigma zu wechseln, wenn dies von Vorteil ist. Stattdessen bleibt es dem Entwickler überlassen, die Anzahl der Anwendungen und Geräte sowie deren Anforderungs- und Aktualisierungsraten vorherzusehen und zur Entwurfszeit das beste Kommunikationsschema (d.h. Pull oder Push) auszuwählen. In der Literatur wurden mehrere adaptive Kommunikationsschemata vorgestellt, insbesondere für die Verbreitung von Webdaten. Franklin und Zdonik [54] erkunden den Gestaltungsspielraum für die Informationsverbreitung (z.B. Client Pull/-Server Push, Unicast/Multicast, ereignisgesteuerte aperiodische/geplante periodische Kommunikation). Deolasee et al. [34] präsentieren adaptive Push/Pull-Schemata für Clients und zwischengeschaltete Proxys, die zwischen beiden Modi wechseln oder sie gleichzeitig verwenden können. Dies gewährleistet ein gewisses Maß an Datenkonsistenz und verbessert die Anwendungsstabilität bei gleichzeitiger Begrenzung der entstehenden Netzwerkkosten. Ebenso lockern die Autoren von [190] bzw. [71] für Caching- und Cloud-Umgebungen die Datenkonsistenzbeschränkungen, um Datenanforderungen und/oder Aktualisierungsbenachrichtigungen zu verschieben und zu sichern, was zu Kostensenkungen führt. Minson und Theodoropoulos [102] wenden Techniken des maschinellen Lernens an, um Aktualisierungsraten besser vorherzusagen und so das Lieferschema zu optimieren. Die erzielten Ergebnisse sind jedoch nicht wesentlich besser, wenn man sie mit viel einfacheren Anpassungsschemata vergleicht. Obwohl die Lockerung von

Konsistenzbeschränkungen und die Vorhersage von Aktualisierungsraten orthogonal zum in dieser Arbeit vorgestellten Ansatz sind, ist ihre Hinzufügung/Integration nicht unkompliziert, da keines davon für Gruppenkommunikation und Multicast-Übertragung entwickelt wurde. Acharya et al [2] zeigen, wie man Anforderungs- und Aktualisierungsraten ausbalanciert, wenn man einen Sendekanal mit hoher Bandbreite für Broadcast-Daten und einen Rückkanal mit geringer Bandbreite für Pull-Anfragen verwendet. Obwohl sehr interessant, sind die Ergebnisse eng an die asymmetrische Einstellung sowie die Verwendung eines Broadcast-Mediums gebunden.

4.8 Diskussion

Dieses Kapitel zeigt die Notwendigkeit adaptiver Kommunikation auf, um redundanten Datenverkehr zu reduzieren. Dabei gilt es Anfragen auf unveränderte Zustände von Datenproduzenten sowie Statusaktualisierungen an Datenkonsumenten zu vermeiden für die keine Weiterverarbeitung vorgesehen ist. Es werden die grundlegenden Interaktionsmuster motiviert für die anfrage- (Pull) und aktualisierungs-orientierte (Push) Kommunikation in Kombination mit einfacher (Unicast) oder mehrfacher (Multicast) Empfängeradressierung. Ein Kostenmodell aus der Kombination von Interaktionsfrequenzen und Kostenfunktionen bildet die Grundlage für die Anwendung in Heuristiken. Da die Gesamtübersicht eines Netzwerks mit Konfigurationen fehlt und Kostenberechnungen zu komplex sind, vereinfachen drei präsentierte Heuristiken mit lokalem Wissen auf der Basis von historischen Daten die Abschätzungen für die Anwendung von Pull- oder Push-Kommunikation. Die Heuristiken haben unterschiedliche Wirkweisen, von der Bewertung einer einzelnen Verbindung zwischen Datenproduzent (Gerät) und -konsument (Anwendung) über lokale Gesamtsicht eines Produzenten mit all seinen registrierten Konsumenten sowie die Mischform durch Partitionierung in anfragende und aktualisierende Teilmengen zu Konsumenten. Für die realitätsnahe Umsetzung der Heuristiken werden die Prozeduren präsentiert, die Statistiken berücksichtigen, welche die Datengrundlagen für die Heuristiken bilden. Parameter zum Glätten der statischen Werte und Schwellwerte für Umschaltungen zwischen den Kommunikationsparadigmen und Statistikaktualisierungen lassen die Stabilität sowie Reaktivität der Heuristiken konfigurieren. Eine Evaluierung mittels diskreter Ereignissimulation bestätigt das Einsparpotential von adaptiver Kommunikation durch den Einsatz von Heuristiken und zeigt die Wirkungen der Konfigurationsparameter auf.

Die Anwendung der Heuristiken erfordert zusätzlichen Kontrollverkehr zum Austausch von Statistikwerten oder Instruktionen zum Wechsel zwischen Push und Pull. Die Abbildung auf Transportprotokolle ist individuell entsprechend derer Fähigkeiten zu bewerten. Das folgende Kapitel 5 präsentiert die Abbildung der Interaktionen auf das Publish/Subscribe-Kommunikationsmuster für eine lose Kopplung zwischen den Klienten und die Anwendung auf nachrichten-basierten Protokolle.

Kapitel 5

Adaptives Publish/Subscribe

Inhalt

5.1 Motivation **138**
5.2 Anforderungen **139**
5.3 Adressierungsschema **140**
5.4 Interaktionsmuster **142**
5.4.1 Initialisierung 142
5.4.2 Bindung 144
5.4.3 Datenkommunikation 145
5.4.4 Adaptivität 146
5.4.5 Beendigung 147
5.5 Kommunikationsprotokoll **148**
5.5.1 Nachrichtentypen 148
5.5.2 Protokollabbildung 151
5.6 Serialisierung **151**
5.7 Architektur **153**
5.8 Evaluierung **155**
5.9 Verwandte Arbeiten **158**
5.10 Diskussion **159**

5.1 Motivation

Verteilte Systeme und das für diese Arbeit motivierte Teilgebiet Internet der Dinge (IoT) sind geprägt von Heterogenität in unterschiedlichen Bereichen der Kommunikation zwischen den Akteuren. Die vernetzten Sensoren und Aktuatoren können je nach Hersteller in Bezug auf ihre Funktionalität und die verwendeten Protokolle variieren. Zu den Merkmalen gehören auch das Transportprotokoll, das Kommunikationsparadigma und die Serialisierung der Nutzdaten. Zum Beispiel kann eine Anwendung nicht das Netzwerkprotokoll unterstützen, durch welches sich ein neu bereitgestelltes Gerät ansprechen lässt. Datenstrukturen benötigen Konventionen, um zwischen Sender und Empfänger gleich verarbeitet werden zu können. Für die Übertragung über das Netzwerk ist eine Serialisierung notwendig, deren Datenverarbeitungsroutinen allen Teilnehmern bekannt und von diesen umgesetzt werden müssen, um semantisch und syntaktisch die gleichen Daten zu beschreiben und zu verarbeiten. Entwickler können nur Entwurfszeit nicht alle möglichen Protokolle und die dafür notwendige Programmlogik für eine Anwendung implementieren. Daher sind Anpassungen zur Laufzeit entsprechend der Dynamik der Netzwerktopologie und des Teilnehmerensembles notwendig. Verbindungs-, Interaktions- und Fehlerbehandlungsroutinen sollen bestmöglich in eine Middleware ausgelagert und abstrahiert werden, um Entwicklern eine allgemeine Schnittstelle zur Interaktion zur Verfügung zu stellen, deren Konfigurationstiefe wählbar bleibt. Asynchrone, nachrichtenbasierte Kommunikation über Warteschlangen oder das Publish/Subscribe-Muster ermöglichen Entkopplungen zwischen Produzenten und Konsumenten in Raum, Zeit, Zustand und Synchronisation (vgl. Abschnitt 2.2.2). Nachrichtenbasierte Protokolle durchdringen auch die Anwendungsdomänen von IoT und allgegenwärtigen Systemen.

Dieses Kapitel stellt die Abbildung der Interaktionsmuster zu entfernten Methodenaufrufen unter Anwendung der Programmierabstraktion der dynamischen Mengen aus Kapitel 3 und der adaptiven Kommunikation aus Kapitel 4 auf nachrichtenbasierte Protokolle zur Anwendung in Publish/Subscribe-Systemen vor. Die Begrifflichkeiten basieren auf Ausführungen in den Vorkapiteln, welche zur Verbesserung des Leseflusses vereinfacht angewendet werden: Der Methodenaufruf aus einer Anwendung über die Programmierabstraktion der dynamischen Mengen zur Gruppenkommunikation mit mehreren Geräten ist über den Stellvertreter (Proxy) verwaltet. Der Proxy realisiert die Kommunikation über Netzwerk-Klienten. Analog meint die Nennung eines Gerätes in diesem Kapitel die Interaktion dessen software-seitigen Stellvertreters mit Verbindung zur Transport-Komponente. Kommunikationsarten, d.h. Anfragen von Anwendungen und Aktualisierungen von Geräten sowie die adaptive Kommunikation mit Auswertungsprozeduren der Heuristiken und erforderlicher Kontrollnachrichten beziehen sich auf die Vorarbeiten aus Kapitel 4 zur Pull- oder Push-Kommunikation und deren Wechsel. Übertragen in die Publish/Subscribe-Architektur ist eine dynamische Menge in einer Anwendung generalisiert ein Datenkonsument und ein Gerät ein Datenproduzent in Bezug auf Nachrich-

tenfluss zur Kommunikation von Zuständen eines Gerätes, d.h. Aufrufe von Get-Methoden im gekapselten objekt-orientierten Paradigma. Für andere Methodenaufrufe an einem Gerät, die dessen Zustand verändern, d.h. Attribute über Set-Methoden setzen oder Prozeduren aufrufen, ist die Abbildung des Anfrage/Antwort-Paradigmas auf Publish/Subscribe ohne Anwendung einer adaptiven Optimierung des Kommunikationsparadigmas vorgesehen.

Die Grundlage für die Ausführungen in diesem Kapitel bildet die eigene Veröffentlichung [128]. Nach Definition der Anforderungen wird in Abschnitt 5.3 das Adressierungsschema für themen-basiertes Routing in Publish/Subscribe dargestellt. Auf Basis der Interaktionsmuster werden anschließend die Publish/ Subscribe-Interaktionen in Abschnitt 5.4 präsentiert. Die Definition von Nachrichtenobjekten zur Überführung in Formate verschiedener Protokolle schließt in Abschnitt 5.5 an und wird durch Serialisierungsstrategien der Nutzlast in Abschnitt 5.6 ergänzt. Die Architektur der Transportabstraktion wird in Abschnitt 5.7 skizziert. Eine Evaluierung der Funktionsweise der Methodenabbildung sowie Anwendung der Heuristiken aus dem Vorgängerkapitel und Verweise auf verwandte Arbeiten schließen dieses Kapitel ab.

5.2 Anforderungen

Eine angestrebte Plattform- und Protokollunabhängigkeit verlagert die Berücksichtigung schwer vorhersagbarer Netzwerkeigenschaften aus der Anwendung während der Entwurfszeit zur Laufzeit in die Ausführungsumgebung. Die folgenden Anforderungen müssen von der vorgeschlagenen Programmierabstraktion abgedeckt werden.

Geräteauswahl. Zur Geräteauswahl sendet der Stellvertreter einer dynamischen Menge eine (Multicast-)anfrage an eine Gruppe von Geräten, die die gleiche Schnittstelle implementieren. Die Anfrage enthält normalerweise eine oder mehrere Einschränkungen, die auf den Geräten validiert werden. Wenn die Auswahlkriterien mit der Anfrage übereinstimmen, wird eine (Unicast-)antwort vom Gerät an den Stellvertreter der dynamischen Menge zurückgesendet. Diese Geräte werden dann der dynamischen Menge als Mitglieder für nachfolgende Interaktionen hinzugefügt.

Pull-Kommunikation. Sobald eine dynamische Menge mit Mitgliedern erstellt wurde, können entfernte Aufrufe für die Methoden der Schnittstelle durchgeführt werden, indem die entsprechende Methode des Stellvertreters der dynamischen Menge aufgerufen wird. Nach Erhalt einer Anfrage wird die Methode auf allen ausgewählten Geräten aufgerufen. Schließlich werden die Rückgabewerte von den Geräten an den Stellvertreter der dynamischen Menge zurückgesendet, der diese Werte mithilfe einer Aggregationsfunktion, wie beispielsweise der Berechnung eines Durchschnittswerts, zusammenfasst.

Push-Kommunikation. Bei der Push-Kommunikation werden registrierte dynamische Mengen darüber informiert, dass an einem Gerät eine Änderung stattgefunden hat. Dies dient zur Optimierung von Getter-Methoden, deren Aufrufe den Zustand eines Datenproduzenten, hier eines Gerätes, nicht verändern.

Kontrolldaten. Für die Umsetzung adaptiver Kommunikation mit dem Wechsel zwischen Pull- und Push-Kommunikation sind Austausche mit Anweisungen zum Wechsel des Kommunikationsparadigmas an einzelne oder gruppierte Empfänger notwendig. Werden die Umschaltentscheidungen auf Basis von bspw. Heuristiken getroffen sind je nach Ausprägung zusätzlich Austausche von Statistiken erforderlich.

Metadaten. Für die vorgenannten Interaktionsmuster sind Metadaten erforderlich, die sowohl Transportprotokollen mit also auch ohne modifizierbaren Kopfdaten zugeordnet werden sollen. Die restlichen Daten werden als Nutzlast in den Nachrichtentext eingebettet.

Serialisierung. Um unterschiedliche Programmiersprachen und Plattformen zu unterstützen, wird ein einheitlicher Serialisierungsmechanismus zum Senden und Empfangen serialisierter Nutzdaten eingeführt. Beispielsweise beim Senden eines Datumsobjekts in der Programmiersprache Java weiß ein empfangendes Python-Programm nicht, wie ein Datumsobjekt zu interpretieren ist. Eine einheitliche Serialisierung schließt diese Lücke und ermöglicht Interoperabilität.

Adressierungsschema. Ein grundlegendes Adressierungsschema ist verfügbar. Es stellt beispielsweise sicher, dass Geräte, die die gleiche Schnittstelle implementieren, erreicht werden können, ohne dass eine Nachricht an alle Geräte gesendet wird. Dadurch wird die Netzwerklast reduziert. Des Weiteren ist zu berücksichtigen, dass Designbeschränkungen, die sich aus den Protokollen und Nachrichtenvermittlern ergeben, eine Einschränkung der Abdeckung zur Folge haben.

5.3 Adressierungsschema

Nachrichtenvermittler, engl. Message Broker, wie Apache ActiveMQ [170] oder RabbitMQ [21] sind Softwaredienste, die die Verteilung einer Nachricht basierend auf Warteschlangen und/oder Themen anbieten. Bei Warteschlangen werden Nachrichten zwischen Klienten mittels Eins-zu-eins-Kommunikation ausgetauscht; so erreicht eine Nachricht (in der Regel) genau einen Empfänger. Mithilfe von Themen können Nachrichten an eine potenziell größere Anzahl von Empfängern verteilt werden. Ein Klient sendet eine Nachricht zu einem Thema, die dann an alle interessierten Empfänger gesendet wird. Die Interaktionen,

die für die Implementierung dynamischer Mengen erforderlich sind, werden der themenbasierten Kommunikation zugeordnet, wie im Folgenden beschrieben.

Ein Thema kann als hierarchische Struktur dargestellt werden, die durch Ebenen unterteilt ist. Die Ebenen werden durch ein Trennzeichen, beispielsweise einen Punkt oder Schrägstrich, unterschieden. Interessenten können bestimmte Themen abonnieren. Werden Nachrichten an ein Thema gesendet, erhalten diese die registrierten Clients. Einige Protokolle und Broker bieten die Möglichkeit, Platzhalter (auch Wildcards genannt) im Themennamen zu verwenden. Dieses flexible Schema ermöglicht die Verwendung der Programmierabstraktion mit einer Vielzahl von Nachrichtenbrokern und Kommunikationsprotokollen.

Für die nachrichten-basierte Adressierung in Publish/Subscribe-Systemen werden maximal vier Themenebenen verwendet, die hier durch einen Punkt getrennt sind:

<Client-Typ>.<ID>.<Nachrichtentyp>[.<Schnittstelle>]

1. Der Kliententyp der ersten Ebene identifiziert den Typ des Netzwerkteilnehmers und ist auf dev und set beschränkt. Nach diesem Schema wird die Art des Empfängers festgelegt. Set ist die Abkürzung für *Dynamic Set* (dynamische Menge) und stellt für Aufrufe an Getter-Methoden zum Erhalt eines Zustandes von einem Gerät einen Datenkonsumenten auf Anwendungsseite dar. In dieser Terminologie ist ein Datenproduzent ein Gerät und wird mit dev abgekürzt.

2. Die ID der folgende Ebene schränkt den Empfänger mittels einer Kennung ein. Beispielsweise ist aus dem Topic set.123 bekannt, dass sich die Anfrage an die dynamische Menge mit der ID 123 richtet. Das Schlüsselwort all stellt einen speziellen Bezeichner dar. Eine Nachricht, die all als Bezeichner verwendet, erreicht alle registrierten Instanzen des angegebenen Typs.

3. Die dritte Ebene zum Nachrichtentyp enthält die Spezifikation des Kommunikationsmusters. Das verwendete Vokabular wird später in der Beschreibung der Kommunikationsmuster aufgezeigt.

4. Die Spezifikation der Schnittstelle hat den Vorteil, dass nur Produzenten erreicht werden, die sich durch den entsprechenden Status auszeichnen. Nach Bereitstellung von dynamischen Mengen und Geräten erfolgt deren Registrierung über Themenkanäle, die deren Typen, d.h. Schnittstelle, entsprechen. Darauffolgende Auswahlanfragen für andere Schnittstellen sind für den Produzenten irrelevant und werden aufgrund der Struktur der Themenhierarchie nicht entgegengenommen. Ohne dies würde eine Anfrage alle Produzenten erreichen und zu unnötigem Ressourcenverbrauch führen.

Der Aufbau eines Adressierungsschemas kann unterschiedlich ausgestaltet werden und dabei verschiedene Merkmale aufweisen: Ein grobkörnigeres Adressierungsschema ohne Adressierung des Nachrichtentyps erfordert die Übertragung

der beabsichtigten Aktion in der Nutzlast einer Nachricht selbst, was wiederum Netzwerklast erzeugt und Filterung und Fallunterscheidungen an empfangenden Klienten erfordert. Das Weglassen der Schnittstelle im Adressierungsschema zum Aufbau der Bindung zwischen dynamischen Mengen und Geräte würde außerdem zu einer redundanten Zustellung von Auswahlnachrichten führen, die wiederum beim Klienten gefiltert werden müssten. Im Gegensatz dazu könnte die Adressierung auch detaillierter und bis auf die Ebene der Methodensignatur erfolgen, sodass diese Informationen nicht in der Nachricht vorgehalten werden müssten. Allerdings kann die zunehmende Anzahl instanziierter Themen auf den Brokern eine zusätzliche Belastung für den Nachrichtendienst darstellen, was zu Skalierungsproblemen führen kann, die in Leistungsengpässen münden.

Eine Alternative zur hier angewendeten themen-basierte Adressierung ist das inhalts-basierte Routing in Publish/Subscribe-Systemen. Abonnenten definieren Interesse auf Basis komplexer Filterausdrücke, wie Attribute, Werte oder Muster, die auf den Inhalt der Nachrichten angewendet werden. Nach Verteilung im Publish/Subscribe-System halten Nachrichtenvermittler (Broker) die Filterausdrücke in Routing-Tabellen vor und leiten Nachrichten nach Evaluierung weiter. Für die vollständige Umsetzung des inhaltsbasierten Routings ist es erforderlich, dass die Broker zwischen Produzenten und Abonnenten die Funktionalität unterstützen. Um die Anzahl an Nachrichten und deren Filterung in einem System zu beherrschen sind die Anwendung effizienter Algorithmen und die Ausstattung der Broker mit skalierbaren Ressourcen erforderlich. Die Herausforderungen der Heterogenität unterschiedlich komplexer Filtersprachen und Datenformate können durch die Verwendung eines gemeinsamen Vokabulars und Datenschemata sowie deren Transformationsmöglichkeiten aufgelöst werden. Des Weiteren erfordert inhaltsbasiertes Routing Zugriff auf den Nachrichteninhalt, was bei vertraulichen Nachrichteninhalten in Bedenken bezüglich des Datenschutzes resultieren kann bzw. dieser durch Verschlüsselung nicht zur Verfügung steht.

5.4 Interaktionsmuster

Im Folgenden werden die erforderlichen Interaktionen einer Selektion sowie einer Pull- und Push-basierten Kommunikation erläutert. Abbildung 5.1 zeigt eine Übersicht über Publish/Subscribe-Aktionen an definierten Themen, die entsprechend ihrer Interaktionsgruppe gefärbt sind: Grün für Auswahl und Bindung, Blau für Datenaustausch und Orange für adaptive Kommunikation.

5.4.1 Initialisierung

Bei Bereitstellung eines Klienten – entweder einer dynamischen Menge oder eines Gerätes – werden dauerhafte Abonnements ausgelöst, um Auswahl- und Bindungsverfahren sowie Datenaustausch- und Kontrollnachrichten zu ermöglichen.

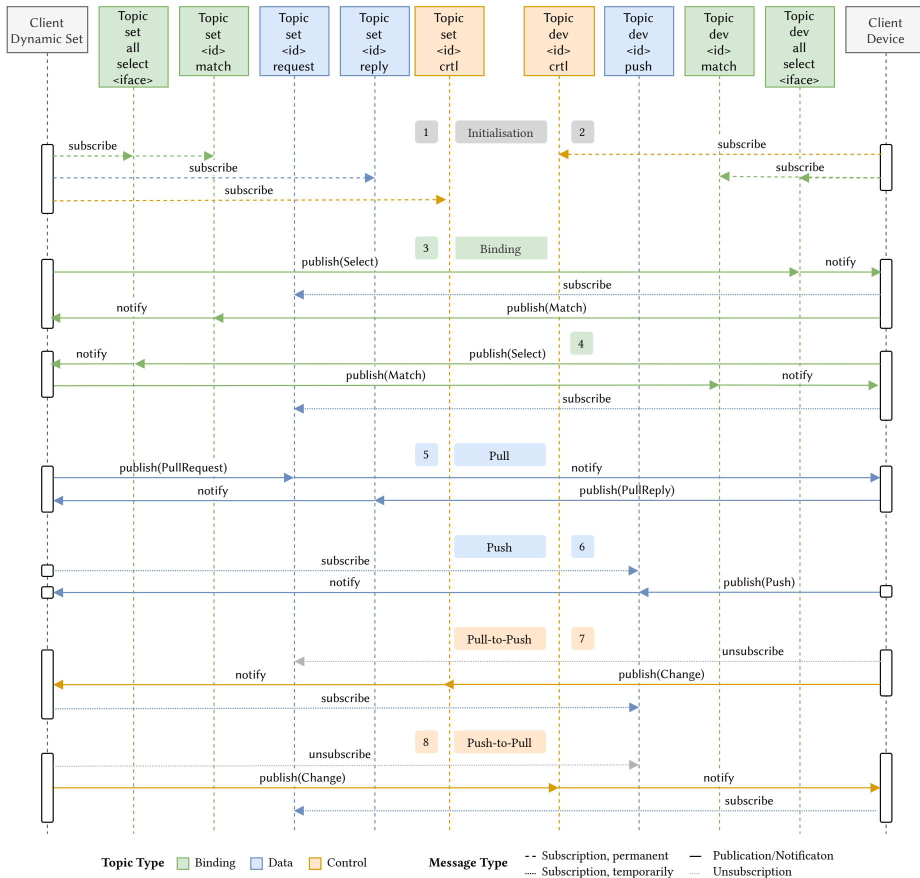


Abbildung 5.1: Aktivitätsdiagramm mit Interaktionsmustern abgebildet auf Publish/Subscribe-Themen

Dynamische Menge

Mit Bereitstellung einer dynamischen Menge (Datenkonsument) abonniert diese bei der Initialisierung ihre eigene Thema set.<id>.match, in dem Antwortnachrichten von Geräten empfangen werden, die die angegebenen Selektionskriterien im Auswahlverfahren erfüllen. Alle dynamischen Mengen abonnieren das einheitliche Thema dev.all.select.<interface>, um benachrichtigt zu werden, wenn neue Geräte (Datenproduzenten) bereitgestellt werden und ihren Status veröffentlichen. Eine dynamische Menge abonniert auch ihr eigenes AntwortThema set.<id>.reply, wo Antwortnachrichten auf Anfragen mittels Pull-Kommunikation empfangen werden. Darüber hinaus abonniert sie dauerhaft das eigene Kontroll-Thema set.<id>.control, um Nachrichten zu verarbeiten, die zur Änderung des angewandten Kommunikationsparadigmas verwendet werden.

Gerät

Ein Gerät abonniert bei der Initialisierung dauerhaft das universell quantifizierte Auswahl-Thema der Schnittstelle(n) set.all.select.<interface>, um Auswahlanforderungsnachrichten von dynamischen Mengen zu empfangen. In umgekehrter Reihenfolge, wo bestehende dynamische Mengen bei ihrer Initialisierung mit dem Gerätestatus informiert werden, registriert sich ein Gerät bei seinem Thema dev.<id>.match, um passende Antwortnachrichten zu erhalten. Ein Gerätes registriert sich auch bei seinem eigenen Kontrollthema dev.<id>.control, um mit Statistik- sowie Wechselnachrichten für das Kommunikationsparadigma benachrichtigt zu werden.

5.4.2 Bindung

Bindung beschreibt den Prozess, bei dem sich eine dynamische Menge und ein Gerät gegenseitig auf Anforderungen einigen, entweder ausgelöst durch eine Auswahl aus einer dynamischen Menge oder durch ein Statusangebot eines Geräts.

Geräteauswahl

Eine Auswahl wird durch eine dynamische Menge ausgelöst, um nach vorhandenen Geräten abzufragen, die seinen Einschränkungen entsprechen. Die Auswahl der Geräte erfolgt bei der Initialisierung einer dynamischen Menge und auch wiederholt später zur Laufzeit, um beispielsweise zu prüfen, ob Geräte die Auswahlkriterien noch oder neu erfüllen.

Zunächst veröffentlicht eine dynamische Menge eine Nachricht an das universelle Thema dev.all.select.<interface>, um alle Geräte anzusprechen, die auch die Schnittstelle implementieren. Die auf Prädikatenlogik basierenden Auswahlbeschränkungen für Methoden der Schnittstellen sind in der Nachrichtennutzlast enthalten. Geräte, die die angeforderte Schnittstelle implementieren, werden mit der Auswahlnachricht benachrichtigt und werten die Auswahlbeschränkungen lokal anhand ihres eigenen Status aus.

Bei einer Übereinstimmung fügt der Geräte-Stellvertreter die Referenz mit Adressierung zu der dynamischen Menge in dessen lokalen Datenstruktur hinzu und abonniert das Thema set.<id>.request der dynamischen Menge, um standardmäßig die Anfragen über Pull-Kommunikation zu verarbeiten. Das Gerät veröffentlicht den Erfolg auch im Übereinstimmungs-Thema der dynamischen Menge set.<id>.match, das sich ausschließlich an den Anfragenden richtet. Beim Empfang erkennt die dynamische Menge, dass das Gerät die Auswahlanforderung erfüllt, und übernimmt die Referenz auf das Gerät lokal in die Mitgliederliste. Abb. 5.1/3 zeigt die zuvor beschriebenen Abonnements.

Anbieten von Geräten

Ein Geräte-Stellvertreter veröffentlicht eine Auswahlnachricht mit seinem aktuellen Status im universellen Thema set.all.select.<interface>, die an bestimmte dynamische Mengen weitergeleitet wird, welche die gleiche Schnittstelle implementiert und sich bei der Initialisierung für das Thema dauerhaft abonniert haben.

Die benachrichtigten dynamischen Mengen prüfen lokal den übermittelten Gerätestatus anhand ihrer Auswahleinschränkungen. Bei Übereinstimmung wird das Gerät als Mengenmitglied in die lokale Datenstruktur zur Mitgliederverwaltung hinzugefügt (sofern es nicht bereits Mitglied ist) und das Hinzufügen wird dem Gerät durch Veröffentlichung einer Antwortnachricht im Thema dev.<id>.match bestätigt. Bei Empfang der Übereinstimmungsnachricht, abonniert das Gerät das Anfragethema der dynamischen Menge set.<id>.request, um über Anfragenachrichten mithilfe der Pull-Kommunikation benachrichtigt zu werden (siehe Abbildung 5.1/4).

5.4.3 Datenkommunikation

Für den Datenaustausch über das Anfrage-/Antwort-Kommunikationsmuster sowie für Statusaktualisierungen werden Themen zur Adressierung eingeführt, die Pull- und Push-basierte Kommunikation abdecken.

Pull-Kommunikation

Eine Pull-Kommunikation besteht aus einer Multicast-Anfrage und mehreren Unicast-Antworten von erreichten Empfängern (vgl. Abschnitt 4.2.4). Die Interaktionssequenz, in Abbildung 5.1/5 dargestellt, wird durch eine Nachricht von einer dynamischen Menge an ihr eigenes Anfragethema set.<id>.request initiiert, die von dem Nachrichtendienst an alle passenden Geräte weitergeleitet wird, die das Thema zuvor abonniert haben. Nachdem eine Anfrage auf einem Gerät verarbeitet wurde, wird eine Antwortnachricht an das Antwortthema der dynamischen Menge veröffentlicht set.<id>.reply. An der dynamischen Menge werden Antwortnachrichten der Anfrage gruppiert, aggregiert und an die Geschäftslogik des Clients zurückgegeben.

Push-Kommunikation

Im Gegensatz zur Pull-Kommunikation wird die Push-Kommunikation nicht durch eine dynamische Menge, sondern durch Zustandsänderungen eines Geräts ausgelöst und an alle registrierten dynamischen Mengen gesendet (vgl. Multicast Push, Abschnitt 4.2.3). Push-Kommunikation wird nur für Schnittstellenmethoden verwendet, die den Gerätezustand nicht ändern, d.h. Getter-Methoden.

Ein Gerät veröffentlicht seine Statusänderung in seinem Thema dev.<id>.push und die Nachricht wird vom Nachrichtendienst an die dynamische Mengen weitergeleitet, die Statusaktualisierungen des Gerätes zuvor abonniert haben (Abbildung 5.1/6). Der Stellvertreter der dynamischen Menge speichert den neuesten Gerätestatus aller Geräte lokal, die Mitglieder der Menge sind. Aus diesen zwischengespeicherten Zuständen wird dann ein aggregierter Rückgabewert abgeleitet für nachfolgende Aufrufe aus der Anwendung.

5.4.4 Adaptivität

In Abschnitt 4.4 wurden drei Heuristiken vorgestellt, die das Kommunikationsparadigma ändern, das zwischen einer dynamischen Menge und einem Gerät für Methoden angewendet wird. Die Heuristik ändert das Paradigma entweder für einzelne Verbindungen zwischen Geräten und dynamischen Mengen oder für alle Verbindungen zwischen einem bestimmten Gerät und allen seinen assoziierten dynamischen Mengen.

Zusätzliche permanente Abonnements werden eingerichtet, um eine adaptive Kommunikation zu implementieren: Während der Initialisierung registrieren sich Klienten zusätzlich für das eigene Kontrollthema, d.h. eine dynamische Menge für set.<id>.ctrl bzw. ein Gerät für dev.<id>.ctrl, um über Unicast gerichtete Kontrollnachrichten benachrichtigt zu werden, die von der angewendeten Heuristik ausgelöst werden.

In den Sequenzen zur Bindung zwischen dynamischen Mengen und Geräten enthalten die initialen, bei Bereitstellung veröffentlichten Auswahlnachrichten von dynamischen Mengen sowie die Angebotsnachrichten von Geräten zusätzlich Daten über die Bezeichnungen verfügbarer Heuristiken und der vorgeschlagenen Variante mit dem beabsichtigten Kommunikationsparadigma. Die linkbasierte Heuristik h_{lnk} (vgl. Abschnitt 4.4.1) operiert verbindungsindividuell und sowohl die dynamische Menge als auch das Gerät entschieden über das Kommunikationsparadigma. Bei den anderen beiden Heuristiken h_{dev} und h_{hyb} (vgl. Abschnitte 4.4.2 und 4.4.3) gibt ausschließlich das Gerät das Kommunikationsparadigma vor, da es für eine Menge von registrierten Anwendungen entscheidet. Insofern nach angewendeter Heuristik und Paradigma der Verbindung zuständig, stellt ein Empfänger die Initialkommunikation für die Verbindung ein und meldet sich bei den zugehörigen Themen für Pull- oder Push-Kommunikation an. Darüber hinaus konfiguriert ein Klient korrekte Auswertungsverfahren lokal, wendet diese periodisch an löst entsprechend der angewendeten Heuristik Kontroll- und Statistikmeldungen aus.

Von Pull nach Push

Eine Umstellung von Pull- auf Push-Kommunikation wird durch den Geräte-Stellvertreter ausgelöst und erfordert folgende Änderungen:

(i) Das Gerät meldet sich vom Anforderungsthema set.<id>.request der dynamischen Menge ab, um keine Nachrichten aus Multicast-Anfragen über Pull-Kommunikation zu empfangen.

(ii) Es veröffentlicht eine Kontrollnachricht mit den lokalen Push-Statistiken im Kontrollthema set.<id>.control der dynamischen Menge. Bei deren Empfang aktualisiert die dynamische Menge die abgerufenen Push-Statistiken lokal und

(iii) abonniert das Push-Thema dev.<id>.push des Geräts, um über Datenänderungen benachrichtigt zu werden. Für den Fall, dass ein Paradigmenwechsel durch eine Heuristik an einer dynamischen Menge ausgelöst wird, ist die zuvor beschriebene Reihenfolge umgekehrt. Hier wird eine Steuernachricht aus Schritt ii) von der dynamischen Menge an das Steuerthema dev.<id>.control des Geräts gesendet, wie in Abbildung 5.1/7 dargestellt.

Von Push nach Pull

Im Rahmen der Umstellung von Push- auf Pull-Kommunikation ist eine umgekehrte Durchführung der im vorherigen Abschnitt präsentierten Modifikationen erforderlich. Wenn eine dynamische Menge den Wechsel auslöst, meldet sich ihr Stellvertreter zunächst vom Push-Thema des potenziellen Geräts dev.<id>.push ab. Anschließend wird eine Steuernachricht mit Wechselabsicht und Pull-Statistiken an das Steuerungsthema des Gerätes dev.<id>.ctrl gesendet. Daraufhin abonniert das Gerät beim Empfang der Steuernachricht das Anfragethema der dynamischen Menge set.<id>.request, um über Pull-Anfragen benachrichtigt zu werden, siehe Abbildung 5.1/8.

5.4.5 Beendigung

Vor einem geplanten Herunterfahren senden dynamische Mengen sowie Geräte eine Abschlussnachricht, um die beidseitige Buchhaltung über Mengenmitglieder synchron zu halten. Um dies zu erreichen, werden im Vergleich zu den Initialisierungs- und Bindungssequenzen umgekehrte Vorgänge ausgeführt: Bei der Dekonstruktion veröffentlicht der Stellvertreter eines Gerätes eine Abschlussnachricht an jedes Steuerthema der betroffenen dynamischen Mengen: set.⟨id⟩.ctrl. Die benachrichtigten dynamischen Mengen melden sich entweder vom Antwortthema dev.⟨id⟩.reply oder vom Push-Thema dev.⟨id⟩.push des Geräts ab – je nach dem zuletzt angewendeten Kommunikationsparadigma. Ein Gerät meldet sich auch vom Anforderungsthema set.⟨id⟩.request einer dynamischen Menge ab, falls das Kommunikationsparadigma derzeit Pull ist. Abschließend meldet sich das Gerät von seinem Kontrollthema dev.⟨id⟩.ctrl ab, damit der Messaging-Dienst das abgemeldete Thema bereinigen kann.

Wenn der Mengen-Stellvertreter durch seine Anwendung heruntergefahren wird, werden die die gleichen Abmeldungen wie zuvor beschrieben durchgeführt. Die

anfängliche Abschlussnachricht wird von der dynamischen Menge an jedes Gerät, das in ihr Mitglied ist, an dessen Steuerungsthema dev.⟨id⟩.ctrl publiziert. Um unerwartete Abschaltungen von Klienten oder Netzwerkstörungen abzudecken können dynamische Mengen wiederholt Selektionsroutinen durchführen oder Geräte ihren aktualisierten Status anbieten, um aus zurückkommenden Übereinstimmungsnachrichten zu schlussfolgern, dass ein Mitglied weiterhin aktiv ist. Diese Vital-Nachrichten (*heartbeat*) werden entsprechend der Laufzeitkonfiguration ausgelöst, wenn eine Datenanfrage (Request) von einer dynamischen Menge über eine definierte Zeitspanne ausbleibt oder wenn eine bestimmte Zeit keine Statusaktualisierung (Push) von einem Gerät empfangen wurde.

5.5 Kommunikationsprotokoll

Es werden die Nachrichtentypen vorgestellt, die zur Umsetzung der eingeführten Interaktionsmuster und Heuristiken notwendig sind. Nachrichtenprotokolle verfügen über unterschiedliche Möglichkeiten zum Speichern von Adressierungs- und Metadaten.

Tabelle 5.1 zeigt einen Auszug von Kriterien ausgewählte Protokolle MQTT, CoAP, AMQP und HTTP, wie sie in der IoT-Domäne Anwendung finden [106]. Die abweichende Abstraktion sowie Semantiken/Methoden zeigen die Notwendigkeit der Adaption von Anwendungsaufrufen auf Protokollspezifika auf. Dynamische Mengen fragen bspw. mit der Methodensignaturen einer Geschäfts-Schnittstelle an, welche in die Aufrufprozeduren je Protokoll bzw. die Konfiguration zur Dienstgüte übersetzt werden.

MQTT [9, 8] und AMQP [61] sind Protokolle, die in verschiedenen Umfragen und vergleichenden Bewertungen zu IoT-Protokollen berücksichtigt wurden [107, 106, 153, 149]. Weiterhin sind CoAP [150] und DDS [115] in vielen Veröffentlichungen einbezogen. MQTT 3.1.1 und AMQP 1.0 werden in den folgenden Abschnitten als Protokolle referenziert, um Unterschiede aufzuzeigen Daten und Metadaten in bestimmten Kopffeldern (Header) zu transportieren.

5.5.1 Nachrichtentypen

Zur Umsetzung der Interaktionen und des zugehörigen Adressierungsschemas werden sechs Nachrichtentypen verwendet (vgl. Abb. 5.2): Select, Match und Shutdown dienen zum Binden und Entbinden, PullRequest, PullReply und Push dienen dem Daten- und Statistikaustausch sowie Änderungen zur adaptiven Umschaltung.

Abbildung. 5.2 zeigt ein Klassendiagramm der Nachrichtentypen und ihrer zugehörigen Attribute. Die folgenden Metadaten-Attribute werden zur Implementierung aller Interaktionen verwendet: Quell-ID srcId, Anforderungs-ID reqId, Korrelations-ID corrId und Eigenschaft property. Der Nachrichtentyp ist nicht

Kriterium	MQTT	CoAP	AMQP	HTTP
Architektur				
Client/Broker	C/B	C/B	C/B	
Client/Server		C/S	C/S	C/S
Abstraktion				
Request/Response		R/R	R/R	R/R
Publish/Subscribe	P/S	P/S	P/S	
Semantiken/ Methoden	Connect Disconnect Publish Subscribe Unsubscribe Close	Get Post Put Delete	Consume Deliver Get Select Ack Delete Nack Recover Reject Open Close	Get Post Head Put Patch Options Connect Delete
Dienstgüte				*Limited (via TCP)*
At-most-once	QoS 0	*Confirmable Message*	*Settle Format*	
At-least-once	QoS 1	*Non-Conf. Message*	*Unsettle Format*	
Exactly-once	QoS 2			

Tabelle 5.1: Vergleichsanalyse Nachrichtenprotokolle, Auszug aus [106]

als zusätzliches Attribut erforderlich, da der Typ bereits in der Adressierung der Nachricht im Thema vorhanden ist oder anhand der Struktur der Nutzlast einer Nachricht unterschieden werden kann. Die Quelle ist in jeder Nachricht angegeben.

Für die Auswahl und das Angebot werden die Nachrichtentypen Select und Match verwendet. Eine ausgewählte Nachricht verfügt über eine Anforderungskennung im reqId-Attribut. Die Antwort auf die Auswahl enthält diesen Bezeichner im corrId-Attribut. Ein Mechanismus zur Generierung von Identifikatoren kann entweder lokal oder über eine zentrale Stelle vorhanden sein. Darüber hinaus können beide Nachrichtentypen auch Statistiken enthalten, falls adaptive Kommunikation verwendet wird. Um dies zu erreichen, gibt das Attribut para das aktuell angewendete Kommunikationsparadigma beim Absender an, die Bezeichnung der Heuristik, die der Absender aktiv verwendet oder zu der er wechseln kann, und der Umfang bestimmt die Granularitätsebene der Heuristik, um die Datenstrukturen für die Auswertung korrekt einzurichten. Bei Auswahl oder

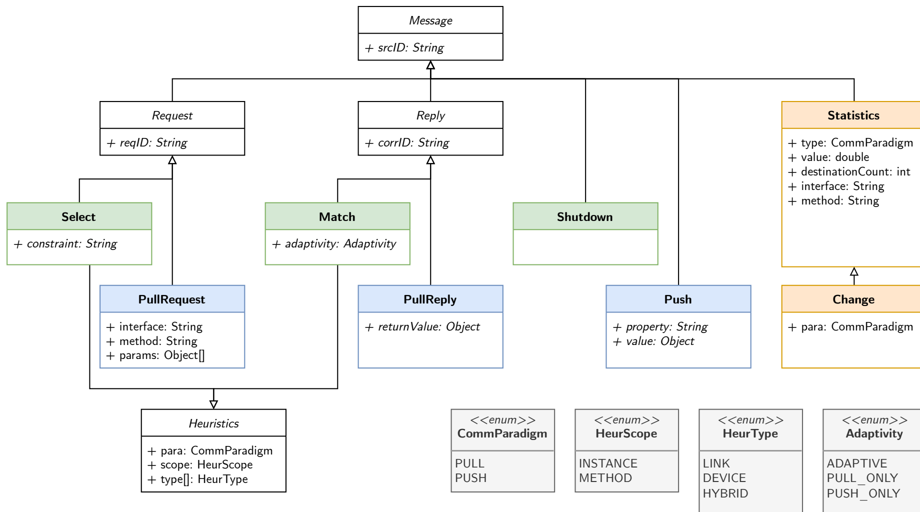


Abbildung 5.2: Klassendiagramm der Nachrichtentypen

Angebot initialisieren benachrichtigte Klienten, die den Kriterien entsprechen, die Kommunikation gemäß einer gegenseitigen Reihenfolge der Heuristiken.

Für die Pull-Kommunikation sind die Nachrichtentypen Request und Reply von entscheidender Bedeutung. Es ist notwendig die zu einer Anfrage gehörende Antwort zu identifizieren. Dafür wird die Anfragekennung im Attribut reqId und in der Antwort im Feld corrId angegeben. Da es sich um einen Remote-Prozeduraufruf handelt, werden in der Nachricht auch die Felder iface für den Klassennamen, method für den Methodennamen und params für die Parameter gesetzt. Der Rückgabewert der ausgeführten Methode wird im Attribut returnValue an den Datenkonsumenten gesendet. Für die Push-Kommunikation ist lediglich der Nachrichtentyp Push erforderlich. Eine Push-Nachricht enthält den Namen der aktualisierten Eigenschaft im property-Feld und ihren Wert im value-Feld.

Die Nachrichtentypen Statistics und Change dienen dem Austausch heuristik-relevanter Daten bzw. der Ermöglichung einer adaptiven Kommunikation. Die Felder iface für den Klassennamen, method für den Methodennamen und params für die Parameter werden auf die gleiche Weise wie bei der Anfrage der Pull-Kommunikation verwendet. Darüber hinaus gibt Typ den Statistikeintrag an, während Wert die normalisierte und geglätteten Statistikzähler zu Interaktion enthält. Der destinationCount gibt die Mitgliedssatzgröße eines dynamischen Satzes oder Geräts an, damit die Heuristik auf der Empfängerseite den verbindungsbasierten Anteil mithilfe der jeweiligen Kostenfunktion berechnen kann. Der geerbte Nachrichtentyp Change teilt dem Empfänger zusätzlich mit para das zukünftige Kommunikationsparadigma mit. Mit einer Shutdown-Nachricht

informiert der Absender über die geplante Abschaltung. Es enthält als einzige Information die in allen Nachrichten enthaltene srcID.

5.5.2 Protokollabbildung

Ein Protokoll muss in der Lage sein, die in Abbildung 5.2 dargestellten Attribute zu übermitteln. Die zusätzlichen Attribute sind entweder im Kopfbereich (*header*) oder in der Nutzlast (*payload*) einer Nachricht enthalten. Individuelle Datenfelder werden zur besseren Analyse und Auswahl des Netzwerkverkehrs genutzt. Die Auswahlbedingung, den Schnittstellennamen, den Methodennamen und die Parameter für einen entfernten Prozeduraufruf sowie die Rückgabewerte einer Antwort- und Push-Nachricht werden vorgeschlagen in der Nutzlast zu senden. Die Bezeichner srcId, reqId, corrId und der Eigenschaftsname werden als separate Kopffelder angegeben. AMQP bietet die Möglichkeit, benutzerdefinierte Schlüssel-Wert-Paare hinzuzufügen, während MQTT 3.1.1 diese Funktion nicht unterstützt. Beide Protokolle dienen als Beispiele für die Umsetzung des Konzepts in ein leichtes und flexibles Protokoll.

Bei MQTT ist es notwendig, Metainformationen der Nutzlast der Nachricht zuzuordnen. Nur das Thema wird im dafür vorgesehenen Header-Feld platziert. Alle anderen Daten werden abhängig vom gewählten Serialisierungsmechanismus in der Nutzlast kodiert. Im Gegensatz dazu besteht bei AMQP die Möglichkeit, neben standardisierten Header-Feldern auch eigene Eigenschaften hinzuzufügen. Drei der fünf erforderlichen Metainformationen können im Eigenschaftenbereich einer Nachricht festgelegt werden: Das Feld srcId wird dem Feld user-id zugeordnet. Das Attribut message-id erfüllt auch den vorgesehenen Zweck für die Anforderungskennung. Eine AMQP-Nachricht verfügt über das Korrelations-ID-Feld für den Korrelationsbezeichner. Das reqFrom-Attribut, das die Kennung der Menge in einer Auswahlnachricht enthält, kann keinem Feld zugewiesen werden und wird daher im Kopfbereich der Anwendungseigenschaften festgelegt. Zusätzlich wird in diesem Bereich bei einer Push-Nachricht der Name des Attributes angegeben für den der Rückgabewert übermittelt wird. Zusätzlich zur Zuordnung der erforderlichen Metainformationen kann das Antwortfeld mit der Kennung der Quelle in einer Anforderungsnachricht festgelegt werden. Das Feld steht hinter der Angabe der Adresse des Knotens, der die Antworten empfängt und erfüllt somit den vorgesehenen Verwendungszweck. Die restlichen Informationen sind in der Nutzlast kodiert. Im Falle von z.B. eine Anfragenachricht, erfordert dies die Zuordnung der Daten zu den Schlüsseln iface, method und params.

5.6 Serialisierung

Bei der Übermittlung von Daten ist darauf zu achten, dass sowohl Sender als auch Empfänger die Daten richtig interpretieren. Der Absender serialisiert Objekte oder Datenstrukturen in ein geeignetes Austauschformat und überträgt

Listing 5.1: Nachrichtendeklaration in Protocol Buffers

```
message SerializableMessage {
  string srcId = 1;
  oneof Type {
    SELECT select = 2;
    MATCH match = 3;
    REQUEST request = 4;
    REPLY reply = 5;
    PUSH push = 6;
    CHANGE change = 7;
    STATS stats = 8;
  }
  // display message types of request/reply
  message REQUEST {
    string reqId = 1 ;
    string iface = 2 ;
    string method = 3 ;
    repeated Argument params = 4;
  }
  message REPLY {
    string corrId = 1 ;
    Argument returnValue = 2;
  }
  // further message types skipped here
  message Argument {
    oneof data {
      double double = 1;
      float float = 2;
      int32 int = 3;
      bool boolean = 4;
      string string = 5;
      google.protobuf.Any any = 6;
    }
  }
}
```

über das Netzwerk. Der Empfänger deserialisiert die empfangen kodierte Daten, die möglicherweise in einer anderen Programmiersprache als der Sender geschrieben sind, um semantisch identische Daten wiederherzustellen.

Einige Programmiersprachen verfügen über eine integrierte Serialisierung, sind jedoch auf ihre eigene Sprache beschränkt (z. B. Java-Objektserialisierung oder XML-basierte Serialisierung in .NET [100]). Die JavaScript Object Notation (JSON) [20] ist ein leichtes Textformat zum Austausch strukturierter Daten. In Kombination mit der GSON [62]-Bibliothek können Java-Objekte in JSON (de)serialisiert werden und umgekehrt. Mit sprachunabhängigen Serialisierungen können Entwickler die Datenstruktur in einer von der Programmiersprache unabhängigen Notation deklarieren (z. B. in einer separaten Datei) nach einer *Interface Definition Language (IDL)*, vgl. Abschnitt 2.1.3.

Abbildung 5.2 zeigt die erforderlichen Metainformationen und ob diese in den Header der Nachricht (Attribute in Kursivschrift) oder in die Nutzlast (normal) eingefügt werden. Die Nutzlast enthält je nach Nachrichtentyp die Auswahlkriterien, den Rückgabewert, Parameter eines RPC oder den Wert einer Eigenschaft. Erforderliche Informationen, die nicht in den Header integriert werden können, werden in die Nutzlast verschoben. Abhängig vom Transportprotokoll unterscheidet sich die Datenstruktur, die serialisiert und im Nachrichtentext übertragen werden soll. Im Gegensatz zu einer AMQP-Nachricht enthält die Nutzlast einer MQTT-Nachricht die Informationen srcId und reqId, da diese nicht in den Header gestellt werden können.

Die Struktur eines entfernten Methodenaufrufes (RPC) beschränkt die Übertragung nicht auf primitive Datentypen. Für die Serialisierung und Deserialisierung anwendungsspezifischer Datentypen ist die Verwendung eindeutiger Datentypkennungen seitens des Senders und Empfängers erforderlich. Dies ist zum Beispiel der Fall, wenn eine Methode mit einer Zeichenkette (String) und einem Ganzzahl-Parameter (Integer) aufgerufen wird. Ohne Angabe eines Datentyps kann der Empfänger nicht erkennen, ob der zweite Parameter ein Integer, Float oder Double ist. Um dieses Problem zu vermeiden, werden die Datentypen vor der Kommunikation definiert. Bei protobuf muss für jedes mit Any gekennzeichnete Feld der Datentyp angegeben werden.

Analog zur Java-Objektserialisierung wird bei der Definition der Nachrichtenstruktur von Protokollpuffern eine Differenzierung der Nachrichtentypen vorgeschlagen. Listing 5.1 enthält einen Auszug in der Sprache *proto3* [63]. Es gibt eine abstrakte Nachricht, die die einzelnen Typen enthält. Jede einzelne Instanz besitzt die für den Nachrichtentyp erforderlichen Metainformationen. Die einzelnen Werte werden in *proto3* durch eine eindeutige Nummer identifiziert. Sie wird zur Transformation in das binäre Nachrichtenformat verwendet. Oneof verhindert, dass in einer Nachricht mehrere Typen festgelegt werden. Da die Quellinformationen für alle Nachrichtentypen gesendet werden, wird diese Kennung keinem speziellen Typ zugewiesen.

5.7 Architektur

Die nachrichtenbasierte Protokollabstraktion stellt die Verbindung zwischen den Stellvertretern der Anwendungen oder Geräte, die Daten senden oder empfangen wollen, und den Netzwerkklienten dar, die die entsprechenden Prozeduren eines Transportprotokolls implementieren, und mit entfernten Teilnehmern so über Publish/Subscribe interagieren. Abbildung 5.3 zeigt schematisch die relevanten Teile der Middleware-Architektur.

Der Stellvertreter einer dynamischen Menge hat verschiedene Möglichkeiten, die Protokollabstraktion aufzurufen, wie z.B. die eigene Registrierung, Selektionsanfragen und Methodenaufrufe. Durch eine übergebene Callback-Referenz im

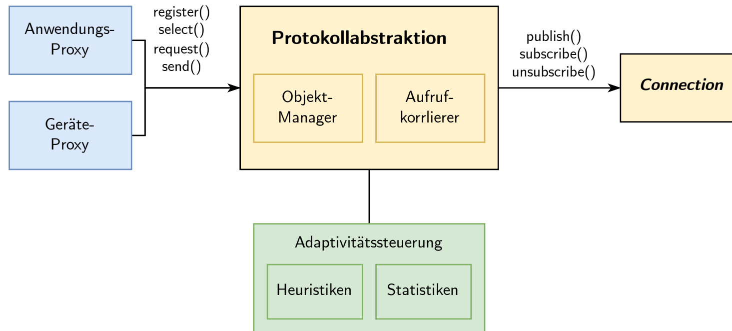


Abbildung 5.3: Schematische Darstellung der Transportabstraktion

Parameter werden Antworten an die Anwendung zurückgegeben. Analog dazu kann sich ein Gerät auch registrieren und Statusaktualisierungen veröffentlichen.

Der Methodenaufruf enthält neben den Nutzdaten wie der Schnittstellenbezeichnung, der Methodensignatur und den zugewiesenen Werten weitere Metadaten, die zur internen Verarbeitung benötigt werden. Diese können Ausführungsstrategien wie Dienstgüteeigenschaften und Aggregationsanweisungen enthalten.

Bei der Initialisierung wird eine eindeutige Senderkennung für den lokalen Klient erzeugt und Datenstrukturen für die dynamische Menge oder das Gerät angelegt, in der Referenzen auf entfernte Kommunikationspartner gehalten werden. Neben der Identifikation der Gegenseite und des verwendeten Transportklienten werden Zwischenspeicher (Caches) angelegt, die Werte und Ankunftszeiten früherer Methodenaufrufe (Getter-Methoden) halten, um diese ohne entfernte Aufrufe zurückzugeben, wenn sie die Wertegültigkeit gemäß der Aufrufkonfiguration erfüllen. Ein Korrelationsmanager verwaltet gesendete, unbeantwortete Aufrufe, deren eingehende Antwortnachricht den Eintrag löschen. In der Folge wird der Rückgabewert an die Aggregationsprozedur übergeben, welche unter Berücksichtigung der Ausführungsstrategie mehrere Rückgabewerte mit Hilfe einer Aggregationsfunktion zu einem zusammenfasst, der schließlich an den Klienten zurückgegeben wird. Des Weiteren werden Datenstrukturen für Zählstatistiken initialisiert. Dabei wird für die lokalen Interaktionen des Klienten, d.h. Anfragen einer Anwendung oder Aktualisierungen eines Gerätes, sowie für jede Verbindung jeweils eine Statistik zum Zählen entfernter Interaktionen erstellt (vgl. Abschnitt 4.5.2).

Die Protokollabstraktion wandelt den Methodenaufruf in universelle Daten für die spätere Adressierung und Umsetzung einer Nachricht des verwendeten Protokolls. Für die Adressierung wird das hierarchische Thema nach Abschnitt 5.3 gebildet. Je nach Nachrichtentyp wird die eigene ID oder die des adressierten Partners verwendet (vgl. Abschnitt 5.4). Die Konfiguration von Transportprotokollen führt zu einem Aufruf von Publish/Subscribe-Methoden an aktiven Instanzen. Die jeweilige Call-Implementierung bietet Methoden zum Publizieren von Nach-

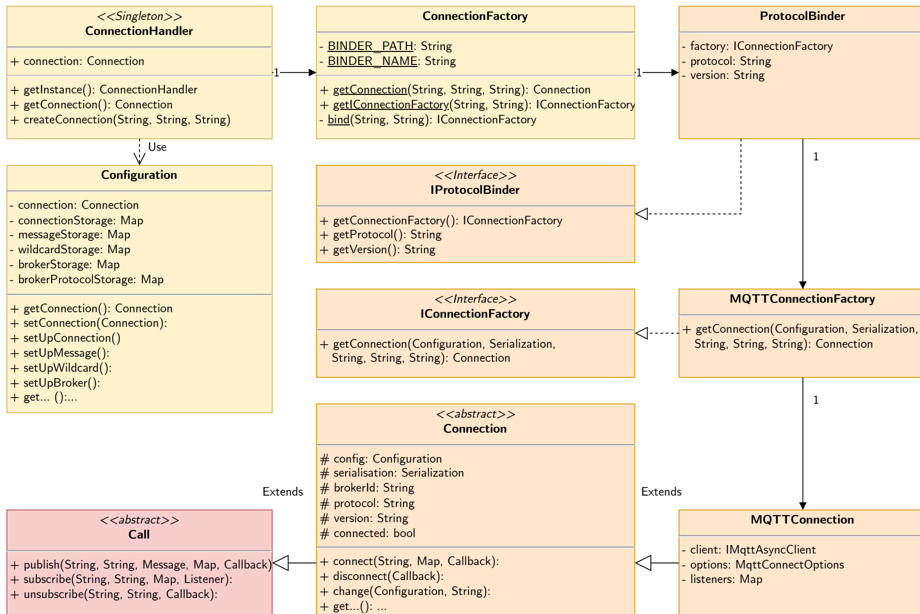


Abbildung 5.4: Klassendiagramm zur Protokollbindung

richten, zum Abonnement- und Stornierungsmanagement eines bestimmten Themas an.

Abbildung 5.4 zeigt ein UML-Klassendiagramm mit einem Ausschnitt für die Integration eines neuen Transportprotokolls, wie es prototypisch in der Programmiersprache umgesetzt wurde. Pro Protokoll ist die Implementierung der abstrakten Klassen Connection und Call erforderlich. Die gezeigte Abdeckung von MQTT erfordert die Bereitstellung eigener Klassen, hier MQTTConnection und MQTTConnectionFactory (siehe Klassen auf der rechten Seite). Die Klasse ProtocolBinder enthält die Felder protocol, version und connectionFactory, die mittels Reflexion beim Einbinden ausgewertet werden. Eine ConnectionFactory generiert eine Instanz der Protokoll-Implementierung. Da bei Verwendung mehrerer Protokolle gleichnamige Klassen existieren, werden die Namen durch die zugewiesene Paketstruktur eindeutig gehalten.

5.8 Evaluierung

Die Protokollabstraktionen wurden als Teil des Middleware-Frameworks für dynamische Mengen in Java 11 implementiert. Exemplarisch wurden Protocol Buffers und GSON als Serialisierungsformate sowie AMQP und MQTT als zugrunde liegende Publish/Subscribe-Nachrichtenprotokolle unterstützt. Um die Implementierung zu evaluieren, wurde ein Setup aus 20 Anwendungsklienten

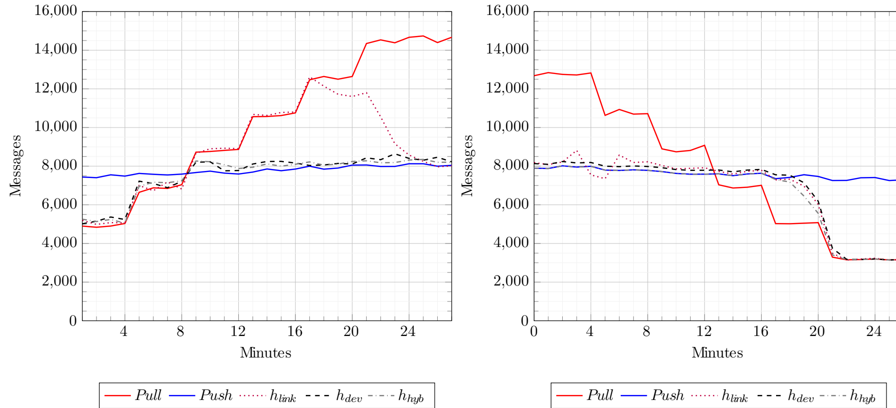
(dynamische Mengen) und 20 Geräten entworfen, die mit einem Apache ActiveMQ Message Broker [170] der Version 5.16.6 verbunden sind. Es wurde die Kombination aus GSON-Serialisierung und den MQTT-Transportkonnektor für die Verwendung in den Experimenten konfiguriert. Jeder Anwendungsklient erstellt eine dynamische Menge, um gleichzeitig mit fünf zufällig ausgewählten Geräten zu interagieren. Sowohl Anwendungsklienten als auch Geräte werden als Poisson-Prozesse modelliert, die durch λ_{get} bzw. λ_{set} gekennzeichnet sind. Die Pull-Rate λ_{get} definiert, wie oft ein Klient seine dynamische Menge nach Datenaktualisierungen abfragen soll, während die Push-Rate λ_{set} angibt, wie viele Datenaktualisierungen voraussichtlich von einem einzelnen Gerät erzeugt werden.

In den folgenden Experimenten variieren die Pull-Rate λ_{get}, während die Push-Rate $\lambda_{set} = 1s^{-1}$ konstant gehalten wird. Auf diese Weise wird eine Anpassung des Nachrichtenschemas von Pull zu Push und umgekehrt ausgelöst. Nach einer Aufwärmphase von 120 Sekunden beginnt die Messung der Anzahl der Nachrichten pro Minute, die zwischen Broker und Klienten ausgetauscht werden, wobei sowohl Anwendungen als auch Geräte umfasst werden. Jede Messung wird zehnmal wiederholt und die Durchschnittswerte angegeben. Das Auslesen eingehender Nachrichten den Kanälen auf den ActiveMQ-Brokern sowie das Zurücksetzen der Zähler erfolgte periodisch über JMX-Aufrufe[1]. In Kombination mit den Zählungen an den Klienten wurden die Hops im Netzwerk zusammengerechnet und aufgeschlüsselt nach Pull (Themen request und reply), Push (push), Kontrollnachrichten (ctrl) sowie der Addition aller zu den Gesamtnachrichten im Betrachtungszeitraum.

Im ersten Experiment wird die Pull-Rate $\lambda_{get} \in \{0.2, 0.3, \ldots, 0.7\}$ schrittweise alle vier Minuten erhöht. Daher wird jeder dynamische Satz häufiger nach Datenaktualisierungen gefragt. Auf diese Weise wird die erwartete Zwischenankunftszeit der exponentiell verteilten Anfragen von einer Anfrage alle 5 Sekunden auf eine Anfrage in etwa 1,5 Sekunden reduziert. Die Ergebnisse sind in Abb. 5.5a grafisch dargestellt. Die rote Kurve stellt eine reine Anfrage-basierte Interaktion (Pull) ohne Anpassung dar. Sein stufenförmiger Verlauf zeigt deutlich, wann im Experiment die Anforderungsrate erhöht wird. Offensichtlich macht eine steigende Anforderungsrate ein Pull-basiertes Interaktionsschema im Hinblick auf die ausgetauschten Nachrichten teurer. Im Gegensatz dazu misst die blaue Kurve die Anzahl der Nachrichten in einer reinen aktualisierungs-basierten Konfiguration (Push) und bleibt nahezu konstant. Dies liegt daran, dass sich die Push-Rate während des Experiments nicht ändert.

Dennoch ist weiterhin ein leichter Anstieg der Meldungen erkennbar. Diese Meldungen sind auf die wachsende Zahl der an den Broker gesendeten Anfragen zurückzuführen. In unserem Publish/Subscribe-Protokoll-Mapping werden alle Anfragen immer an den Broker veröffentlicht, jedoch weder an die Geräte weiter-

[1] Java Management Extensions: Spezifikation zur Verwaltung und Überwachung von Java-Anwendungen



(a) Umschaltung von Pull nach Push für ansteigende λ_{get}

(b) Umschaltung von Push nach Pull für absteigendes λ_{get}

Abbildung 5.5: Anpassung des Push/Pull-Interaktionsschemas für unterschiedliche Anfrageraten.

geleitet noch beantwortet, da die Geräte im Push-Modus keine Anfragen mehr abonnieren. Ebenso werden Datenaktualisierungen auch von Geräten veröffentlicht, auch wenn alle dynamischen Mengen diese über Pull-Requests abrufen.

Die anderen Kurven zeigen die Ergebnisse der Anpassungsheuristiken (vgl. Abschnitt 4.4). Alle Heuristiken wechseln schließlich von der pull-basierten Interaktion, die zu Beginn des Experiments günstig ist, zum Push-Modus, der am Ende für höhere Anfrageraten von Vorteil ist. Die gerätebasierte Heuristik h_{dev} und die hybride Heuristik h_{hyb} wechseln kurz nachdem dies von Vorteil ist, in den Push-Modus, wohingegen die verbindungsbasierte Heuristik h_{lnk} länger braucht, um die Konfiguration anzupassen. Letzteres liegt daran, dass h_{lnk} die Push-Kosten überschätzt. Da die Interaktionen für jedes Paar einer dynamischen Menge und eines Geräts separat ausgewertet werden, kann die h_{lnk}-Heuristik die Einsparungen nicht berücksichtigen, die durch Datenaktualisierungen erzielt werden, die von einem Gerät einmal veröffentlicht, aber an mehrere abonnierte dynamische Mengen übermittelt werden. Im Gegensatz dazu berücksichtigen die Heuristiken h_{dev} und h_{hyb} im Push-Modus immer die gesamte Menge der Empfänger. Allerdings erzielt h_{hyb} kein wesentlich anderes Ergebnis als h_{dev}. Dies liegt daran, dass h_{hyb} in diesem Setup nicht in der Lage ist, eine Gruppe von Geräten und/oder dynamischen Mengen mit einem wesentlich unterschiedlichen Aktualisierungs- bzw. Anforderungsverhalten zu identifizieren, für die gemischte Push/Pull-Interaktionen von Vorteil sind.

In dem zweiten Experiment wurde die Pull-Rate alle vier Minuten schrittweise verringert $\lambda_{get} \in \{0.6, 0.5, \ldots, 0.1\}$. Die Ergebnisse sind in Abb. 5.5b dargestellt. Die absteigende Treppenform der Pull-Kurve ist deutlich zu erkennen, während die Push-Kurve nahezu unbeeinflusst bleibt und nur leicht abnimmt. Alle drei

Anpassungsheuristiken starten im Push-Modus und wechseln dann in den Pull-Modus, sobald Pull-basierte Interaktionen deutlich günstiger werden. In der Tat wird ein Schwellenwert verwendet, bei dem die neue Konfiguration in Bezug auf die gespeicherten Meldungen besser sein muss, um Oszillationen entgegenzuwirken. Diesmal folgt die h_{lnk}-Heuristik beim Wechsel zu Pull eng den h_{dev}- und h_{hyb}-Heuristiken. Dies ist darauf zurückzuführen, dass h_{lnk} keine Schätzung der Push-Kosten vornimmt, da diese Kosten im Push-Knoten exakt in der Nachrichtenstatistik widergespiegelt werden. Anstelle der Schätzung der Push-Kosten erfolgt eine Schätzung der Pull-Kosten, um zu bestimmen, wann Pull von Vorteil ist. Da die Pull-Kosten von den individuellen Antworten dominiert werden, die jedes Gerät in einer dynamischen Menge auf den Empfang einer entsprechenden Anfrage sendet, ist diese Schätzung wesentlich genauer. Dies ist auch der Grund, warum alle Heuristiken nahezu gleichzeitig auf Pull umschalten.

5.9 Verwandte Arbeiten

Remote Procedure Calls (RPCs) sind ein grundlegender Baustein in vielen verteilten Systemen, da sie es Entwicklern ermöglichen, ein einfaches, bekanntes und gut verstandenes Programmiermodell zu nutzen, um fortgeschrittene vernetzte Anwendungen zu erstellen [17]. RPCs sind weit verbreitet und unterstützen sowohl unerfahrene als auch professionelle Entwickler, indem sie beispielsweise dabei helfen, Webanwendungen (z.B. Wetter-, Verkehrs- oder Luftqualitätsberichte) in eigene Programmierprojekte zu integrieren [22] oder indem sie Cloud-Anwendungen und -Dienste in einem modernen Rechenzentrum verbinden [83]. Daher ist die Integration von Anfrage-/Antwort-Interaktionen in PubSub- und Messaging-Protokolle kein neues Konzept [70, 32]. Dennoch ist diese Integration in realen Anwendungsdomänen oft erforderlich, um etablierte Standards und Protokolle zusammenarbeiten zu lassen, beispielsweise um OPC UA und DDS in industrielle Netzwerke zu integrieren [120, 42, 154] oder um REST und MQTT in IoT [29] oder Robotik [15] zu kombinieren. Im Gegensatz zu diesen sehr spezifischen Protokollzuordnungen konzentrieret sich die Arbeit auf die wesentlichen Interaktionen dynamischer Mengen und zeigen, wie diese auf verschiedene Publish/Subscribe-Protokolle abgebildet werden können. Neben herkömmlichen RPCs berücksichtigen wir auch viele-zu-viele-Aufrufe [82].

Die Interaktion mit mehreren Entitäten, die eine natürliche Gruppe bilden (z.B. um Daten zu speichern/abzurufen oder ihren Zustand zu manipulieren), ist in vielen verteilten Systemen eine gängige Aufgabe. Daher bieten Middleware-Plattformen häufig spezifische Abstraktionen zum Gruppieren dieser Entitäten und zur Vereinfachung ihrer Verwaltung, beispielsweise zum Sammeln, Adressieren und Verwalten von Objektsammlungen in verteilten Objektsystemen [44] oder in mobilen Ad-hoc-Netzwerken (MANETs) [123]. Ebenso ermöglichen moderne Smart-Home-Plattformen im IoT den Benutzern, Geräte und Zubehör in Räumen und Zonen [5] zu gruppieren und diese mithilfe von Szenen [157] zu manipulieren, aber dennoch überlassen diese Plattformen den Entwicklern mühsame

und umständliche Implementierungsdetails, z.B. indem sie jedes Gerät einzeln durchlaufen. Dynamischen Mengen bieten einen viel bequemeren Ansatz, u.a. in der Laufzeitflexibilität. So ist es beispielsweise möglich, das Interaktionsschema zwischen Anwendungen und Geräten dynamisch anzupassen, um die Netzwerkbandbreite zu minimieren und die Anwendungsleistung zu optimieren.

Die Analyse des Kompromisses zwischen Push-basierter und Pull-basierter Kommunikation ist eine bekannte Forschungsfrage in der Datenverteilung [2], die in erster Linie von den konkreten Raten abhängt, mit denen Daten produziert bzw. konsumiert werden. Da diese Raten sehr anwendungsspezifisch sind und im Laufe der Zeit sogar erheblich schwanken können, ist eine individuelle, auf den Problembereich zugeschnittene Lösung erforderlich. Beispielsweise vergleicht [19] verschiedene Push/Pull-Techniken, um Webanwendungen zu realisieren, die Echtzeitdaten anzeigen, und [14] analysiert Push/Pull-Strategien, um die verteilte Graphenverarbeitung im Hochleistungsrechnen zu beschleunigen, während [132] Push/Pull-Abfragepläne mit Prädikaten ableitet, um die Übertragungskosten bei der verteilten komplexen Ereignisverarbeitung zu senken.

5.10 Diskussion

In diesem Kapitel wurde die Abbildung von Interaktionsmustern auf Publish/Subscribe vorgestellt. Die nachrichtenbasierte Kommunikation erlaubt eine mehrdimensionale Entkopplung zwischen Sender und Empfänger, wodurch Skalierung und Wartung verteilter Systeme gefördert werden. Die Auswahl zum gegenseitigen Auffinden von Kommunikationspartnern, der Datenaustausch über das Pull- und Push-Paradigma sowie Kontrollanweisungen zur adaptiven Umschaltung wurden als Anforderungen definiert, welche die konzeptuellen Arbeiten aus den vorherigen Kapiteln auf Protokolle abbilden. Für die Adressierung wurde themenbasiertes Routing gewählt und eine Struktur definiert, die eine weitreichende Unterstützung verschiedener Protokolle und Dienste berücksichtigt. Die Interaktionsmuster wurden auf Basis von Publish/Subscribe-Sequenzen abgebildet und eine Vielzahl von Nachrichtentypen definiert, um den erforderlichen Nachrichtenfluss mit Einzel- und Gruppenadressierungen zu gewährleisten. Im Folgenden wurde eine Architektur präsentiert, welche die Integration neuer Nachrichtenprotokolle und deren individueller Aufrufe in die Programmierabstraktion ermöglicht. Um eine einheitliche Interpretation der Nutzlast einer Nachricht in unterschiedlichen Umgebungen zu gewährleisten, wurde die Notwendigkeit der Serialisierungsfähigkeit dargelegt. Die Funktionalität wurde prototypisch mit den Protokollen MQTT und AMQP sowie den Serialisierungen Protobuf und GSON umgesetzt. Im Rahmen der Evaluierung wurde aufgezeigt, dass die Umschaltung von Pull- auf Push-Kommunikation bzw. umgekehrt auf Basis der im Vorgängerkapitel 4 vorgestellten Heuristiken auch im realen System die gewünschten Nachrichteneinsparung bringt. Dies wurde anhand einer Steigerung und Senkung der Anfrageraten demonstriert.

Kapitel 6

Zusammenfassung und Ausblick

Inhalt

6.1 Diskussion 162
6.2 Ausblick 165

Die Verfügbarkeit vernetzter Alltagsgegenstände sowie spezialisierter Sensoren und Aktuatoren führt zu einer Durchdringung von IT-Systemen in den privaten, geschäftlichen und öffentlichen Raum. Die Erfassung von Zuständen der Umgebung oder physischer Geräte erfolgt auf Grundlage von Sensordaten, welche als Grundlage für die Verarbeitung in unterschiedlichsten Anwendungen dienen. Diese allgegenwärtigen Anwendungen in Verbindung mit dem Teilgebiet Internet der Dinge ermöglichen Mehrwert in Form von abgeleiteten Informationen und Assistenz für Nutzer sowie Entscheidungen und Optimierungen für Prozesse im industriellen Bereich. Ubiquitäre Systeme sind durch eine Vielzahl unterschiedlicher Geräte gekennzeichnet, die ein dynamisches kooperierendes Ensemble bilden. Die hohe Dynamik und Heterogenität der Systeme stellt die Anwendungsentwickler vor Herausforderungen, die mit den aktuell verfügbaren Möglichkeiten kaum beherrschbar sind. Der Umfang sowie die Anzahl der im verteilten System vorhandenen Objekte sind während der Entwicklung häufig unbekannt. In der Phase der Umsetzung ist die konkrete Ausgestaltung der Umgebung zur Laufzeit nicht prognostizierbar. Die detaillierte Ausprogrammierung der Bindungsroutinen kaum realisierbar und limitiert die Entscheidungsfreiheit einer Laufzeitumgebung hinsichtlich möglicher Anpassungen, um das Ziel mit alternativen Strategien zu erreichen. Des Weiteren ist zu berücksichtigen, dass Anwendungsentwickler entsprechend ihrer Ziele eine bestimmte Sichtweise bezüglich der Art von entfernten Objekten haben, die für die Lösung ihrer Herausforderungen dienlich sind. In verteilten Umgebungen interagieren auch andere Anwendungen mit vorhandenen Ressourcen, sodass eine Vielzahl weiterer Interaktionen besteht, die einem Anwendungsentwickler nicht bekannt sein können. Um die Netzlast zu reduzieren, ist eine adaptive Anwendung von Kommunikationsmustern erforderlich. Die Protokollvielfalt mit verschiedenen Interaktionsparadigmen und Ausführungssemantiken führt zu einer Faktorisierung des Entwicklungsaufwands gegenüber ubiquitären Systemen. Jedoch ist die Anschlussfähigkeit zur Einbeziehung verfügbarer Gerätekandidaten sowie dem technischen Fortschritt in der Entwicklung von Transportprotokollen notwendig.

6.1 Diskussion

Die vorliegende Forschungsarbeit befasst sich mit der Konzeption von Programmierabstraktionen, welche die Bewältigung diverser Herausforderungen im Kontext der Programmierung verteilter Systeme zum Ziel haben. Die Unterstützung von Anwendungsentwicklern zielt darauf ab, die Programmierung gegen eine bekannte Anzahl beabsichtigter Geräte zu ermöglichen, welche während der Laufzeit in einem dynamischen Ensemble bereitgestellt werden können. Die genannten Abstraktionen greifen in verschiedenen Bereichen des Datenflusses, beginnend bei der Anwendungsintegration, über die Verwaltung von Mengenmitgliedern und Methodenaufrufen, bis hin zur adaptiven Anwendung von Kommunikationsmustern durch Heuristiken sowie der Abbildung auf ein universelles Adressierungsschema für Publish/Subscribe samt Nachrichtenstrukturen.

Die für das Verständnis der Hauptbeiträge dieser Arbeit notwendigen theoretischen Hintergrundinformationen werden in Kapitel 2 bereitgestellt. Beginnend werden die Charakteristika sowie Herausforderungen bei der Nutzung ubiquitärer und verteilter Systeme erörtert. Im Anschluss erfolgt eine Vorstellung diverser Kommunikationsarten, gefolgt von einer Skizzierung der Akteure und Komponenten des Publish/Subscribe-Musters, auf dessen Basis eine nachfolgende Protokollabbildung vorgenommen wird. Des Weiteren werden die verschiedenen Eigenschaften der Selbstorganisation von IT-Systemen erörtert, welche auch bei den Programmierabstraktionen Berücksichtigung finden, um anpassungsfähige und robuste Ausführungen anzustreben. Im Folgenden werden ausgewählte Elemente der Softwareentwicklung skizziert, die für die Gestaltung und Umsetzung von Programmierabstraktionen Anwendung finden. Dabei wird zunächst auf Konfigurationsmöglichkeiten in unterschiedlichen Phasen der Softwareentwicklung eingegangen. Anschließend werden Reflexionsfähigkeiten sowie Funktionalitäten von Laufzeitumgebungen und Middleware erörtert, welche zum Stellvertreter-Entwurfsmuster überleiten, das für die Programmierabstraktion der dynamischen Mengen von Relevanz ist.

Ein Hauptbeitrag dieser Arbeit besteht in der Entwicklung einer Programmierabstraktion für dynamische Mengen [124, 126, 129], welche in Kapitel 3 vorgestellt wird. Diese gestattet es eine unbekannte Anzahl von Instanzen hinter der Schnittstelle einer Instanz zu bündeln. Im Rahmen der Integration in eine Anwendung besteht zudem die Möglichkeit, die Transparenz aufzuheben. Dies setzt voraus, dass sich Anwendungsentwickler der Tatsache bewusst sind, dass sie gegen eine Menge programmieren und diese feingranularer steuern wollen. Die Angabe von Metadaten ist eine notwendige Voraussetzung für die Verwendung der Schnittstelle als dynamische Menge. So kann die den Kontrollfluss steuernde Laufzeitumgebung einen dynamischen Stellvertreter injizieren, der anschließend Methodenaufrufe an der Schnittstelle abfängt und verarbeitet. Die Anwendung von Metadaten für verschiedene Wirkungsgrade zu unterschiedlichen Phasen der Softwareentwicklung wurde dargelegt, welche kaskadierend in eine anwendbare Konfiguration münden. Der Stellvertreter übernimmt die Selektion und die Verwaltung der Mitglieder einer dynamischen Menge, repliziert Methodenaufrufe an Aufrufkandidaten entsprechend der Ausführungsstrategie und vermittelt den Aufruf zur Übertragung an Stellvertreter entfernter Geräte. Der begleitende Aufrufkontext setzt sich aus den übergebenen Richtlinien zusammen, passt das Aufrufverhalten der aktuellen Situation an und korreliert eingehende Antworten. Mehrere Rückgabewerte werden auf einen durch Anwendung bereitgestellter Funktionen aggregiert um diesen entsprechend der Methodensignatur an die Anwendung zurückzugeben. Eine Fallstudie mit der Programmiersprache Java demonstriert die Funktionsweise der Programmierabstraktion mit dynamischem Stellvertreter, der von der Aufrufreplikation bis zur Aggregation Komplexität verbirgt. Zudem wird die Anwendung von Metadaten über Annotationen sowie deren Verarbeitung durch die Laufzeitumgebung aufgezeigt.

Die erzeugte Transparenz suggeriert eine lokale Instanz, während die Replikation und Aufrufe über ein Netzwerk verborgen bleiben. Bei der Repräsentation

in der Anwendung besteht die Möglichkeit, dass Methoden mit hoher Frequenz vergleichbar einer lokalen aufgerufen werden. Die Umsetzung einer adaptiven Kommunikation wird in Kapitel 4 vorgestellt, um die Transportkosten über das Netzwerk zu reduzieren und unnötige Kommunikation zu vermeiden [130, 127]. Die präsentierten Interaktionsmuster mit Einzel- oder Gruppenadressierungen (Unicast/Multicast) berücksichtigen anfrage- und aktualisierungsbasierte Kommunikation (Pull/Push). Da eine zeitnahe Berechnung der realen Netzwerkkosten mit einem hohen Aufwand verbunden wäre, werden in diesem Kontext drei Heuristiken präsentiert, welche die Netzwerkkosten approximieren. Dabei werden unterschiedliche Wirkungsweisen auf Basis von Interaktionen und Kostenfunktionen berücksichtigt. Die Statistikführung, Evaluierung und Umschaltung werden für Anwendungsentwickler abstrahiert und zur Laufzeit periodisch ausgeführt. Eine diskrete Ereignissimulation dient der Validierung des Wirkungspotenzials der Heuristiken unter Anwendung lokalen Wissens sowie variabler Konfigurationsparameter.

Im UbiComp bzw. IoT-Bereich haben sich nachrichtenbasierte Protokolle etabliert, die Entkopplungen ermöglichen und Systeme skalierbar und wartbar halten können. Die in den vorherigen Kapiteln erläuterten Konzepte der Gruppenkommunikation und adaptiven Interaktion werden in Kapitel 5 zusammengeführt und auf das Publish/Subscribe-Muster abgebildet [128]. Die Gestaltung eines universelles Adressierungsschema respektiert Einschränkungen einzelner Protokolle und Dienste. Für die Implementierung von Bindungsroutinen, dem Datenaustausch in Form von Push- und Pull-Mechanismen sowie für Kontroll- und Verwaltungsinstruktionen der Adaptivitätssteuerung wurden spezifische Nachrichtentypen entwickelt, die auf die jeweiligen Fähigkeiten der verwendeten Transportprotokolle abgebildet werden. In der prototypischen Umsetzung werden AMQP und MQTT als Fallbeispiele herangezogen. Für die Nutzlast wurde die Serialisierung der Daten motiviert und das Verwenden unterschiedlicher Umsetzungen konzipiert. Protobuf und GSON werden verwendet, um die plattformunabhängige Übertragung und Interpretation der Daten zu demonstrieren. Die Funktionsfähigkeit sowie die erfolgreiche Umschaltung durch die Adaptivitätssteuerung mit den vorgestellten Heuristiken werden durch Experimente abschließend validiert.

Die drei vorgestellten Beiträge liefern Abstraktionen in verschiedenen Teilbereichen für die Interaktion in verteilten Systemen. Die Intention, die mit der Umsetzung verfolgt wurde, besteht darin, Anwendungsentwickler zu entlasten und Systeme auch zur Laufzeit konfigurierbar zu halten. Die Verlagerung von Funktionalität aus dem Anwendungskontext in die Laufzeitumgebung ermöglicht die Vornahme von Entscheidungen im Sinne des Geräteensembles.

Die Wahlfreiheit wird bei der Gestaltung der Abstraktionen berücksichtigt. Durch die Verwendung von Entwurfsmustern wie Dependency Injection wird die Abhängigkeit der Anwendung zur Laufzeitumgebung aufgelöst, so dass die Anwendung mobil gegenüber anderen die Abstraktion implementierende Umgebungen bleibt. Die Integration von zusätzlichen Transportprotokollen ist möglich, da

deren Aufrufverhalten und Mapping der Konfigurationen durch übergeordnete Methodenaufrufen verborgen wird.

Die Programmierabstraktion der dynamischen Menge verkleinert Wissenslücke zwischen Entwurfs- und Laufzeit sowie den Abstand zwischen Makro und Mikroebene. Die Auswahl der Geräte einer Menge erfolgt erst bei Bereitstellung oder sogar erst zur Laufzeit. Die Verwaltung der Menge erfolgt durch die Middleware und nicht durch die Anwendungsentwickler. Dies bietet zudem Raum und Flexibilität für Selbstorganisation und -optimierung. Hinter der Mengenschnittstelle kann zwischen Pull und Push adaptiert sowie Gruppen- und Einzelkommunikation einbezogen werden. Die Abbildung auf existierende Publish/Subscribe-Protokolle beweist die Kompatibilität zu aktuellen Techniken.

6.2 Ausblick

Die Arbeiten zur Abstraktion und Gruppierung von Objekten in verteilten Systemen und deren Abbildung auf Publish/Subscribe liefern eine Vielzahl neuer offener Fragestellungen.

Die Integration dynamischer Mengen in eine ereignisbasierte Publish/Subscribe-Middleware wie Rebeca [116] kann den Reifegrad der Software erhöhen und in einer entwickelten Laufzeitumgebung eine schnellere Reaktion auf Änderungen der Mitglieder ermöglichen. Dies soll durch Leistungsmessungen und Benchmarks evaluiert werden.

Die Erweiterung des Funktionsumfangs dynamischer Mengen stellt eine vielversprechende Herausforderung dar, die sich durch Transaktionen in mehrfacher Hinsicht aufzeigt. Zum einen ist eine niedrigschwellige Integration in die verwendete Programmiersprache wünschenswert, zum anderen ist der transaktionale Entwurf von entfernten Methodenaufrufen über nachrichtenbasierte Protokolle herausfordernd. Bei Methodenaufrufen, die den Zustand von entfernten Objekten verändern, ist zudem die Modellierung und Anwendung von Rücknahmemethoden von großem Interesse, um ein Zurücksetzen auf den vorherigen Zustand zu realisieren und auch zu garantieren. Die Berücksichtigung von Sicherheitsmechanismen, beispielsweise der exklusive Zugriff auf eine Ressource, sowie die damit verbundene Behandlung von Statusmeldungen, beispielsweise wenn ein Gerät bereits durch eine andere Anwendung gebunden ist, wirft die Frage auf, inwiefern die Abstraktionen in diesem Kontext unterstützen können.

Dynamische Mengen werden derzeit im Anwendungskontext gehalten. In einer Middleware können mehrere Anwendungen ihre Laufzeit haben. Eine Abstraktion in Richtung Netzwerk bringt die Idee virtueller Mengen mit eigener Adressierung zu erstellen, die Methodenaufrufe kombinieren, wenn sich deren Selektionskriterien abdecken oder überlappen bzw. die Schnittstellenhierarchie dies gestattet und die Frequenzen der Methodenaufrufe ähnlich ist. Ziel ist hier ebenso die Netzlast zu minimieren.

Für die Adressierung wurde ein themenbasiertes Adressierungsschema verwendet, um eine möglichst umfassende Abdeckung zu gewährleisten. Die Anwendung auf inhaltsbasiertes Routing zielt darauf ab, die Praktikabilität für Publish/Subscribe-Systeme ohne Themen unter Hinzufügen zusätzlicher Attribute zur Nachricht zu erforschen.

Die Heuristiken agieren gegenwärtig auf den Clients eines Publish/Subscribe-Systems. Die Platzierung der Heuristiken auf Zwischenknoten in einer zu definierenden Topologie, die ein Umschalten zwischen Push- und Pull-Kommunikation im Inneren ermöglicht, eröffnet neue Möglichkeiten hinsichtlich der Flexibilität der Kommunikation und trägt vermutlich zu einer weiteren Reduktion der Kommunikationskosten bei. Ein Beispiel für die Verwendung der Push- und Pull-Kommunikation ist die Übertragung von Daten von einem Gerät an ein Zwischengerät mittels Push-Kommunikation, während die Daten am Zwischenknoten mittels Pull-Kommunikation angefragt und zwischengespeichert werden. Des Weiteren kann es für spezifische Anwendungsbereiche ausreichend sein, dass Datenaktualisierungen lediglich übertragen werden, sofern sich der jeweilige Wert um mindestens einen bestimmten Wert geändert hat. Hier sind Schwellwerte von einer Anwendung über die Middleware zu kommunizieren, die als zusätzliche Filter im Publish/Subscribe-System abgebildet werden, um Nachrichten weiterzuleiten oder zu verwerfen. Die prototypische Umsetzung demonstrierte die adaptive Umschaltung der Kommunikation auf der Ebene einer Methode oder eines Objekts. Auch für diese Dimension ist die Granularität in Kombination zu erforschen, unter welchen Umständen es ratsam ist, mehrere Attribute eines Objekts in einer Nachricht zu kombinieren und an der Senke beim Empfänger aufzuteilen. Eine weitere Möglichkeit stellt auch die Verlagerung in die Innendienste eines Publish/Subscribe-Systems dar, um auf diese Weise die Anzahl an Nachrichten zu reduzieren.

Im Rahmen der Experimente wurde der Nachrichtenfluss zur Abbildung der Interaktionsmuster und adaptiven Umschaltung in einem Setup mit einem zentralen Broker durchgeführt. Die Übertragung der Ergebnisse auf andere Netzwerktopologien, wie beispielsweise Stern- oder Ringtopologien, kann Aufschluss darüber geben, wie sich Änderungen der Topologie auf die Funktionsweise der Heuristiken auswirken.

Literatur

[1] K. Aberer, M. Hauswirth und A. Salehi. “A Middleware for Fast and Flexible Sensor Network Deployment”. In: *VLDB ’06: Proceedings of the 32nd international conference on Very large data bases*. Seoul, Korea: VLDB Endowment, 2006, S. 1199–1202. ISBN: 1-59593-385-9.

[2] S. Acharya, M. Franklin und S. Zdonik. “Balancing Push and Pull for Data Broadcast”. In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’97. Tucson, Arizona, USA: ACM, 1997, S. 183–194. ISBN: 0-89791-911-4. DOI: `10.1145/253260.253293`.

[3] Amazon Web Services, Inc., Hrsg. *Amazon Simple Notification Service*. URL: `https://aws.amazon.com/de/sns/` (besucht am 03. 07. 2024).

[4] *An architectural blueprint for autonomic computing*. IBM. White Paper, `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0e99837d9b1e70bb35d516e32ecfc345cd30e795`. 2006.

[5] Apple Inc., Hrsg. *HomeKit*. 2023. URL: `https://developer.apple.com/documentation/homekit` (besucht am 18. 07. 2024).

[6] A. Arasu, S. Babu und J. Widom. “The CQL continuous query language: semantic foundations and query execution”. In: 15. Juni 2006, 121–142. DOI: `10.1007/s00778-004-0147-z`.

[7] H. Balzert. “Der Software-Lebenszyklus”. In: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Heidelberg: Spektrum Akademischer Verlag, 2011, S. 1–4. ISBN: 978-3-8274-2246-0. DOI: `10.1007/978-3-8274-2246-0_1`.

[8] A. Banks, E. Briggs, K. Borgendale und R. Gupta. *MQTT Version 5.0, OASIS Standard*. `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html`. 7. März 2019.

[9] A. Banks und R. Gupta. *MQTT Version 3.1.1. OASIS Standard*. `https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`. 29. Okt. 2014.

[10] K. Beck. *Implementation Patterns : der Weg zu einfacherer und kostengünstigerer Programmierung*. ger. Hrsg. von K. Beck. Programmer’s choice. Addison-Wesley, 2008, S. 191. ISBN: 978-3-8273-2644-7.

[11] S. Behnel, L. Fiege und G. Muhl. “On Quality-of-Service and Publish-Subscribe”. In: *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*. 2006, S. 20–20. DOI: 10.1109/ICDCSW.2006.77.

[12] R. Berbner, O. Heckmann, A. Mauthe und R. Steinmetz. “Eine Dienstgüte unterstützende Webservice-Architektur für flexible Geschäftsprozesse”. In: *Wirtschaftsinformatik* 47 (Aug. 2005), S. 268–277. DOI: 10.1007/BF03254914.

[13] P. A. Bernstein. “Middleware: A Model for Distributed System Services”. In: *Commun. ACM* 39.2 (Feb. 1996), 86–98. ISSN: 0001-0782. DOI: 10.1145/230798.230809.

[14] M. Besta, M. Podstawski, L. Groner, E. Solomonik und T. Hoefler. “To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations”. In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. Washington, DC, USA: ACM, Juni 2017, S. 93–104. ISBN: 978-1-45034-699-3. DOI: 10.1145/3078597.3078616.

[15] P. Bhavsar, S. H. Patel und T. M. Sobh. “Hybrid Robot-as-a-Service (RaaS) Platform (Using MQTT and CoAP)”. In: *Proceedings of the International Conference on Internet of Things (iThings 2019)*. Atlanta, GA, USA: IEEE, Juli 2019, S. 974–979. DOI: 10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00171.

[16] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham und P. Shenoy. “Adaptive Push-Pull: Disseminating Dynamic Web Data”. In: *Transactions on Computers* 51.6 (Juni 2002), S. 652–668. ISSN: 0018-9340. DOI: 10.1109/TC.2002.1009150.

[17] A. D. Birrell und B. J. Nelson. “Implementing Remote Procedure Calls”. In: *ACM Trans. Comput. Syst.* 2.1 (Feb. 1984), S. 39–59. ISSN: 0734-2071. DOI: 10.1145/2080.357392.

[18] J. Bonér, D. Farley, R. Kuhn und M. Thompson. *The Reactive Manifesto*. 16. Sep. 2024. URL: https://www.reactivemanifesto.org/ (besucht am 24. 07. 2024).

[19] E. Bozdag, A. Mesbah und A. van Deursen. “A Comparison of Push and Pull Techniques for AJAX”. In: *2007 9th IEEE International Workshop on Web Site Evolution*. Paris, France, Okt. 2007, S. 15–22. DOI: 10.1109/WSE.2007.4380239.

[20] T. Bray, Hrsg. *The JavaScript Object Notation (JSON) Data Interchange Format*. Internet Engineering Task Force (IETF), Dez. 2017. DOI: 10.17487/RFC8259.

[21] Broadcom Inc. *RabbitMQ: One broker to queue them all | RabbitMQ*. URL: https://www.rabbitmq.com/ (besucht am 24. 04. 2024).

[22] B. Broll, A. Lédeczi, P. Volgyesi, J. Sallai, M. Maroti, A. Carrillo, S. L. Weeden-Wright, C. Vanags, J. D. Swartz und M. Lu. “A Visual Programming Environment for Learning Distributed Programming”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. Seattle, WA, USA: ACM, 2017, S. 81–86. ISBN: 978-1-45034-698-6. DOI: 10.1145/3017680.3017741.

[23] B. Bruegge und A. A. Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. USA: Prentice Hall PTR, 1999. ISBN: 978-0-13-489725-7.

[24] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè und M. Shaw. “Engineering Self-Adaptive Systems through Feedback Loops”. In: *Software Engineering for Self-Adaptive Systems*. Hrsg. von B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi und J. Magee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 48–70. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_3.

[25] J. Bugeja, A. Jacobsson und P. Davidsson. “Smart Connected Homes”. In: *Internet of Things A to Z*. John Wiley & Sons, Ltd, 2018. Kap. 13, S. 359–384. ISBN: 978-1-11945-673-5. DOI: 10.1002/9781119456735.ch13.

[26] *Build a publish-subscribe WebSocket server | Bun Examples*. URL: https://bun.sh/guides/websocket/pubsub (besucht am 31. 07. 2024).

[27] C. Cetina, P. Giner, J. Fons und V. Pelechano. “Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes”. In: *Computer* 42.10 (2009), S. 37–43. DOI: 10.1109/MC.2009.309.

[28] Cloud Native Computing Foundation. *NATS – Connective Technology for Adaptive Edge & Distributed Systems*. 2024. URL: https://nats.io/ (besucht am 15. 07. 2024).

[29] M. Collina, G. E. Corazza und A. Vanelli-Coralli. “Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST”. In: *Proceedings of the IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2012)*. Sydney, NSW, Australia: IEEE, Sep. 2012, S. 36–41. DOI: 10.1109/PIMRC.2012.6362813.

[30] *CORBA to WSDL/SOAP Interworking, Version 1.2.1*. https://www.omg.org/spec/C2WSDL/1.2.1/About-C2WSDL. 2008.

[31] G. F. Coulouris, J. Dollimore und T. Kindberg. *Distributed Systems : Concepts and Design*. 5. Aufl. Addison-Wesley, 2012. ISBN: 978-0-13-214301-1.

[32] G. Cugola, M. Migliavacca und A. Monguzzi. “On Adding Replies to Publish-Subscribe”. In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*. DEBS ’07. Toronto, Ontario, Canada: ACM, 2007, S. 128–138. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266918.

[33] *DCE 1.1 Remote Procedure Call*. https://pubs.opengroup.org/onlinepubs/009629399/toc.pdf. The Open Group, Okt. 1997.

[34] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham und P. Shenoy. "Adaptive Push-pull: Disseminating Dynamic Web Data". In: *Proceedings of the 10th International Conference on World Wide Web*. WWW '01. Hong Kong, Hong Kong: ACM, 2001, S. 265–274. ISBN: 1-58113-348-0. DOI: 10.1145/371920.372066.

[35] G. Di Marzo Serugendo, N. Foukia, S. Hassas, A. Karageorgos, S. K. Mostéfaoui, O. F. Rana, M. Ulieru, P. Valckenaers und C. Van Aart. "Self-organisation: Paradigms and applications". In: *International Workshop on Engineering Self-Organising Applications*. Springer. 2003, S. 1–19. ISBN: 978-3-540-24701-2. DOI: 10.1007/978-3-540-24701-2_1.

[36] F. Dressler. *Self-Organization in Sensor and Actor Networks*. John Wiley & Sons, Nov. 2007. ISBN: 9780470724460. DOI: 10.1002/9780470724460.

[37] J. Dunkel, Hrsg. *Systemarchitekturen für Verteilte Anwendungen : Client-Server, Multi-Tier, SOA, Event-Driven Architectures, P2P, Grid, Web 2.0*. ger. Hanser eLibrary. In: ciando-Library. München: Hanser, 2008. ISBN: 978-3-44641-745-8. DOI: 10.3139/9783446417458.

[38] *E.800 : Definitions of terms related to quality of service*. https://www.itu.int/rec/T-REC-E.800-200809-I. Sep. 2008.

[39] Eclipse Foundation AISBL. *Jakarta Messaging™*. Jan. 2018. URL: https://projects.eclipse.org/projects/ee4j.messaging (besucht am 03. 05. 2024).

[40] Eclipse Foundation AISBL. *Eclipse Cyclone DDS™*. 2024. URL: https://projects.eclipse.org/projects/iot.cyclonedds (besucht am 10. 08. 2024).

[41] M. Elliott und K. Kraemer. *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*. ASIST monograph series. Information Today, 2008. ISBN: 978-1-57387-311-6.

[42] R. Endeley, T. Fleming, N. Jin, G. Fehringer und S. Cammish. "A Smart Gateway Enabling OPC UA and DDS Interoperability". In: *Proceedings of the IEEE SmartWorld Congress (SmartWorld 2019)*. Leicester, UK: IEEE, Aug. 2019, S. 88–93. DOI: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00058.

[43] P. Eugster. "Uniform Proxies for Java". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, 139–152. ISBN: 1595933484. DOI: 10.1145/1167473.1167485.

[44] P. Eugster, R. Guerraoui und J. Sventek. “Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction”. In: *14th European Conference on Object-Oriented Programming*. Hrsg. von E. Bertino. LNCS 1850. Sophia Antipolis und Cannes, France: Springer, Juni 2000, S. 252 –276. ISBN: 978-3-540-45102-0. DOI: 10.1007/3-540-45102-1_13.

[45] P. T. Eugster, P. A. Felber, R. Guerraoui und A.-M. Kermarrec. “The Many Faces of Publish/Subscribe”. In: *ACM Comput. Surv.* 35.2 (2003), 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078.

[46] P. T. Eugster, R. Guerraoui und J. Sventek. *Type-Based Publish/Subscribe*. http://infoscience.epfl.ch/record/52358. 2000.

[47] P. Felber. “The CORBA object group service : a service approach to object groups in CORBA”. en. Diss. Lausanne: EPFL, 1998. DOI: 10.5075/epfl-thesis-1867.

[48] P. Felber, R. Guerraoui und A. Schiper. “Replication of CORBA Objects”. In: *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*. Springer, 1999, S. 254–276. ISBN: 3-540-67196-X.

[49] I. Fette und A. Melnikov. *The WebSocket Protocol (RFC 6455)*. Internet Engineering Task Force (IETF), Dez. 2011. DOI: 10.17487/RFC6455.

[50] I. R. Forman. *Java reflection in action*. Manning Publications Co., 2005. ISBN: 1-932394-18-4.

[51] P. S. Foundation. *FastAPI Websocket Pub/Sub (fastapi-websocket-pubsub 0.3.9)*. 2. Juli 2024. URL: https://pypi.org/project/fastapi-websocket-pubsub/ (besucht am 31. 07. 2024).

[52] M. Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: https://martinfowler.com/articles/injection.html (besucht am 23. 06. 2024).

[53] *Framework Vs. Library: Concept, Example and Differences*. 2. Juni 2023. URL: https://www.shiksha.com/online-courses/articles/framework-vs-library/ (besucht am 23. 06. 2024).

[54] M. Franklin und S. Zdonik. “"Data in Your Face: Push Technology in Perspective”. In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’98. Seattle, Washington, USA: ACM, 1998, S. 516–519. ISBN: 0-89791-995-5. DOI: 10.1145/276304.276360.

[55] E. Freeman, S. Hupfer und K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional, Juni 1999. ISBN: 978-0-201-30955-3.

[56] M. Friedewald und O. Raabe. “Ubiquitous computing: An overview of technology impacts”. In: *Telematics and Informatics* 28.2 (2011), S. 55–65. ISSN: 0736-5853. DOI: 10.1016/j.tele.2010.09.001.

[57] E. Gamma, R. Helm, R. Johnson, J. Vlissides und D. Patterns. *Design Patterns: Elements of reusable object-oriented software*. 2. Aufl. Addison-Wesley Professional Computing Series, 1995. ISBN: 978-0-201-48537-0.

[58] E. Gamma, R. Helm, R. Johnson und J. M. Vlissides. *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*. Hrsg. von D. Riehle. Professionelle Softwareentwicklung. Addison-Wesley, 2004, S. 479. ISBN: 978-3-8273-2199-2.

[59] H. Garcia-Molina und A. Spauster. "Ordered and reliable multicast communication". In: *ACM Transactions on Computer Systems (TOCS)* 9.3 (1991), S. 242–271. ISSN: 0734-2071. DOI: 10.1145/128738.128741.

[60] D. Gelernter. "Multiple tuple spaces in Linda". In: *PARLE '89 Parallel Architectures and Languages Europe*. Berlin, Heidelberg: Springer, 1989, S. 20–27. ISBN: 978-3-540-46184-5.

[61] R. Godfrey, D. Ingham und R. Schloming, Hrsg. *OASIS Advanced Message Queuing Protocol (AMQP), Version 1.0, Part 3: Messaging*. 29. Okt. 2012. URL: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-messaging-v1.0-os.html (besucht am 27. 03. 2023).

[62] Google LLC, Hrsg. *Gson (GitHub)*. 2023. URL: https://github.com/google/gson/ (besucht am 06. 04. 2023).

[63] Google LLC, Hrsg. *Protocol Buffers*. 2023. URL: https://protobuf.dev (besucht am 29. 04. 2023).

[64] Google LLC. *Pub/Sub für Anwendungs- und Datenintegration | Google Cloud*. 2024. URL: https://cloud.google.com/pubsub (besucht am 28. 07. 2024).

[65] D. Gouscos, M. Kalikakis und P. Georgiadis. "An approach to modeling Web service QoS and provision price". In: *Fourth International Conference on Web Information Systems Engineering Workshops, 2003. Proceedings.* 2003, S. 121–130. DOI: 10.1109/WISEW.2003.1286794.

[66] M. D. Groves und B. Wagner. *Definieren und Lesen von benutzerdefinierten Attributen*. 22. März 2023. URL: https://docs.microsoft.com/de-de/dotnet/csharp/tutorials/attributes (besucht am 06. 04. 2024).

[67] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar und Y. Lafon, Hrsg. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. 27. Apr. 2007. URL: https://www.w3.org/TR/soap12-part1/ (besucht am 28. 07. 2024).

[68] U. Hansmann, L. Merk, M. S. Nicklous und T. Stober. *Pervasive computing handbook*. eng. Includes bibliographical references (S. 387 - 389) and index. Berlin: Springer, 2001. ISBN: 3540671226 | 3-540-67122-6. DOI: 10.1007/978-3-662-04318-9.

[69] J. L. Hennessy und D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 5. Aufl. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 978-0-12-383872-8.

[70] J. C. Hill, J. C. Knight, A. M. Crickenberger und R. Honhart. *Publish and Subscribe with Reply*. https://apps.dtic.mil/sti/citations/ADA479195. Okt. 2002.

[71] H. Huang und L. Wang. "P&P: A Combined Push-Pull Model for Resource Monitoring in Cloud Computing Environment". In: *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*. Miami, FL, USA: IEEE, 2010, S. 260–267. DOI: 10.1109/CLOUD.2010.85.

[72] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), S. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[73] H.-A. Jacobsen, A. Cheung, G. Li, B. Maniymaran, V. Muthusamy und R. S. Kazemzadeh. "The PADRES publish/subscribe system". In: *Principles and Applications of Distributed Event-Based Systems*. IGI Global, 2010, S. 164–205. ISBN: 978-1-60566-697-6. DOI: 10.4018/978-1-60566-697-6.ch008.

[74] M. A. Jaeger, H. Parzyjegla, G. Mühl und K. Herrmann. "Self-organizing broker topologies for publish/subscribe systems". In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC '07. Seoul, Korea: Association for Computing Machinery, 2007, 543–550. ISBN: 978-1-59593-480-2. DOI: 10.1145/1244002.1244128.

[75] R. E. Johnson und B. Foote. "Designing reusable classes". In: *Journal of object-oriented programming* 1.2 (1988). http://www.laputan.org/drc.html, S. 22–35.

[76] JSR-299 Expert Group. *JSR-299: Contexts and Dependency Injection for the Java® EE platform*. https://jcp.org/en/jsr/detail?id=299. 10. Dez. 2009.

[77] JSR-330 Expert Group. *JSR-330: Dependency Injection for Java*. https://jcp.org/en/jsr/detail?id=330. 14. Okt. 2009.

[78] JSR-343 Expert Group. *JSR-343 Java™ Message Service 2.0 – Final Release*. https://jcp.org/en/jsr/detail?id=343. 21. März 2013.

[79] JSR-369 Expert Group. *JSR-369 Java™Servlet 4.0 Specification*. https://jcp.org/en/jsr/detail?id=369. 5. Sep. 2017.

[80] F. Junqueira und B. Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, Inc., 2013, S. 238. ISBN: 978-1-4493-6130-3.

[81] S. Kalepu, S. Krishnaswamy und S. Loke. "Verity: a QoS metric for selecting Web services and providers". In: *Fourth International Conference on Web Information Systems Engineering Workshops, 2003. Proceedings.* 2003, S. 131–139. DOI: 10.1109/WISEW.2003.1286795.

[82] A. Kaminsky und H.-P. Bischof. “Many-to-Many Invocation: A New Object Oriented Paradigm for Ad Hoc Collaborative Systems”. In: *Companion of the 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’02. Seattle, WA, USA: ACM, 2002, S. 72–73. ISBN: 1581136269. DOI: 10.1145/985072.985109.

[83] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei und D. Brooks. “Profiling a Warehouse-Scale Computer”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA ’15. Portland, OR, USA: ACM, 2015, S. 158–169. ISBN: 9781450334020. DOI: 10.1145/2749469.2750392.

[84] J. Kephart und D. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (2003), S. 41–50. DOI: 10.1109/MC.2003.1160055.

[85] S. Keßler. “Softwarelebenszyklus und Softwareevolution”. In: *Anpassung von Open-Source-Software in Anwenderunternehmen*. Wiesbaden: Springer Fachmedien Wiesbaden, 2013, S. 43–61. ISBN: 978-3-658-01955-6. DOI: 10.1007/978-3-658-01955-6_3.

[86] J. Krumm, Hrsg. *Ubiquitous Computing Fundamentals*. CRC Press, 2010. ISBN: 978-1-4200-9360-5.

[87] R. Kuhn, B. Hanafee und J. Allen. *Reactive Design Patterns*. Manning Publications Company, 2017. ISBN: 978-1-61729-180-7.

[88] B. N. Levine und J. J. Garcia-Luna-Aceves. “A comparison of reliable multicast protocols”. In: *Multimedia systems* 6.5 (1998), S. 334–348. DOI: 10.1007/s005300050097.

[89] B. Linnert. *Indirect Communication - II [Foliensatz: Netzprogrammierung (Algorithmen und Programmierung V)]*. 2015. URL: https://www.mi.fu-berlin.de/w/SE/VorlesungALPVNetzprogrammierung2015 (besucht am 15. 08. 2024).

[90] B. Liskov und L. Shrira. “Promises: linguistic support for efficient asynchronous procedure calls in distributed systems”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, 260–267. ISBN: 978-0-89791-269-3. DOI: 10.1145/53990.54016.

[91] M. Macero. *Reactive Programming with Java 9’s Flow*. en. 31. Jan. 2018. URL: https://thepracticaldeveloper.com/reactive-programming-java-9-flow/ (besucht am 22. 04. 2023).

[92] P. Maes. “Concepts and experiments in computational reflection”. In: *SIGPLAN Not.* 22.12 (1987), 147–155. ISSN: 0362-1340. DOI: 10.1145/38807.38821.

[93] S. Maffeis. “Adding Group Communication and Fault-Tolerance to CORBA”. In: *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS 1995)*. https://www.usenix.org/conference/coots-95/adding-group-communication-and-fault-tolerance-corba. Monterey, CA, USA: USENIX Association, Juni 1995.

[94] L Magnoni. “Modern Messaging for Distributed Sytems”. In: *Journal of Physics: Conference Series* 608.1 (2015), S. 012038. DOI: 10.1088/1742-6596/608/1/012038.

[95] P. Mandl, A. Bakomenko und J. Weiss. *Grundkurs Datenkommunikation: TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards*. Technische und Ingenieurinformatik. Vieweg+Teubner Verlag, 2010. ISBN: 978-3-83489-699-5.

[96] J. L. Martins und S. Duarte. “Routing algorithms for content-based publish/subscribe systems”. In: *IEEE Communications Surveys & Tutorials* 12.1 (2010), S. 39–58. DOI: 10.1109/SURV.2010.020110.00065.

[97] M. Mattsson, J. Bosch und M. E. Fayad. “Framework integration problems, causes, solutions”. In: *Commun. ACM* 42.10 (1999), 80–87. ISSN: 0001-0782. DOI: 10.1145/317665.317679.

[98] A. Medina, A. Lakhina, I. Matta und J. Byers. *BRITE: Boston university Representative Internet Topology gEnerator*. URL: https://www.cs.bu.edu/brite/ (besucht am 06. 02. 2022).

[99] Microsoft. *Tutorial: Publish and subscribe messages using WebSocket API and Azure Web PubSub service SDK*. 13. März 2023. URL: https://learn.microsoft.com/en-us/azure/azure-web-pubsub/tutorial-pub-sub-messages (besucht am 31. 07. 2024).

[100] Microsoft. *Details der XML-Serialisierung - .NET | Microsoft Learn*. 2024. URL: https://learn.microsoft.com/de-de/dotnet/standard/serialization/introducing-xml-serialization (besucht am 31. 07. 2024).

[101] P. Millard, P. Saint-Andre und R. Meijer. *XEP-0060: Publish-Subscribe (XMPP-Extension)*. https://xmpp.org/extensions/xep-0060.html. Sep. 2023.

[102] R. Minson und G. Theodoropoulos. “Adaptive Interest Management via Push-Pull Algorithms”. In: *2006 Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications*. 2006, S. 119–126. DOI: 10.1109/DS-RT.2006.8.

[103] G. Muehl, M. Werner, M. A. Jaeger, K. Herrmann und H. Parzyjegla. “On the Definitions of Self-Managing and Self-Organizing Systems”. In: *Communication in Distributed Systems - 15. ITG/GI Symposium*. 2007, S. 1–11. ISBN: 978-3-8007-2980-7.

[104] G. Mühl, L. Fiege und P. R. Pietzuch. *Distributed event-based systems*. Springer, 2006. ISBN: 978-3-540-32651-9. DOI: 10.1007/3-540-32653-7.

[105] G. Mühl, H. Parzyjegla und M. Prellwitz. “Analyzing Content-Based Publish/Subscribe Systems”. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS 2015)*. Hrsg. von F. Eliassen und R. Vitenberg. Olso, Norway: ACM, Juni 2015, S. 128–139. ISBN: 978-1-4503-3286-6. DOI: 10.1145/2675743.2771836.

[106] N. Naik. “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP”. In: *2017 IEEE International Systems Engineering Symposium (ISSE)*. 2017, S. 1–7. DOI: 10.1109/SysEng.2017.8088251.

[107] M. Nast, H. Raddatz, B. Rother, F. Golatowski und D. Timmermann. “A Survey and Comparison of Publish/Subscribe Protocols for the Industrial Internet of Things (IIoT)”. In: *Proceedings of the 12th International Conference on the Internet of Things*. IoT ’22. Delft, Netherlands: Association for Computing Machinery, 2023, 193–200. ISBN: 9781450396653. DOI: 10.1145/3567445.3571107.

[108] G. Neufeld und S. Vuong. “An overview of ASN.1”. In: *Computer Networks and ISDN Systems* 23.5 (1992), S. 393–415. ISSN: 0169-7552. DOI: 10.1016/0169-7552(92)90014-H.

[109] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification, Version 3.0.3*. http://www.omg.org/spec/CORBA/3.0.3/. März 2004.

[110] R. Oechsle. *Java-Komponenten: Grundlagen, prototypische Realisierung und Beispiele für Komponentensysteme*. 1. Aufl. Hanser, 2013, S. 310. ISBN: 978-3-446-43591-9. DOI: 10.3139/9783446435919.

[111] OpenDDS Foundation. *OpenDDS*. Juli 2024. URL: https://opendds.org/ (besucht am 03. 08. 2024).

[112] Oracle Inc. *Java© Platform, Standard Edition & Java Development Kit Version 11 API Specification*. URL: https://docs.oracle.com/en/java/javase/11/docs/api/index.html (besucht am 06. 07. 2024).

[113] Oracle Inc. *Lesson: Annotations (The Java™ Tutorials > Learning the Java Language)*. 2022. URL: https://docs.oracle.com/javase/tutorial/java/annotations/index.html (besucht am 23. 06. 2024).

[114] Oracle Inc. *Dynamic Proxy Classes*. 2024. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html (besucht am 23. 06. 2024).

[115] G. Pardo-Castellote. “OMG Data-Distribution Service: architectural overview”. In: *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* 2003, S. 200–206. DOI: 10.1109/ICDCSW.2003.1203555.

[116] H. Parzyjegla, D. Graff, A. Schröter, J. Richling und G. Mühl. “Design and Implementation of the Rebeca Publish/Subscribe Middleware”. In: *From Active Data Management to Event-Based Systems and More: Papers in Honor of Alejandro Buchmann on the Occasion of His 60th Birthday.* Hrsg. von K. Sachs, I. Petrov und P. Guerrero. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 124–140. ISBN: 978-3-642-17226-7. DOI: 10.1007/978-3-642-17226-7_8.

[117] H. Parzyjegla, M. Prellwitz und G. Mühl. “Organizing and Evaluating Publish/Subscribe Systems with Scopes”. In: *Proceedings of the 2015 International Conference on Networked Systems (NetSys 2015).* Cottbus, Germany: IEEE, März 2015, S. 1–8. ISBN: 978-1-4799-5804-7. DOI: 10.1109/NetSys.2015.7089078.

[118] L. L. Peterson und B. S. Davie. *Computernetze : eine systemorientierte Einführung.* ger. Hrsg. von L. L. Peterson. Dt. Ausg. der 4. amerikanischen Aufl. dpunkt-lehrbuch. Heidelberg: dpunkt-verl., 2008, S. 831. ISBN: 978-3-89864-491-4.

[119] R. Pfeifer, M. Lungarella und F. Iida. “Self-Organization, Embodiment, and Biologically Inspired Robotics”. In: *Science* 318.5853 (2007), S. 1088–1093. DOI: 10.1126/science.1145803.

[120] J. Pfrommer, S. Grüner und F. Palm. “Hybrid OPC UA and DDS: Combining architectural styles for the industrial internet”. In: *Proceedings of the IEEE World Conference on Factory Communication Systems (WFCS 2016).* Aveiro, Portugal: IEEE, Mai 2016. DOI: 10.1109/WFCS.2016.7496515.

[121] G. P. Picco, M. Migliavacca, A. L. Murphy und G.-C. Roman. “Distributed Abstract Data Types”. In: *On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE.* Bd. 4276. LNCS. Springer, 2006, S. 1594–1612. ISBN: 978-3-540-48274-1. DOI: 10.1007/11914952_40.

[122] P. Pietzuch und J. Bacon. “Hermes: a distributed event-based middleware architecture”. In: *Proceedings 22nd International Conference on Distributed Computing Systems Workshops.* 2002, S. 611–618. DOI: 10.1109/ICDCSW.2002.1030837.

[123] K. Pinte, A. Lombide Carreton, E. Gonzalez Boix und W. De Meuter. “Ambient Clouds: Reactive Asynchronous Collections for Mobile Ad Hoc Network Applications”. In: *Proceedings of the 13th International Conference on Distributed Applications and Interoperable Systems.* Florence, Italy: Springer Berlin Heidelberg, Juni 2013, S. 85–98. ISBN: 978-3-642-38541-4. DOI: 10.1007/978-3-642-38541-4_7.

[124] M. Prellwitz. “Programming Abstractions for Organic Computing Applications”. In: *Organic Computing - Doctoral Dissertation Colloquium 2014.* Hrsg. von S. Tomforde und B. Sick. kassel university press, 2014, S. 3–13. ISBN: 978-3-86219-832-0.

[125] M. Prellwitz. “Programming Abstractions for Ubiquitous Applications”. In: *Proceedings of the 8th Joint Workshop of the German Research Training Groups in Computer Science - Dagstuhl 2014, Dagstuhl, Germany, June 15-18, 2014*. Hrsg. von A. Jentzsch, T. Pape und S. Pasewaldt. Pro Business GmbH, 2014, S. 94. ISBN: 978-3-86386-719-5.

[126] M. Prellwitz, H. Parzyjegla und G. Mühl. “Dynamic Sets: A Programming Abstraction for Object Bundling”. In: *14th International Workshop on Adaptive and Reflective Middleware*. ARM 2015. Vancouver, BC, Canada: ACM, 2015, 9:1–9:3. ISBN: 978-1-4503-3733-5. DOI: 10.1145/2834965.2834973.

[127] M. Prellwitz, H. Parzyjegla und G. Mühl. “Adaptive Information Distribution for Dynamic Sets Using Multicast Push and Pull”. In: *SIGAPP Applied Computing Review* 18.3 (Okt. 2018), S. 19–31. ISSN: 1559-6915. DOI: 10.1145/3284971.3284974.

[128] M. Prellwitz, H. Parzyjegla und G. Mühl. “Programming Abstractions for Messaging Protocols in Event-based Systems”. In: *Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’23. Neuchatel, Switzerland: Association for Computing Machinery, 2023, 157–167. ISBN: 979-8-40070-122-1. DOI: 10.1145/3583678.3596896.

[129] M. Prellwitz, H. Parzyjegla, G. Mühl und D. Timmermann. “Dynamic Sets: A Programming Abstraction for Ubiquitous Computing and the Internet of Things”. In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. MOTA ’16. Trento, Italy: ACM, 2016, 1:1–1:6. ISBN: 978-1-4503-4669-6. DOI: 10.1145/3007203.3007213.

[130] M. Prellwitz, H. Parzyjegla, S. Steiner und G. Mühl. “Adaptive Information Distribution for Dynamic Sets”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC 2018)*. Hrsg. von H. M. Haddad, R. L. Wainwright und R. Chbeir. au, France: ACM, Apr. 2018, S. 395–402. ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167177.

[131] J. Protic, M. Tomasevic und V. Milutinovic. “Distributed shared memory: concepts and systems”. In: *IEEE Parallel & Distributed Technology: Systems & Applications* 4.2 (1996), S. 63–71. DOI: 10.1109/88.494605.

[132] S. Purtzel, S. Akili und M. Weidlich. “Predicate-based push-pull communication for distributed CEP”. In: *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’22. Copenhagen, Denmark: Association for Computing Machinery, Juni 2022, 31–42. ISBN: 978-1-45039-308-9. DOI: 10.1145/3524860.3539640.

[133] A. Rahman und E. Dijk. *Group Communication for the Constrained Application Protocol (CoAP)*. IETF: Internet Engineering Task Force (IETF), Okt. 2014. DOI: 10.17487/RFC7390.

[134] V. Rajamani, S. Kabadayi und C. Julien. “An Interrelational Grouping Abstraction for Heterogeneous Sensors”. In: *Transactions on Sensor Networks* 5.3 (2009), 27:1–27:31. ISSN: 1550-4859. DOI: 10.1145/1525856.1525865.

[135] M. A. Razzaque, M. Milojevic-Jevric, A. Palade und S. Clarke. “Middleware for Internet of Things: A Survey”. In: *IEEE Internet of Things Journal* 3.1 (2016), S. 70–95. DOI: 10.1109/JIOT.2015.2498900.

[136] *ReactiveX Observable*. 2024-06-10. URL: https://reactivex.io/documentation/observable.html (besucht am 22. 07. 2024).

[137] Real-Time Innovations (RTI). *Connext Product Suite*. 2024. URL: https://www.rti.com/products (besucht am 06. 08. 2024).

[138] S. M. Roberto Chinnici, C. Jean-Jacques Moreau, I. Arthur Ryman und W. Sanjiva Weerawarana, Hrsg. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. 27. Juni 2007. URL: http://www.w3.org/TR/wsdl20 (besucht am 28. 07. 2024).

[139] K. Rose, S. Eldridge und L. Chapin. *The Internet of Things: An Overview – Understanding the Issues and Challenges of a More Connected World*. http://www.internetsociety.org/sites/default/files/ISOC-IoT-Overview-20151221-en.pdf. Okt. 2015.

[140] A. Rotem-Gal-Oz. “Fallacies of distributed computing explained”. https://arnon.me/wp-content/uploads/Files/fallacies.pdf. 2006.

[141] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core, RFC 6120*. IETF: Internet Engineering Task Force (IETF), März 2011. DOI: 10.17487/RFC6120.

[142] P. Saint-Andre, K. Smith und R. Tronon. *XMPP: The Definitive Guide Building Real-Time Applications with Jabber Technologies*. O’Reilly Media, Inc., 2009, S. 308. ISBN: 978-0-596-52126-4.

[143] J. Salas, F. Perez-Sorrosal, M. Patiño Martínez und R. Jiménez-Peris. “WS-Replication: A Framework for Highly Available Web Services”. In: *WWW ’06: Proceedings of the 15th international conference on World Wide Webb*. Edinburgh, Scotland: ACM, 2006, S. 357–366. ISBN: 978-1-59593-323-2. DOI: 10.1145/1135777.1135831.

[144] A. Saleh, R. Morabito, S. Tarkoma, S. Pirttikangas und L. Lovén. *Towards Message Brokers for Generative AI: Survey, Challenges, and Opportunities*. Dez. 2023. DOI: 10.48550/arXiv.2312.14647.

[145] A. Saleh, S. Tarkoma, S. Pirttikangas und L. Lovén. *Publish/Subscribe for Edge Intelligence: Systematic Review and Future Prospects*. en. Techn. Ber. 4872730. Rochester, NY, Juni 2024. DOI: 10.2139/ssrn.4872730.

[146] A. Schill und T. Springer. *Verteilte Systeme : Grundlagen und Basistechnologien*. Springer, Berlin, Heidelberg, 2007. ISBN: 978-3-540-68471-8. DOI: 10.1007/978-3-540-68471-8.

[147] J. Schmitt und L. Wolf. *Quality of Service - An Overview.* https://www.kom.tu-darmstadt.de/publications/SW97-3. Apr. 1997.

[148] S. Sehic, F. Li, S. Nastic und S. Dustdar. “A Programming Model for Context-aware Applications in Large-scale Pervasive Systems”. In: *8th Int’l Conf. on Wireless and Mobile Computing, Networking and Communications.* IEEE, 2012, S. 142–149. ISBN: 978-1-4673-1429-9. DOI: 10.1109/WiMOB.2012.6379067.

[149] A. Serianni und F. De Rango. “Application Layer Protocols for Internet of Things”. In: *Advances in Computing, Informatics, Networking and Cybersecurity: A Book Honoring Professor Mohammad S. Obaidat’s Significant Scientific Contributions.* Hrsg. von P. Nicopolitidis, S. Misra, L. T. Yang, B. Zeigler und Z. Ning. Cham: Springer International Publishing, 2022, S. 535–558. ISBN: 978-3-030-87049-2. DOI: 10.1007/978-3-030-87049-2_18.

[150] Z. Shelby, K. Hartke und C. Bormann. *The Constrained Application Protocol (CoAP).* The Constrained Application Protocol (CoAP). Internet Engineering Task Force (IETF), Juni 2014. DOI: 10.17487/RFC7252.

[151] J. Siegel. “An Overview of CORBA 3”. In: *Distributed Applications and Interoperable Systems II.* Hrsg. von L. Kutvonen, H. König und M. Tienari. Boston, MA: Springer US, 1999, S. 119–132. ISBN: 978-0-387-35565-8. DOI: 10.1007/978-0-387-35565-8_9.

[152] J. Sifakis. “A vision for computer science — the system perspective”. In: *Open Computer Science* 1.1 (2011), S. 108–116. DOI: 10.2478/s13537-011-0008-y.

[153] D. Silva, L. I. Carvalho, J. Soares und R. C. Sofia. “A Performance Analysis of Internet of Things Networking Protocols: Evaluating MQTT, CoAP, OPC UA”. In: *Applied Sciences* 11.11 (2021). ISSN: 2076-3417. DOI: 10.3390/app11114879.

[154] W. Sim, B. Song, J. Shin und T. Kim. “Data Distribution Service Converter Based on the Open Platform Communications Unified Architecture Publish–Subscribe Protocol”. In: *Electronics* 10.20 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10202524.

[155] E. Sisinni, A. Saifullah, S. Han, U. Jennehag und M. Gidlund. “Industrial Internet of Things: Challenges, Opportunities, and Directions”. In: *IEEE Transactions on Industrial Informatics* 14.11 (2018), S. 4724–4734. DOI: 10.1109/TII.2018.2852491.

[156] M. Slee, A. Agarwal und M. Kwiatkowski. *Thrift: Scalable cross-language services implementation.* 2007. URL: https://thrift.apache.org/static/files/thrift-20070401.pdf (besucht am 28. 04. 2023).

[157] SmartThings, Inc., Hrsg. *Welcome to SmartThings.* 2023. URL: https://developer.smartthings.com/docs/ (besucht am 17. 08. 2024).

[158] B. Snyder, D. Bosanac und R. Davies. *ActiveMQ in Action.* Manning Publications Co., 2011.

[159] R. Srinivasan. *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC. Internet Engineering Task Force (IETF), Aug. 1995. DOI: 10.17487/RFC1831.

[160] W. Stallings und G. K. Paul. "Operating systems : internals and design principles". eng. In: 7. Aufl. Pearson, 2012. ISBN: 978-0-13230-998-1.

[161] M. van Steen und A. S. Tanenbaum. "A brief introduction to distributed systems". en. In: *Computing* 98.10 (Okt. 2016), S. 967–1009. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0508-7.

[162] D. H. Steinberg und S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., Dez. 2005. ISBN: 9780596101008.

[163] R. Steinmetz und K. Nahrstedt. "Quality of Service". In: *Multimedia Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 9–76. ISBN: 978-3-662-08878-4. DOI: 10.1007/978-3-662-08878-4_2.

[164] *STOMP Protocol Specification, Version 1.2*. 8. März 2021. URL: https://stomp.github.io/stomp-specification-1.2.html (besucht am 22. 04. 2023).

[165] R. Storm. *Wahrscheinlichkeitsrechnung, mathematische Statistik und statistische Qualitätskontrolle*. 10. Aufl. Fachbuchverlag Leipzig – Köln, 1995. ISBN: 978-3-34300-871-1.

[166] P. Szostack. "Dienstgüteaspekte für verteilte dynamische Objektmengen". unveröffentlicht. Master's Thesis. Universität Rostock, Juli 2017.

[167] A. S. Tanenbaum und M. v. Steen. *Verteilte Systeme : Prinzipien und Paradigmen*. 2. Aufl. it-informatik. 761 Seiten, Illustrationen, Diagramme, Karten, 25 cm. München: Pearson, 2008. ISBN: 978-3-8273-7293-2 | 3-8273-7293-3.

[168] A. S. Tanenbaum und D. Wetherall. *Computernetzwerke*. ger. 5. Aufl. Pearson Studium - IT. München: Pearson, 2012, S. 1032. ISBN: 978-3-86326-536-6 | 978-1-299-74700-5.

[169] S. Tarkoma. *Publish/Subscribe Systems: Design and Principles*. John Wiley & Sons, 2012. DOI: 10.1002/9781118354261.

[170] The Apache Software Foundation. *ActiveMQ – Flexible & Powerful Open Source Multi-Protocol Messaging*. 2024. URL: https://activemq.apache.org/ (besucht am 28. 07. 2024).

[171] The Apache Software Foundation. *OpenWire | ActiveMQ*. 29. Juli 2024. URL: https://activemq.apache.org/components/classic/documentation/openwire (besucht am 07. 08. 2024).

[172] M. Tian, A. Gramm, T. Naumowicz, H. Ritter und J. Freie. "A concept for QoS integration in Web services". In: *Fourth International Conference on Web Information Systems Engineering Workshops, 2003. Proceedings.* 2003, S. 149–155. DOI: 10.1109/WISEW.2003.1286797.

[173] S. Tomforde, B. Sick und C. Müller-Schloer. *Organic Computing in the Spotlight*. 2017. DOI: 10.48550/arXiv.1701.08125.

[174] Universität Rostock, Fakultät für Informatik und Elektrotechnik. *Multimodal Smart Appliance Ensembles for Mobile Applications (MuSAMA).* Jan. 2020. URL: https://musama.informatik.uni-rostock.de/ (besucht am 05. 04. 2024).

[175] Universität Rostock, Fakultät für Informatik und Elektrotechnik. *Multimodal Smart Appliance Ensembles for Mobile Applications (MuSAMA) – Structure.* Jan. 2020. URL: https://musama.informatik.uni-rostock.de/musama.informatik.uni-rostock.de/musama_struktur.html (besucht am 05. 06. 2024).

[176] Universität Rostock, Mobile Multimedia Information Systems. *Smart Appliance Laboratory (Bild).* 14. Dez. 2018. URL: https://www.mmis.informatik.uni-rostock.de/research/infrastructure/smart-appliance-laboratory/ (besucht am 18. 07. 2024).

[177] R.-G. Urma, M. Fusco und A. Mycroft. *Java 8 in Action: Lambdas, Streams, and functional-style programming.* 1. Aufl. USA: Manning Publications Co., 2014. ISBN: 978-1-61729-199-9.

[178] S. Van Deursen und M. Seemann. *Dependency Injection: Principles, Practices, and Patterns.* Manning Publications, 2019. ISBN: 978-1-61729-473-0.

[179] P. A. Vicaire, E. Hoque, Z. Xie und J. A. Stankovic. "Bundle: a group based programming abstraction for cyber physical systems". In: *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems.* ICCPS '10. Stockholm, Sweden: Association for Computing Machinery, 2010, 32–41. ISBN: 9781450300667. DOI: 10.1145/1795194.1795200.

[180] C. Walls und F. Langenau. *Spring im Einsatz.* 3. Aufl. Carl Hanser Verlag GmbH Co KG, 2020. ISBN: 978-3-446-45731-7. DOI: 10.3139/9783446457317.

[181] Z. Wang, Y. Zhang, X. Chang, X. Mi, Y. Wang, K. Wang und H. Yang. "Pub/Sub on stream: a multi-core based message broker with QoS support". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems.* DEBS '12. Berlin, Germany: Association for Computing Machinery, 2012, 127–138. ISBN: 9781450313155. DOI: 10.1145/2335484.2335499.

[182] R. Want und T. Pering. "System challenges for ubiquitous & pervasive computing". In: *Proceedings of the 27th International Conference on Software Engineering.* ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005, 9–14. ISBN: 1581139632. DOI: 10.1145/1062455.1062463.

[183] M. Weiser. "Hot topics-ubiquitous computing". In: *Computer* 26.10 (1993), S. 71–72. DOI: 10.1109/2.237456.

[184] M. Weiser. "The computer for the 21st century". In: *Scientific American* 265.3 (Sep. 1991), S. 94–104. DOI: 10.1145/329124.329126.

[185] M. Weiser und J. S. Brown. “The Coming Age of Calm Technology”. In: *Beyond Calculation: The Next Fifty Years of Computing*. New York, NY: Springer New York, 1997, S. 75–85. ISBN: 978-1-4612-0685-9. DOI: 10.1007/978-1-4612-0685-9_6.

[186] *X.901: Information technology - Open Distributed Processing - Reference Model: Overview*. Techn. Ber. International Telecommunication Union, Aug. 1997.

[187] *X.920: Information technology - Open Distributed Processing - Interface Definition Language*. Techn. Ber. International Telecommunication Union, Dez. 1997.

[188] M. Yassein, M. Shatnawi und D. Al-zoubi. “Application Layer Protocols for the Internet of Things: A Survey”. In: *2016 International Conference on Engineering MIS (ICEMIS)*. Agadir, Morocco: IEEE, Sep. 2016, S. 1–4. DOI: 10.1109/ICEMIS.2016.7745303.

[189] C. Yeo, B. Lee und M. Er. “A survey of application level multicast techniques”. In: *Computer Communications* 27.15 (2004), S. 1547–1568. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2004.04.003.

[190] S. Zhu und C. V. Ravishankar. “Stochastic Consistency, and Scalable Pull-based Caching for Erratic Data Stream Sources”. In: *Proceedings of the 30th Int’l Conference on Very Large Data Bases*. VLDB ’04. Toronto, Canada: VLDB Endowment, 2004, S. 192–203. ISBN: 0-12-088469-0. DOI: 10.5555/1316689.1316708.

Veröffentlichungen

Erstautor

1. M. Prellwitz. "Programming Abstractions for Organic Computing Applications". In: *Organic Computing - Doctoral Dissertation Colloquium 2014*. Hrsg. von S. Tomforde und B. Sick. kassel university press, 2014, S. 3–13. ISBN: 978-3-86219-832-0

2. M. Prellwitz, H. Parzyjegla und G. Mühl. "Dynamic Sets: A Programming Abstraction for Object Bundling". In: *14th International Workshop on Adaptive and Reflective Middleware*. ARM 2015. Vancouver, BC, Canada: ACM, 2015, 9:1–9:3. ISBN: 978-1-4503-3733-5. DOI: `10.1145/2834965.2834973`

3. M. Prellwitz, H. Parzyjegla, G. Mühl und D. Timmermann. "Dynamic Sets: A Programming Abstraction for Ubiquitous Computing and the Internet of Things". In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. MOTA '16. Trento, Italy: ACM, 2016, 1:1–1:6. ISBN: 978-1-4503-4669-6. DOI: `10.1145/3007203.3007213`

4. M. Prellwitz, H. Parzyjegla, S. Steiner und G. Mühl. "Adaptive Information Distribution for Dynamic Sets". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC 2018)*. Hrsg. von H. M. Haddad, R. L. Wainwright und R. Chbeir. au, France: ACM, Apr. 2018, S. 395–402. ISBN: 978-1-4503-5191-1. DOI: `10.1145/3167132.3167177`

5. M. Prellwitz, H. Parzyjegla und G. Mühl. "Adaptive Information Distribution for Dynamic Sets Using Multicast Push and Pull". In: *SIGAPP Applied Computing Review* 18.3 (Okt. 2018), S. 19–31. ISSN: 1559-6915. DOI: `10.1145/3284971.3284974`

6. M. Prellwitz, H. Parzyjegla und G. Mühl. "Programming Abstractions for Messaging Protocols in Event-based Systems". In: *Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems*. DEBS '23. Neuchatel, Switzerland: Association for Computing Machinery, 2023, 157–167. ISBN: 979-8-40070-122-1. DOI: `10.1145/3583678.3596896`

Koautor

7. G. Mühl, H. Parzyjegla und M. Prellwitz. “Analyzing Content-Based Publish/Subscribe Systems”. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS 2015)*. Hrsg. von F. Eliassen und R. Vitenberg. Olso, Norway: ACM, Juni 2015, S. 128–139. ISBN: 978-1-4503-3286-6. DOI: 10.1145/2675743.2771836
8. H. Parzyjegla, M. Prellwitz und G. Mühl. “Organizing and Evaluating Publish/Subscribe Systems with Scopes”. In: *Proceedings of the 2015 International Conference on Networked Systems (NetSys 2015)*. Cottbus, Germany: IEEE, März 2015, S. 1–8. ISBN: 978-1-4799-5804-7. DOI: 10.1109/NetSys.2015.7089078