Dynamische Verwaltung heterogener Kontextquellen in global verteilten Systemen

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Thomas Hamann
geboren am 20. Februar 1976 in Räckelwitz

Gutachter:

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill	TU Dresden
Prof. Dr.-Ing. habil. Martin Wollschlaeger	TU Dresden
Dr.-Ing. Albert Held	Daimler AG

Dresden im Dezember 2008

Für Anja, Lena und Paula

Danksagung

Die vorliegende Dissertation entstand während meines über einjährigen Promotionsstudiums am Lehrstuhl Rechnernetze der Technischen Universität Dresden. Sie wäre jedoch undenkbar ohne die inhaltlichen Grundlagen gewesen, welche ich zuvor in fünf Jahren Tätigkeit am selben Lehrstuhl legen konnte. Ich möchte den vielen Menschen danken, die Anteil am Erfolg dieser Arbeit hatten.

Zuerst danke ich Herrn Prof. Alexander Schill für die Möglichkeit die Promotion durchführen zu können. Ohne das von ihm entgegengebrachte Vertrauen und die Sicherheit eines damit verbundenen Promotionsstipendiums wäre die Beendigung der Dissertation in so kurzer Zeit unmöglich gewesen. Seine Bereitschaft innerhalb kürzester Zeit sowohl für inhaltliche Diskussionen als auch mit organisatorischem Rat zur Verfügung zu stehen, war für mich von unschätzbarem Wert. Ebenso möchte ich Herrn Prof. Martin Wollschlaeger und Herrn Dr. Albert Held für ihre Bereitschaft zur Begutachtung dieser Dissertation danken. Ihre wertvollen Fragen und Hinweise waren mir sehr willkommen.

Ich danke weiterhin meinen Kollegen für das stets freundliche Arbeitsklima und ihre Bereitschaft zu inhaltlichen Erörterungen. An erster Stelle möchte ich Gerald Hübsch nennen, welcher in der gleichen Situation wie ich, oft Zeit zur Diskussion – auch fachfremder Themen – hatte. Darüber hinaus gilt mein Dank Thomas Springer, Daniel Schuster, Stephan Gross, Iris Braun und Sandro Reichert für ihre Bemerkungen zum vorliegenden Manuskript sowie ihre Unterstützung bei der Vorbereitung der Verteidigung. Ohne die kritischen Fragen und die engagierte Mitwirkung von

Herrn Abdulsalam Ginebas wäre die Umsetzung der hier vorgestellten Konzepte nur schwer möglich gewesen.

Wenngleich das Stipendium denkbar günstige Voraussetzungen für diese Dissertation schuf, so gingen diesem doch mehrere Jahre der Vorbereitung voraus. Während dieser langen Zeit konnte ich mich immer auf die Unterstützung durch meine Familie verlassen. Besonders herzlich danke ich meiner Frau Anja. Sie schuf mir stets die erforderlichen zeitlichen Freiräume zur Anfertigung des Manuskriptes, selbst wenn dies mit großen persönlichen Einschränkungen verbunden war. Ich danke meinen Eltern Regina und Andreas und meinen Schwiegereltern Evelin und Frank für ihr Vertrauen, ihr Verständnis und ihre Unterstützung. Durch die Betreuung unserer Kinder Lena und Paula gaben sie mir nicht nur Zeit zur Weiterentwicklung der Dissertation, sondern gelegentlich auch zur notwendigen Entspannung. Besonders herzlich möchte ich mich bei meinen wundervollen Töchtern Lena und Paula für ihr Verständnis und ihre gelegentlich notwendigen Ablenkungen danken.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Ziele der Arbeit 2
1.2. Gliederung 2

2. Kontext **5**
2.1. Was ist Kontext? 5
2.1.1. Relevanz 7
2.1.2. Relevanzfaktoren und -funktion 9
2.1.3. Kontextabhängigkeit 11
2.1.4. Fazit und weitere Begriffe 12
2.2. Eigenschaften von Kontext 13
2.2.1. Dynamik 13
2.2.2. Imperfektion 15
2.3. Kontextklassen 16
2.3.1. Klassifikationen 16
2.3.2. Spezielle Kontexttypen 19
2.4. Kontextbezug und kontextbezogene Software 19
2.4.1. Kontextbereitstellung 19
2.4.2. Kontextnutzung 23
2.4.3. Dimensionen des Kontextzugriffs 25
2.5. Anforderungskatalog 27

2.5.1. Einschätzung 29

3. Anwendungsszenario **31**
3.1. Einführung in das Szenario 31
3.1.1. Aufgaben von Wartungsfirmen 32
3.1.2. Zeitaufwand 35
3.1.3. Notwendige Systemunterstützung 36
3.2. Informationen 36
3.2.1. Technische Dokumentation 36
3.2.2. Wartungsdokumentation 37
3.2.3. Kundendaten 38
3.2.4. Anlagenzustand 38
3.2.5. Umgebungsbeschreibung 40
3.2.6. Technikersituation 40
3.2.7. Heterogene Informationen 41
3.3. Akteure und Informationsquellen 43
3.3.1. Beim Anlagenhersteller 44
3.3.2. In der Wartungsfirma 44
3.3.3. Im Produktionsbetrieb 44
3.3.4. Beim Techniker 45
3.3.5. Gerätelandschaft 46
3.4. Informationssenken und -zugriff 47
3.4.1. Aufgaben 47
3.4.2. Senken 48
3.5. Quantitative Abschätzungen 49
3.5.1. Akteure 52
3.5.2. Informationsquellen, -knoten und -senken 53
3.5.3. Informationen und ihr Zugriff 55
3.6. Anwendungsanforderungen 58

4. Verwandte Arbeiten **61**
4.1. Ubiquitous Computing Middleware und Frameworks 62
4.1.1. Semantic Space und SOCAM 63
4.1.2. Context Broker Architecture 68
4.1.3. Gaia 70
4.1.4. Hydrogen 74
4.1.5. JCAF 76
4.1.6. Nexus 77
4.2. Vergleich 80
4.2.1. Zusammenfassung 80
4.2.2. Abgrenzung 84

4.3. Aktuelle Entwicklungen 84
4.3.1. Meteor 84
4.3.2. Weitere Arbeiten 87

5. Verwaltung heterogener Kontextquellen 89
5.1. Überblick 89
5.2. Context Provider 91
5.2.1. Provider Framework 93
5.3. Kontextmetamodell 97
5.4. Lokale Providerverwaltung 99
5.4.1. Providerbeschreibung 100
5.4.2. Filterung von Providerbeschreibungen 101
5.4.3. Lebenszyklus eines Providers 105
5.4.4. Provider gestartet 106
5.4.5. Provider gestoppt 111
5.4.6. Bemerkungen 113
5.5. Zentrale Providerverwaltung 114
5.5.1. Erreichbarkeit 115
5.5.2. Clients 116
5.5.3. Server 117
5.5.4. Nachrichten 118
5.5.5. Hybrider Austausch von Providerbeschreibungen 121
5.6. Dezentrale Providerverwaltung 129
5.6.1. Hybrides Peer-to-Peer-Netzwerk 130
5.6.2. Netzwerkverwaltung 133
5.6.3. Edge-Peer-Betrieb 134
5.6.4. Super-Peer-Betrieb 135
5.6.5. Aufwandsbetrachtungen 145

6. Realisierung und Evaluation 151
6.1. Realisierung des Context Service 151
6.1.1. Technische Voraussetzungen 152
6.2. Context Service Architektur 157
6.2.1. Mögliche Konfigurationen des Context Service 158
6.2.2. Die Zugriffskomponente 160
6.2.3. Die Verwaltungskomponente 168
6.2.4. Die Kommunikationskomponente 169
6.3. Implementierung der Context Provider 172
6.3.1. Aufgaben eines Providers 173
6.3.2. Externe Schnittstellen 174
6.3.3. Weitere Klassen 181

6.4. Realisierung von Consumern 185
6.4.1. Integration der Kontextsenken 185
6.4.2. Durchführung des Kontextzugriffs 187
6.5. Verwendung von Filtern und Konvertern 191
6.6. Evaluierender Einsatz des Context Service 192
6.6.1. Metamodelle 192
6.6.2. Entwicklungsmethodik 194
6.6.3. Laufzeitumgebung für multimodale, adaptive Software 195
6.6.4. Evaluationsanwendung 196
6.6.5. Bewertung des Einsatzes des Context Service 197
6.7. Überprüfung der funktionalen Anforderungen 198
6.8. Diskussion der nicht-funktionalen Anforderungen 207

7. Zusammenfassung und Ausblick 215
7.1. Zusammenfassung 215
7.2. Ausblick 217

A. XML Schemata für Nachrichteninhalte 219
A.1. Nachrichten für proaktive Verteilung 219
A.2. Nachrichten für hybriden Austausch 229

Literatur 231

Abkürzungen 243

Abbildungsverzeichnis

1.1. Verteilter Kontextdienst als Middleware 3

2.1. Exemplarische Relevanzfunktionen – kontinuierlich bzw. diskret . . . 10
2.2. Exemplarische Abstraktion mittels Fuzzylogik 14
2.3. Zusammenhänge der Anforderungen, welche durch die Eigenschaften von Kontext, sowie seiner Bereitstellung und Nutzung induziert sind 27

3.1. Heterogenität von Informationen – mittlere Volumina V und mittlere Änderungszeiten t_Δ 42
3.2. Zusammenhänge zusätzlicher Anforderungen, welche durch die Charakteristika des Anwendungsszenarios bedingt sind 58

4.1. Überblick der Architektur von SOCAM (nach [GPZ04b]) 65
4.2. Auswirkung homogener und heterogener Dienstlandschaften eines Semantic Space auf die Anfrageweiterleitung zwischen SLS Servern (angelehnt an [GPY05]) 67
4.3. Architektur von GaiaOS (angelehnt an [RC00, CHRC01, RHR+01, RHC+02]) 72
4.4. Architektur der Nexus-Plattform (angelehnt an [HKL+99]) 78

5.1. Kopplung von Consumern und Providern über den verteilten Context Service, sowie Kapselung und Anbindung von Kontextquellen durch Provider . . . 90
5.2. Emulation des Push-Zugriffs bei Kapselung einer Pull-basierten Kontextquelle . . . 95
5.3. Emulation des Pull-Zugriffs bei Kapselung einer Push-basierten Kontextquelle, welche nur bei Änderungen benachrichtigt . . . 95
5.4. Emulation des Pull- und Push-Zugriffs bei Kapselung einer Push-basierten Kontextquelle die regelmäßig benachrichtigt . . . 96
5.5. Typen des Kontextmetamodells . . . 98
5.6. Erweiterung der Restriktion eines Context Pattern um getypte Merkmale . . . 102
5.7. Datenfluss bei der Selektion zweier Provider unter Verwendung eines Filters und eines Konverters . . . 105
5.8. Lebenszyklus eines Context Providers . . . 106
5.9. Behandlung eines gestarteten Context Providers . . . 110
5.10. Behandlung eines gestoppten Context Providers . . . 112
5.11. Zerlegung eines realen High-Level Context Providers in drei virtuelle Provider . . . 114
5.12. Zustandsdiagramm eines Context Service Clients . . . 117
5.13. Zustandsdiagramm des Context Service Servers . . . 117
5.14. Relationen zwischen Nachrichtenumlaufzeit t_{RTT}, Erreichbarkeitsintervall t_{PING}, Antwortintervall t_{PONG} und Antwortschwelle m_{PING} 119
5.15. Korrelation der Topologien zwischen (I) physischem, (II) Super-Peer- und (III) Überlagerungsnetzwerk (angelehnt an [WGR05]) . . . 131
5.16. Protokollstapel eines Edge Peers und zweier Super Peers (angelehnt an [Zim80]) . . . 132
5.17. Übergänge zwischen Edge- und Super-Peer-Betrieb, inkl. Aktivierung und Deaktivierung des Edge- bzw. Super-Peer-Protokolls . . . 134
5.18. Realisierung des Super-Peer-Protokolls durch den DPI auf Basis von KBR . . . 136
5.19. Anfrageaufwand der dezentralen Verwaltung $m_{REQ_{p2p}}$ in Abhängigkeit von der Anzahl der Super Peers verglichen mit dem Anfrageaufwand der zentralen Verwaltung $m_{REQ_{c/s}}$ . . . 148
5.20. Aktualisierungsaufwand der dezentralen Verwaltung $m_{UPD_{p2p}}$ in Abhängigkeit von der Anzahl der Super Peers verglichen mit dem Aktualisierungsaufwand der zentralen Verwaltung $m_{UPD_{c/s}}$ . . . 149

6.1. Architektur des Laufzeitsystems . . . 152
6.2. Schichten der OSGi Service Platform (nach [OSG07a]) . . . 155

6.3. Bundles der Consumer, des Context Service und der Provider, sowie genutzte Bundles der OSGi Service Platform 158
6.4. Sinnvolle Konfigurationen des Context Service A) vollständig und verteilt; B) vollständig, nur lokal; C) nur Kontextnutzung und D) nur Kontextbereitstellung 159
6.5. Zentrale Klassen und Schnittstellen der Zugriffskomponente 161
6.6. Klassen für die Kontextereignisse der unterschiedlichen Typen des Metamodells 163
6.7. Schnittstellen der Kontextelemente, sowie Klassen ihrer Schlüssel . . 165
6.8. Klassen der Datenobjekte der Kontextelemente 167
6.9. Schnittstellen und Klassen zur Implementierung von Context Consumern und Context Providern 170
6.10. Möglichkeiten der Kopplung von Context Consumer und Kontextsenke: A) JNI B) entfernte Kommunikation C) OSGi Bundle und D) Foreign Application Access 186
6.11. Schnittstellen und Klassen für die Filterung von Context Patterns bei der Providerselektion 192
6.12. Komponenten der Laufzeitumgebung für multimodale, adaptive Software 195
6.13. Konfiguration der Evaluationsanwendung, sowie verteilte Kooperation ihrer Komponenten 197

Tabellenverzeichnis

2.1. Kontextklassen der inhaltlichen Klassifikation nach [SAW94] 16
2.2. Kontextklassen der quellenspezifischen Klassifikation nach [HI04] . . 18
2.3. Klassen kontextbezogener Anwendungen nach [SAW94] 24

3.1. Struktur, Volatilität und Volumen von Informationen 43
3.2. Umgebungen und ihre Informationsquellen 46
3.3. Kriterien und Formen des Informationszugriffs 50
3.4. Zugriff und Erreichbarkeit der Informationen bzw. ihrer Quellen . . . 51
3.5. Vorkommen der Akteure in den verschiedenen Umgebungen 53
3.6. Häufigkeit von Informationsquellen und ihren Knoten 55
3.7. Änderungsinduzierte, mittlere Nutzdatenraten und jährlich kumulierte Volumina der kommunizierten Informationen 57

4.1. Gegenüberstellung verwandter Arbeiten 82

5.1. Verwaltete Mengen von Context Patterns 107
5.2. Erreichbarkeitsnachrichten zur Überwachung der Clients und des Servers, sowie zur Verwaltung des Netzwerkes 120
5.3. Providernachrichten zur Verteilung der Beschreibungen verwendbarer und nicht-verwendbarer Provider 121

5.4. Zugriffsnachrichten zum Austausch von Kontextinformationen entfernter Knoten und ihrer Provider 122
5.5. Qualitativer Vergleich der proaktiven Verteilung und der reaktiven Abfrage von Providerbeschreibungen 125
5.6. Kommunikationsaufwand der proaktiven Verteilung, der reaktiven Abfrage und des hybriden Austausches von Providerbeschreibungen (in Nachrichten pro Zustandsänderung) 126
5.7. Providernachrichten für hybriden Austausch 128

Quelltextverzeichnis

6.1. Registrierung eines High-Level Providers 173
6.2. Implementierung eines Low-Level Providers 177
6.3. Synchroner Kontextzugriff durch einen Consumer 188
6.4. Asynchroner Kontextzugriff durch einen Subscriber 189

A.1. XSD der Nachrichteninhalte zur proaktiven Verteilung von Providerbeschreibungen 219
A.2. XSD der Nachrichteninhalte zum hybriden Austausch von Providerbeschreibungen 229

1. Einleitung

Nicht erst seit Mark Weisers Vision des Ubiquitous Computing [Wei91] sind Forscher damit beschäftigt, Software zu entwickeln, welche sich an ihre Nutzer und deren Situation anpasst. Eine Grundvoraussetzung dafür ist die Beobachtung von Umgebungen und Situationen, d. h. die Erfassung so genannter Kontextinformationen. Mit der fortschreitenden Miniaturisierung elektronischer, elektrischer und mechanischer Baugruppen sind elektronische Geräte heutzutage in jedem Teil des menschlichen Alltags anzutreffen. Durch ihre Anwesenheit wird die Beobachtung ermöglicht, weil einerseits die Geräte über Sensoren verfügen können, und andererseits schon die Identifikation der vorhandenen Geräte, Rückschlüsse auf die Umgebung erlaubt. In den meisten Szenarien ist eine Vernetzung der Geräte gegeben, sodass die erfassten Informationen potenziellen Anwendungen in jedem Bereich der Umgebung zur Verfügung stehen. Die Vernetzung erfolgt häufig durch den Einsatz von Ad-hoc-Netzwerken oder durch Anbindung der einzelnen Geräte an das Internet.

Während für die ersten ubiquitären Anwendungen oft spezielle Umgebungen in den Forschungslaboren geschaffen wurden, werden heutzutage immer mehr Systeme direkt für den Alltag entwickelt. Vor dem praktischen Einsatz sind jedoch noch immer zahlreiche Probleme zu bewältigen. So werden immer neue Ansätze erarbeitet, um die komplexe Entwicklung ubiquitärer Software zu vereinfachen. Aktuelle Verfahren verwenden – analog zum Semantic Web – prädikatenlogische Sprachen zur Modellierung von Kontext. Entsprechende Ausführungsumgebungen erlauben es, die Ausdrucksstärke der Sprachen zu nutzen, und die erfassten Kontextinforma-

tionen zu analysieren. Auf diese Weise können z. B. inkonsistente Beobachtungen korrigiert werden, oder es sind komplexe Schlussfolgerungen über die Situation der Nutzer möglich. Neben der Unterstützung der Modellierungsansätze sind die Ausführungsumgebungen dafür zuständig, die Vielzahl verschiedener Geräte zu einem gesamten System integrieren, und die Informationen jedes einzelnen Gerätes für die Übrigen zugreifbar zu machen.

Die erfassten Beobachtungen werden *Kontextinformationen* genannt, wenngleich schon zu Beginn dieser Arbeit erkennbar wird, dass eine klare Definition von *Kontext* nur schwer möglich ist. Geräte bzw. deren Hard- und Softwarekomponenten, welche die Kontextinformationen erfassen, sind so genannte *Kontextquellen*. Dem gegenüber werden Geräte und Anwendungen, die zur Anpassung an die Umgebung Kontextinformationen verarbeiten, *Kontextsenken* genannt.

1.1. Ziele der Arbeit

Die vorliegende Arbeit entwickelt eine Middleware, welche die dynamische Kopplung von Kontextsenken und Kontextquellen gestatten soll (Abb. 1.1 auf der nächsten Seite). Dazu gehört auch die transparente, skalierbare Kopplung der Middlewareinstanzen der vernetzten Geräte. Kontextsenken sollen auf alle erreichbaren Kontextquellen zugreifen können – unabhängig von der Entfernung zwischen beiden. Zur Behandlung der schwankenden Verfügbarkeit und Erreichbarkeit von Kontextquellen und Geräten soll die Verwaltung der Kontextquellen anstatt ihrer Kontextinformationen erfolgen. Die Kopplung der Geräte soll die beschränkten Ressourcen mobiler und integrierter Geräte berücksichtigen. Im Fokus der Betrachtungen sollen dynamische Kontextinformationen und deren Quellen stehen, weil ihre Nutzung aufgrund der Dynamik als problematisch anzusehen ist.

Wegen der Heterogenität von Kontextquellen und -informationen, sowie deren opportunistischer Nutzung durch die Kontextsenken, soll in dieser Arbeit nicht auf die Verwendung von Kontext für zeitkritische Funktionen eingegangen werden. Ebenso wenig sollen weitere Modellierungsansätze für Kontextinformationen entwickelt werden. In der Evaluation soll untersucht werden, ob der Verzicht auf ausdrucksstarke Modellierungsansätze praktikabel ist. Aufgrund der Fokussierung auf die dynamischen Aspekte der Kontextbereitstellung und -nutzung, werden Replikationsverfahren oder ähnliche Konzepte zur Erhöhung der Verfügbarkeit von Informationen nicht Gegenstand der Arbeit sein.

1.2. Gliederung

Im Anschluss erfolgt die Klärung des Begriffes *Kontext* und damit verwandter Begriffe, sowie die Analyse der Eigenschaften, der Entstehung und der Nutzung von

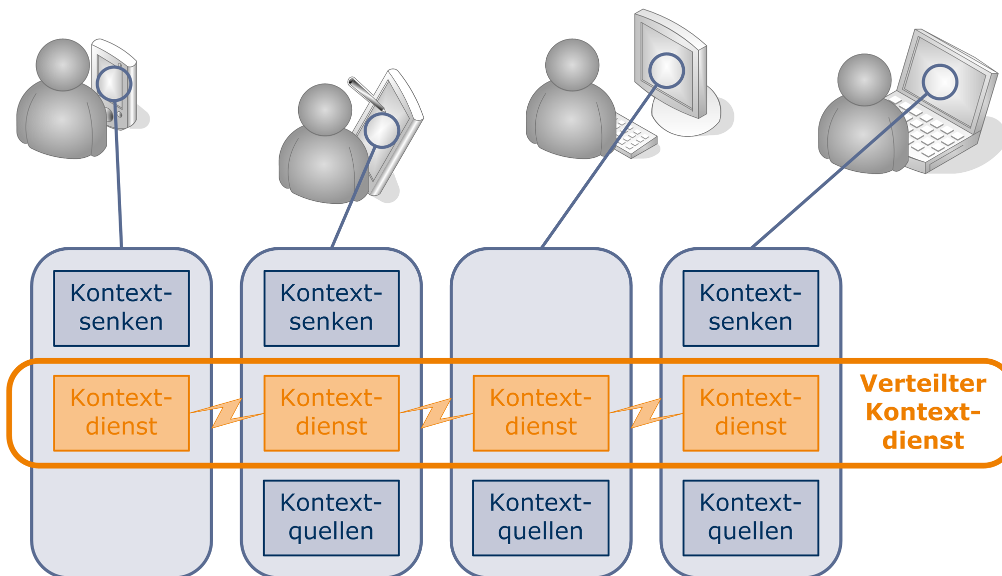

Abbildung 1.1.: Verteilter Kontextdienst als Middleware zwischen den Kontextquellen und Kontextsenken der Geräte einer Umgebung

Kontext. Die daraus gewonnenen Anforderungen werden durch Untersuchung eines umfangreichen Anwendungsszenarios (Kap. 3) vervollständigt. Kapitel 4 zeigt, inwieweit die Anforderungen durch die Konzepte verwandter Arbeiten bereits behandelt sind. Mit diesen Voraussetzungen wird ein Gesamtkonzept eines verteilten Kontextdienstes entwickelt und realisiert, sowie funktional und analytisch evaluiert. Den Abschluss bildet die Zusammenfassung der Arbeit, welche darüber hinaus mögliche Aspekte für zukünftige Forschungsaktivitäten aufzeigt.

2. Kontext

Weil Kontextinformationen von zentraler Bedeutung für die Schaffung ubiquitärer Anwendungen sind, wird nachfolgend der Begriff *Kontext* genauer untersucht. Das beinhaltet in erster Linie seine konkreten Eigenschaften, sowie die unterschiedlichen Klassen von Kontext (Kap. 2.2 und 2.3). Aber auch die Charakteristika der Kontextbereitstellung und -nutzung werden analysiert. Die in diesem Zuge identifizierten grundlegenden Anforderungen /A 1/ bis /A 14/ sind bereits der erste Teil der Anforderungsanalyse. Der Zweite erfolgt anhand eines anwendungsspezifischen Szenarios (Kap. 3).

2.1. Was ist Kontext?

Eine der am häufigsten zitierten Definitionen des Begriffes *Kontext* stammt von Anind K. Dey. In seiner Dissertationsschrift [Dey00] aus dem Jahre 2000 schreibt er:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves."

Diese Definition ist sehr allgemein gefasst, und bedarf weiterer Analyse. Deys Verständnis von Kontext als situative Beschreibung der Interaktion eines Nutzers mit

seiner Anwendung stimmt mit dem allgemeinen intuitiven Verständnis des Begriffes überein. Dies zeigt ein Blick in allgemeine Quellen, welche keine informatikspezifische Sichtweise präsentieren – die Brockhaus-Enzyklopädie [Bib90] definiert Kontext wie folgt:

> "Kontext ... (3) Sprachwissenschaft: Umgebung, in der eine sprachl. Einheit auftritt und die diese beeinflusst (sprachlicher K.); Umstände und Situation, in der ein Text geäußert und verstanden wird (situativer K.)."

Auch im Englischen Sprachraum ist eine ähnliche Definition anzutreffen. So definiert das Oxford Advanced Learners Dictionary [Hor95] Kontext mit:

> "context ... 2 circumstances in which sth happens or in which sth is to be considered: ..."

Somit kann ein Übersetzungsfehler des ungenau definierten Begriffes weitgehend ausgeschlossen werden*. Ohne Beschränkung der Allgemeinheit kann man Deys Aussage auf Software im Speziellen und IT†-Systeme im Allgemeinen ausweiten – nicht nur auf kontextbezogene oder ubiquitäre Software. Allerdings ist seine Definition zu allgemein. Denn jegliche Entitäten, welche an der Interaktion teilnehmen, werden als Kontext betrachtet. Dies schließt nicht nur den Nutzer, sondern vor allem die Umgebung des Nutzers mit all ihren Aspekten ein – inklusive des IT-Systems und all seiner Bestandteile. Entsprechend verallgemeinert Dey die Definition im weiteren Verlaufe seiner Arbeit wie folgt:

> "Our definition of context includes not only implicit input but also explicit input."

Dem muss jedoch widersprochen werden. Wären explizite Eingaben bzw. Informationen auch Kontext, dann würden jegliche Eingaben Kontext darstellen. Eine Unterscheidung der beiden Begriffe *Eingabe* und *Kontext* wäre nicht mehr möglich. Folglich besäße der Begriff *Kontext* keinerlei Signifikanz mehr gegenüber des Begriffes der *(gewöhnlichen, expliziten) Eingabe*, und bedürfte dementsprechend auch keiner eigenen Definition. Somit wären jegliche informationsverarbeitenden Systeme kontextbezogen. Eine Folgerung, welche Dey selbst in seiner Arbeit zieht, aber sogleich ohne Nennung von Gründen einschränkt:

> "Context-awareness uses a generalized model of input, including implicit and explicit input, allowing any application to be considered more or less context-aware insofar as it reacts to input. However, in this thesis, we will concentrate on the gathering and use of implicit input by applications."

*Für eine ausführlichere und interdisziplinäre Betrachtung des Begriffes *Kontext* sei der interessierte Leser auf [BD05] verwiesen.

†Informationstechnik (IT)

2.1.1. Relevanz

Lediglich Dey's Forderung nach der *Relevanz* einer Information für die Charakterisierung der Mensch-Maschine-Interaktion kann zur Konkretisierung der Definition und zur Einschränkung der zu betrachtenden Informationsmenge herangezogen werden. Naiv betrachtet, ist eine Information relevant, wenn sie eine Auswirkung auf die Situation hat. Dey selbst nimmt jedoch keine weitere Analyse des Relevanzaspektes vor. Dies soll hier nachgeholt werden.

Auf der Suche nach einer Definition wird man im Bereich des Information Retrieval fündig. Diese Disziplin beschäftigt sich mit der Suche von Informationen in großen Datenmengen. Im Ubiquitous Computing müssen Kontextinformationen vor ihrer Nutzung ebenfalls zuerst aus der Menge der übrigen irrelevanten Daten isoliert werden. Insofern sind sich Ubiquitous Computing und Information Retrieval durchaus ähnlich. Bei Letzterem wird ein Dokument als relevant betrachtet [Sto07],

> „wenn es objektiv zur Vorbereitung einer Entscheidung dient oder wenn es objektiv eine Wissenslücke schließt oder wenn es objektiv eine Frühwarnfunktion erfüllt.“

Als Pendant zum objektiv definierten Begriff der *Relevanz* existiert die subjektiv definierte *Pertinenz*, welche statt eines Systems den menschlichen Nutzer betrachtet, und sein kognitives Modell in die Betrachtung einbezieht. Folglich ist eine Information für einen Nutzer pertinent, d. h. subjektiv bedeutsam, wenn sie subjektiv entweder zur Vorbereitung einer Entscheidung dient, eine Wissenslücke schließt oder eine Frühwarnfunktion erfüllt. Beide Aspekte – Pertinenz und Relevanz – sind im Ubiquitous Computing die beiden Seiten derselben Medaille. Denn dort wird mit informationstechnischen Mitteln versucht, ein objektives Modell der Umgebung und des Nutzers zu ermitteln, welches als Grundlage des Verhaltens eines IT-Systems dienen kann. Ist die Übereinstimmung zwischen diesem objektiven Modell des Systems und dem subjektiven Modell des Nutzers nicht hinreichend groß, dann führt das Verhalten des Systems nicht zum gewünschten Erfolg – der Unterstützung des Nutzers. Dem entsprechend ist das System nicht in der Lage, Situationen, Motivationen und Ziele des Nutzers hinreichend genau zu erkennen, und kann in seinen Aktionen ein für den Nutzer unerwartetes und ungeeignetes, ja eventuell sogar kontraproduktives, Verhalten zeigen.

Die Übertragung der obigen Aussagen von Dokumenten auf die in ihnen enthaltenen Informationen und somit auch auf Informationen im Allgemeinen erlaubt die Anwendung der Begriffe Relevanz und Pertinenz auf Kontext, welcher in Deys Definition auch mit „any information“ (dt. jegliche Information) umschrieben wurde.

Folglich stellen nicht jegliche Informationen, welche in der Umgebung des Nutzers erfasst oder abgeleitet werden können, Kontext dar, sondern nur diejenigen,

welche relevant[*] sind. Auch wenn es im Forschungsbereich des Information Retrieval Ansätze zur automatischen Bestimmung der Relevanz von Informationen gibt [CH88], so erfolgt doch die grundsätzliche Entscheidung meist durch die Entwickler des kontextbezogenen Softwaresystems. Sie manifestieren ihre Wahl durch Einbeziehung konkreter Informationen in das Systemmodell. Alle darin nicht enthaltenen oder nicht vorgesehenen Informationen werden – bewusst oder unbewusst – als irrelevant eingestuft. Bei der Modellierung werden z. T. nur Informationstypen spezifiziert. Entsprechend grob granular[†] ist die mit der Modellierung einhergehende Entscheidung.

Dourish [Dou04] geht auf dieses Entscheidungsproblem ein, und stellt fest, dass die Relevanz ad hoc durch die Konstellationen der Entitäten in einer konkreten Situation bestimmt wird. Somit kann ihre Feststellung nicht a priori erfolgen. Dourish führt aus, dass Kontext und Inhalt (bzw. Aktivität) nicht voneinander getrennt werden können, weil Kontext aus der Aktivität heraus entsteht. Diese Darstellung verdeutlicht die Problematik:

- Einerseits erwächst daraus die allgemeine Forderung, dass die Aktivitäten des Nutzers und die dabei kommunizierten Inhalte (möglichst) maschinenverständlich sein sollen. Um dies zu erreichen, ist eine Analyse der Inhalte erforderlich, welche z. T. komplex und ressourcenintensiv ist (z. B. natursprachliche Erkennung). Sie erzeugt ergänzende Beschreibungen mit so genannten Metainformationen, welche die Semantik der ausgetauschten Inhalte repräsentieren. Alternativ können solche Informationen auch durch den Erzeuger der Inhalte hinzugefügt werden (z. B. Beschreibung einer Webseite mit Mitteln des Resource Description Framework (RDF) [KC04]).

- Andererseits führt dies zu einem erhöhten Bedarf hinsichtlich der Vielfalt von Informationstypen und der Menge zu erfassender Informationen, weil deren Relevanz ggf. erst zur Laufzeit festgestellt werden kann. Entsprechend präventiv werden Informationen in das Systemmodell integriert, welche zur Laufzeit ggf. als irrelevant identifiziert werden. Die Folge ist eine Ausweitung der Mengen von Kontexttypen und Kontextinformationen, welche in geeigneter Form verwaltet werden müssen, sodass eine effiziente Suche möglich ist.

- Weil Kontext aus dem Gegenstand der Aktivität – quasi dem Text – entsteht und je nach Situation und Nutzer variiert, ist es durchaus möglich bzw.

[*] Es wird in dieser Arbeit vorwiegend von Relevanz gesprochen, weil (a) technische Systeme betrachtet werden, und (b) Pertinenz mit technischen Mitteln nicht erfasst werden kann.

[†] In Kapitel 2.1.2 werden Faktoren präsentiert, welche über die Relevanz von Informationen entscheiden. Dabei handelt es sich nicht immer um typspezifische Entscheidungen – auch instanzspezifische Entscheidungen sind möglich.

wahrscheinlich, dass auch Anwendungsdaten Kontext sein können (oder umgekehrt). Somit wäre für Anwendungen ein homogener Zugriff auf Kontext und Anwendungsdaten ideal.

A 1. *Softwaresysteme bzw. Kontextsenken sollen gleichermaßen einheitlich auf Kontext- und Anwendungsinformationen zugreifen können. Dies beinhaltet neben einheitlichen Zugriffsfunktionen auch die einheitliche Repräsentation der Informationen, ungeachtet ihrer Art und ihres Ursprungs.*

2.1.2. Relevanzfaktoren und -funktion

Nicht immer wird die Entscheidung über die Relevanz durch den Nutzer auf Basis subjektiver Gründe getroffen. Es kann auch objektive Faktoren geben (siehe auch Kap. 2.2 und 2.3), welche über die Relevanz von Informationen entscheiden [SG01]. Ist eine Information ortsabhängig, so ist häufig zu beobachten, dass das Interesse einer Quelle an ihr mit zunehmender Entfernung vom Ursprungsort der Information sinkt. Ab einer bestimmten Entfernung ist die Information irrelevant. Neben dem eben genannten Ort bzw. der räumlichen Entfernung existiert noch ein zweiter wichtiger Faktor – die Zeit bzw. das Alter einer Information. Je älter Informationen sind, desto niedriger ist häufig deren Relevanz bzw. desto höher ist die Gefahr ihrer Ungültigkeit. Dies gilt primär für Informationen die dynamische Vorgänge beschreiben. Entsprechend der Definition von Relevanz ist diese durch den jeweiligen Nutzer bestimmt, d. h. die menschliche Person stellt ebenfalls einen – wenn nicht sogar den wichtigsten – Relevanzfaktor dar.

Die Relevanzfaktoren Ort und Zeit können auf zwei Arten definiert sein: entweder typspezifisch oder nutzerspezifisch*. Erstere beeinflussen die Relevanz in Abhängigkeit der Semantik des betreffenden Kontexttyps. Ein Beispiel dafür ist die Zeitinvarianz statischer (Kontext-)Informationen – sie sind immer gültig und relevant. Bei nutzerspezifischen Faktoren definiert der Nutzer – bewusst oder unbewusst – ob und in welchem Ausmaß ein potentieller Faktor die Relevanz von Informationen tatsächlich beeinflußt. Bei Betrachtung der Eigenschaften des Nutzers wird klar, dass auch Faktoren existieren können, die von elektronischen Systemen äußerst schwer erfassbar sind (z. B. menschliche Motivation, Intuition, Bedürfnisse, Emotionen, etc.).

Sind die Relevanzfaktoren bekannt und quantifizierbar, so kann u. U. eine entsprechende Relevanzfunktion definiert werden, welche die Ermittlung der Relevanz von Informationen erlaubt. Selbst für die grundlegenden Faktoren Zeit und Ort ist eine Erfassung in verteilten mobilen Systemen nicht trivial. Global synchrone Uhren sind ohne stark limitierende Annahmen oder nennenswerten technischen Aufwand (z. B. GPS-Empfänger auf jedem Knoten) nicht realisierbar. Ähnlich komplex ist die

*Die Rolle des Nutzers ist hier stellvertretend genannt: einerseits für ganze Nutzergruppen, andererseits für Software, welche stellvertretend für ihre Nutzer Kontext konsumiert.

Feststellung der Position von Knoten und Nutzern in einer heterogenen Umgebung, welche die Integration einer Vielfalt von Lokalisierungsmechanismen mit abweichenden Charakteristika (z. B. Genauigkeit, Präzision) und zugrunde liegenden Modellen mit sich bringt. Trotzdem ist das Wissen um die Relevanzfaktoren und -funktionen hilfreich, da ein Zugriff auf Kontextinformationen außerhalb ihres Relevanzbereiches nicht sinnvoll ist. Es bleibt zu klären, ob eine entsprechende technische Unterstützung möglich ist.

Kann die Relevanzfunktion formal beschrieben werden, dann steht ihrer Nutzung durch ein Softwaresystem grundsätzlich nichts im Wege. Wendet man die Funktion an, um die Verarbeitung irrelevanter Daten zu verhindern, so könnte dadurch die Belastung des Systems reduziert werden, was wiederum zur Verbesserung von Effizienz und Skalierbarkeit führt. All dies kann natürlich nur geschehen, wenn die konkreten Ausprägungen der Relevanzfaktoren während des Betriebes des Kontextdienstes erfasst werden können*. Im Rahmen dieser Arbeit wird nicht näher auf Erfassung und Modellierung der Faktoren, sowie die Modellierung und die Auswertung der Funktion eingegangen. Es sollen lediglich Maßnahmen betrachtet werden, welche bei Vorliegen der Faktoren und der Funktion, die Bestimmung der Relevanz erlauben, und so Einflussnahme auf die Arbeitsweise des Systems gestatten. Abbildung 2.1 zeigt zwei exemplarische Funktionen, welche die Relevanz von Parkplatzinformationen in Abhängigkeit von der Entfernung eines PKW-Fahrers (und seines Fahrzeuges) von diesen Parkplätzen angeben.

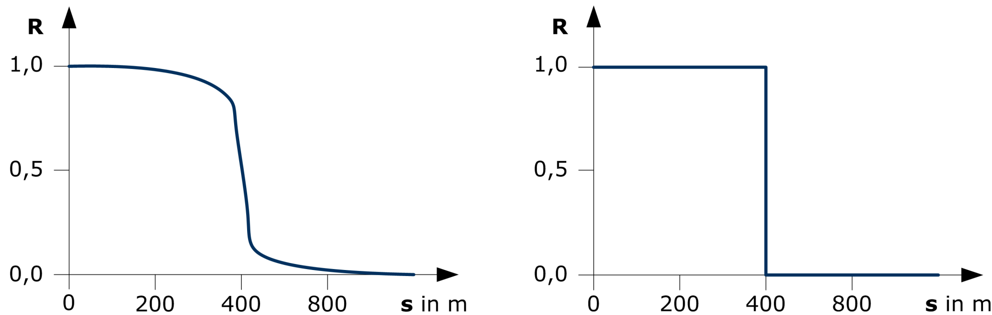

Abbildung 2.1.: Exemplarische Relevanzfunktionen – kontinuierlich bzw. diskret

A 2. *Die Ermittlung der Relevanz von Informationen ist durch eine generische Filterung der Kontextinformationen anhand verschiedener Kriterien zu unterstützen. Gleiches gilt auch für die Selektion entsprechender Kontextquellen.*

*Die Ermittlung des Einflusses des Nutzers bei der Bestimmung des Kontextes ist äußerst schwierig. Häufig muss der Nutzer seine Präferenzen explizit spezifizieren, oder es werden probabilistische Mechanismen für die Entscheidungsfindung eingesetzt, welche auf der Beobachtung des Nutzers basieren.

2.1.3. Kontextabhängigkeit

Zeitgleich zu Dey definieren Chen und Kotz [CK00] Kontext wie folgt:

> "Context is the set of environmental states and settings that either determines an application's behavior or in which an application event occurs and is interesting to the user."

Ohne den Begriff der Relevanz zu bemühen, beziehen sich die Autoren doch auf eines seiner wesentlichen Merkmale – den Aspekt der Beeinflussung von Entscheidungen durch Kontext. Im weiteren Verlauf der Diskussion stimmt das Verständnis von Relevanz bei Chen und Kotz aber nicht mit dem Standpunkt der vorliegenden Arbeit überein, wie man aus folgender Äußerung erkennen kann:

> "Further, we define the first kind of context as active context that influences the behaviors of an application, and the second kind of context as passive context that is relevant but not critical to an application."

Die Relevanz des passiven Kontexts muss hier, im Gegensatz zum Information Retrieval, von subjektiver bzw. ungenauer Art sein, andernfalls würde sie zu einer Beeinflussung der Anwendung führen. Dem entsprechend wäre passiver Kontext gleichbedeutend mit aktivem Kontext. Deshalb wird die Forderung nach Relevanz bei der Diskussion dieser Definition nicht berücksichtigt.

Jedoch zeigen die Autoren einen weiteren Aspekt der Kontextnutzung auf – den Grad der Abhängigkeit einer Software von Kontextinformationen. Wenn Kontext als implizite Information definiert wird, so ist eine Software von ihr weniger abhängig, als von expliziten Informationen – den Anwendungsdaten. Die Beziehung zwischen Software und Kontext wird im Rahmen dieser Arbeit viel mehr im Sinne von Bewusstheit verstanden. Dies wirkt sich direkt auf das Verständnis der Kontextnutzung aus (siehe Kap. 2.4.2), welche entsprechend robust gestaltet werden sollte, um das Fehlen von Kontextinformationen zu tolerieren.

Folglich wird, in Anlehnung an [SG01], nachfolgend der Begriff des *Kontextbezugs*, anstatt der Kontextabhängigkeit verwendet. Dies geschieht, um den Charakter der Kontextnutzung zu unterstreichen. Denn im Gegensatz zu einer existenziellen Abhängigkeit kann kontextbezogene Software z. T. auch ohne verfügbare Kontextinformationen genutzt werden. Meist muss sie sogar derart robust sein, schwankende Verfügbarkeit und Imperfektion (siehe Kap. 2.2.2) von Kontextinformationen zu tolerieren, indem die betroffenen Informationen entsprechend bewertet werden (z. B. ignoriert). Wenngleich solch eine Software dann gar nicht mehr oder nur teilweise in der Lage ist, sich an den Kontext anzupassen, so kann sie u. U. doch eine eingeschränkte Funktionalität erbringen. Dieses Verständnis von Kontextabhängigkeit kommt dem Englischen Original *Context Awareness* nahe – im Sinne von: sich des Kontextes bewusst sein bzw. den Kontext zu kennen.

A 3. *Kontext sollte dermaßen beschrieben werden, dass Kontextsenken in der Lage sind, die Qualität von Kontextinformationen zu bewerten, und ihre Nutzung daran anzupassen, d. h. auch, diese ggf. zu unterlassen.*

2.1.4. Fazit und weitere Begriffe

Diese Arbeit nimmt keine eigene Definition des Kontextbegriffes vor. Zusammengefasst werden jedoch folgende Merkmale als wesentlich angesehen:

- Kontext ist implizite Information, und muss deshalb – im Gegensatz zu expliziter Information – vor der Nutzung ermittelt werden. Alternativ kann Kontext auch entstehen, wenn ohnehin vorhandene Informationen für einen anderen als den ursprünglichen Zweck, d. h. sekundär, genutzt werden.
- Kontext entsteht durch die Aktivitäten eines Nutzers und anderer Entitäten (z. B. weitere Personen, IT-Systeme) seiner Umgebung.
- Kontext ist relevant/pertinent im Sinne des Information Retrieval, d. h. er beeinflußt aktuelle und zukünftige Handlungen von Nutzern bzw. Aktionen von IT-Systemen, weil er einen entsprechend verwertbaren Informationsgehalt besitzt.
- Kontext ist zweitrangig bedeutend, d. h. die Abhängigkeit der Nutzer und ihrer Systeme von Kontext ist geringer, als die Abhängigkeit von Anwendungsdaten (z. B. Eingaben, Dokumenten, etc.).
- Kontext kann wesentlich häufiger schwankender Verfügbarkeit und Qualität unterworfen sein (siehe auch Kap. 2.2), als dies bei Anwendungsdaten der Fall ist.

Zusätzlich zum Begriff des Kontexts werden im Rahmen dieser Arbeit die Begriffe *Kontextinformation* und *Kontexttyp* verwendet. Ihre Definitionen gelten analog zu [Spr04] (dort als Kontextwert und Kontexttyp bezeichnet), und werden um den Begriff der *Kontextvariable* [Gos06] ergänzt:

- Eine *Kontextvariable* bezeichnet einen einzelnen, eigenständigen Teil des gesamten Kontexts, welcher im Laufe der Zeit unterschiedliche Werte/Zustände annehmen kann.
- Eine *Kontextinformation* repräsentiert den Zustand, welchen eine Kontextvariable zu einem konkreten Zeitpunkt und an einem konkreten Ort besitzt.
- Der *Kontexttyp* repräsentiert den semantischen Typ einer Kontextvariable (und somit auch ihrer Kontextinformationen) im Sinne eines Typsystems, welcher eine konkrete Semantik sowie einen festgelegten Datentyp besitzt. So kann

beispielsweise der Typ Temperatur, als Teil des Systems physikalischer Größen definiert sein, welcher eine Temperatur repräsentiert, die in Grad Celsius angegeben wird, und an einem konkreten Punkt im dreidimensionalen Raum zu einem konkreten Zeitpunkt herrscht. Die Speicherung/Übertragung einer solchen Information könnte in Form eines 64-Bit-Fließkommawertes spezifiziert sein.

2.2. Eigenschaften von Kontext

Weil der Kontextdienst kontextbezogenen Anwendungen die erforderlichen Kontextinformationen zur Verfügung stellen soll, haben deren grundsätzliche Eigenschaften (inkl. Entstehung und Nutzung) großen Einfluss auf seine Realisierung. Deshalb sind diese Aspekte Gegenstand des folgenden Kapitels.

2.2.1. Dynamik

Eine hohe Dynamik [Dey00] ist einer Vielzahl verschiedener Kontexttypen eigen. Zuerst jenen, die anhand von Messungen physikalischer Größen ermittelt werden. Denn die ihnen zugrunde liegenden natürlichen Vorgänge sind nicht statisch (z. B. Außentemperatur im Tages- oder Jahresgang).

Weil gemessene Informationen oft die Basis zur Ableitung abstrakter Informationen (z. B. Tages- oder Jahreszeit) darstellen, überträgt sich ihre Dynamik transitiv auf die abgeleiteten Informationen, wenngleich die Dynamik aufgrund der Abstraktion häufig reduziert wird. Schwankt beispielsweise die Außentemperatur an einem Sommertag in Deutschland im Tagesgang zwischen 18°C und 36°C im Schatten, so bedeutet dies eine mittlere Änderungsrate der Temperatur um 1°K alle 40 Minuten. Diese Dynamik wird wesentlich abgeschwächt, wenn eine Abstraktion der objektiven Außentemperatur z. B. durch Anwendung von Fuzzylogik [Zad87] hin zu einer subjektiv empfundenen Temperatur statt findet. Bei einer Abbildung auf vier Werte anstatt der in Abb. 2.2 auf der nächsten Seite dargestellten vierzig Werte, finden Änderungen im Mittel nur noch aller sechs Stunden statt.

Selbst unter der (für diese Arbeit) unzulässigen Annahme lokal statischer Umgebungseigenschaften, entsteht eine Dynamik von Kontextinformationen durch die Bewegung des Nutzers innerhalb der Umgebung, weil nicht von homogenen Umgebungseigenschaften im Raum ausgegangen werden kann. So ändern sich z. B. Temperatur und Lichtintensität in der Umgebung eines beobachteten Nutzers, wenn er ein Gebäude betritt, selbst wenn diese Eigenschaften an den dabei passierten Punkten für hinreichende Zeit konstant bleiben.

A 4. *Der Zugriff auf Kontext muss zeitnah erfolgen, sodass die bereitgestellten Kontextinformationen aktuell sind, und die zeitnahe Anpassung daran möglich ist.*

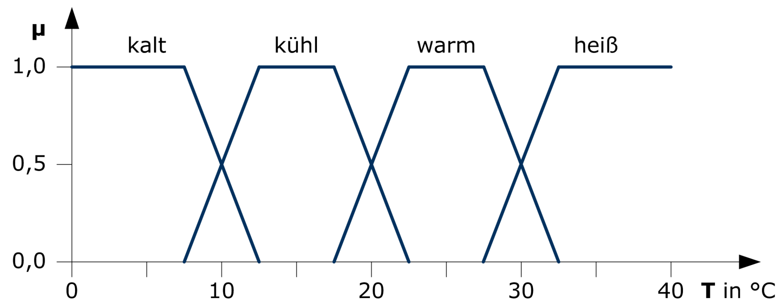

Abbildung 2.2.: Exemplarische Abstraktion mittels Fuzzylogik

Des weiteren ist der Nutzer aufgrund seiner Mobilität an wechselnden Orten – und somit von wechselnden Geräten – beobachtbar, weil er mit ihnen explizit oder implizit interagiert*. Analog zur Heterogenität von Umgebungseigenschaften, kann bzgl. der lokalen Verfügbarkeit von Geräten (speziell Kontextquellen) nicht davon ausgegangen werden, dass flächendeckend die Möglichkeit besteht, überall dieselbe Menge von Kontexttypen erfassen zu können. So kann z. B. innerhalb eines Gebäudes die lückenlose Erfassung von Temperaturinformationen durch separate Sensoren in jedem Raum möglich sein, während dies außerhalb nur in ausgewählten Bereichen um drahtlose Temperatursensoren herum gelingt. Ähnliche Auswirkungen kann Mobilität auf die Qualität des erfassten Kontextes haben. So kann innerhalb von Gebäuden u. U. die Position des Nutzers mit einer Genauigkeit von wenigen Zentimetern ermittelt werden, wogegen die Lokalisierung im Freien eventuell nur mit einer Genauigkeit von mehreren Metern möglich ist.

A 5. *In einer Umgebung verfügbare Kontextquellen müssen jederzeit gefunden und überwacht werden, um Kontextsenken ein Maximum an Kontextquellen bereitstellen zu können. Das gilt besonders, wenn bisher genutzte Quellen nicht mehr verfügbar/erreichbar sind, und somit alternative Quellen für dieselben oder ähnliche Informationen benötigt werden.*

Dem Umstand der inhärent (siehe Kap. 2.2.2) begrenzten, zeitlichen und quantitativen Auflösung von Sensoren, kann im Rahmen dieser Arbeit nur insofern Rechnung getragen werden, dass diese qualitativen Eigenschaften der Kontextquellen beim Kontextzugriff berücksichtigt werden sollen (siehe /A 3/ auf Seite 12).

*Dabei ist es nebensächlich, ob die Interaktion des Nutzers mit den stationären, in die Umgebung eingebetteten Geräten direkt oder indirekt, d. h. über persönliche mobile Geräte, erfolgt.

2.2.2. Imperfektion

Eine weitere wesentliche Eigenschaft von Kontext zeigen Henricksen und Indulska [HI04] auf – die Imperfektion. Sie beziehen sich auf den Sachverhalt, dass Kontext eine Form von Informationen darstellt, welche nicht immer zuverlässig genutzt werden kann. Anhand der Betrachtung möglicher Kontextvariablen zeigen die Autorinnen vier Formen der Imperfektion auf. Folglich kann eine Kontextvariable folgende Merkmale zeigen:

- *unbekannt*, wenn keinerlei Informationen über sie verfügbar sind, d. h. ihr Wert nicht bestimmt werden kann;
- *mehrdeutig*, wenn mehrere unterschiedliche Werte für ein und dieselbe Variable bekannt sind;
- *ungenau*, wenn der ermittelte Wert zwar richtig ist, jedoch bzgl. seiner Auflösung/Genauigkeit eine unzulässige Näherung darstellt, und somit nicht den Anforderungen der Software entspricht; und/oder
- *fehlerhaft*, wenn der ermittelte Wert nicht mit dem tatsächlichen Wert übereinstimmt.

In [DH06] werden ebenfalls vier Formen der Imperfektion – dort Unsicherheit genannt – vorgestellt, welche aber nicht vollständig identisch mit den soeben genannten sind. Entsprechend ergeben sich zwei zusätzliche Merkmale für Kontextvariablen:

- *unvollständig*, weil nicht alle Kontexttypen durch Sensoren erfassbar sind. Dies gilt speziell für alle abstrakt definierten Kontexttypen (z. B. Stressniveau eines Nutzers).
- oft *mangelhaft beschrieben*, wenn die Zuverlässigkeit und andere Eigenschaften (z. B. Genauigkeit) der liefernden Kontextquellen (speziell Sensoren) nicht bekannt sind.

Unter Kenntnis der qualitativen Eigenschaften von Kontext (z. B. Ursprungszeit, -ort, Genauigkeit, Zuverlässigkeit) ist es u. U. möglich die Imperfektion zu behandeln (siehe /A 3/ auf Seite 12). Die Qualität ist primär durch die Entstehung des Kontexts bedingt, und entsprechend vom Typ der Kontextquellen abhängig (Kap. 2.3.1). Mangelnde Plausibilität von Kontext kann durch Kombination mit ggf. vorhandenen zusätzlichen Informationen auf das benötigte Maß gesteigert werden.

Die Entwicklung oder Realisierung von Verfahren zur Fusion, Approximation, Interpolation, etc. ist nicht im Fokus dieser Arbeit. Interessant sind lediglich Mechanismen, welche die Bereitstellung und den Zugriff auf die notwendigen Kontextinformationen und deren Beschreibungsdaten ermöglichen.

2.3. Kontextklassen

Im Laufe der bisherigen Diskussion wurden mehrfach unterschiedliche Klassen* von Kontextinformationen (oder -typen) genannt. Dabei wurde jedoch nicht auf deren spezifische Eigenschaften eingegangen. Dies soll in diesem Abschnitt nachgeholt werden.

2.3.1. Klassifikationen

Bei der Definition der nachfolgend vorgestellten Klassifikationen haben die Autoren unterschiedliche Ansätze gewählt und auch jeweils unterschiedliche Merkmale für die Klassifikation gewählt.

Inhaltlich

Eine der prominentesten und ältesten Klassifikationen berücksichtigt den inhaltlichen Ursprung der Kontextinformationen. Hier repräsentiert jede Kontextklasse einen spezifischen thematischen Ausschnitt des Kontextes. Schilit et. al. [SAW94] definieren drei Klassen (Tab. 2.1).

Tabelle 2.1.: Kontextklassen der inhaltlichen Klassifikation nach [SAW94]

Klasse	**Beispiele**
Physical Context	Beleuchtung, Geräusch, Temperatur, sogar Verkehrssituation
Computing Context	Netzwerkeigenschaften, elektronische Ressourcen (z. B. Drucker, Displays, benachbarte Rechner)
User Context	Nutzerprofil (z. B. Präferenzen, Ort/Position, benachbarte Personen, und soziale Situation)

Dey [Dey00] nimmt eine ähnliche Einteilung vor, benennt die Klassen aber mit Environment statt Context. Jedoch illustriert er seine Klasse User Environment – abweichend von Schilits User Context – beispielhaft durch den Ort des Nutzers, die Menge benachbarter Personen und seine soziale Situation. Somit wird der situative Charakter hervor gehoben. Chen und Kotz [CK00] hingegen messen der Zeit eine besondere Rolle bei, und heben sie aus dem physikalischen Kontext heraus in eine eigene Klasse. Dabei legen sie besonderen Wert auf die zeitliche Änderung der Werte einer Kontextvariablen im Sinne eines *Kontextverlaufs* (engl. context history). Diese Sonderrolle ist durchaus gerechtfertigt (siehe Kap. 2.3.2), eröffnet sie doch eine bisher unberücksichtigte Dimension in der Klassifikation – die Beeinflussung

*Auch mitunter als *Kontextdimensionen* bezeichnet [Gos06].

von Kontexttypen durch andere Kontexttypen – welche im späteren Verlauf noch getrennt betrachtet wird (siehe Kap. 2.3.2).

Bis heute werden inhaltliche Klassifikationen benutzt. Als aktueller Vertreter dieser Gruppe ist [Spr04] zu nennen, welcher die Klassen *physikalischer*, *technischer*, *persönlicher* und *situationsbezogener Kontext* definiert. Weil die vorgestellten inhaltlichen Klassifikationen eine vorwiegend aufzählende Charakterisierung der Klassen vornehmen, bleiben allgemeine Eigenschaften schwer erkennbar. Folglich können diese Klassifikationen nur wenig Aufschluss über die notwendige Behandlung der Kontextklassen durch den Kontextdienst geben.

Abstraktionsbezogen

Die abstraktionsbezogene Klassifikation unterscheidet Kontexttypen anhand ihres Abstraktionsniveaus und definiert zwei Klassen *nieder-* und *höherwertigen Kontext*, so z. B. in [Dar06]. Für diese beiden Begriffe existieren noch zahlreiche alternative Bezeichnungen: low-level context, primitive context, und basic context, sowie high(er)-level context und abstract context – natürlich inklusive ihrer deutschen Pendants. All diesen Klassifikationen ist gemein, dass niederwertige Kontextvariablen direkt erfasst werden können, wogegen höherwertige Kontextvariablen durch eine näher zu bestimmende Form der Ableitung aus einer oder mehreren Kontextvariablen abstrahiert werden müssen. Die so entstandenen höherwertigen Kontextvariablen können wiederum – ggf. zusammen mit anderen höher- oder niederwertigen Kontextinformationen – Grundlage weiterer Ableitungen sein. Derart können komplexe Abhängigkeitsgraphen entstehen, welche bei der systeminternen Verwaltung berücksichtigt werden müssen.

Ob ein Kontexttyp mehr oder weniger abstrakt ist, macht für die Suche nach entsprechenden Kontextinformationen keinen Unterschied. Jedoch kann sich die inhaltliche Abhängigkeit abstrakter Kontextinformationen von ihren zugrunde liegenden Kontextinformationen auf ihre Verfügbarkeit auswirken. Die Transitivität dieser Abhängigkeit wirkt sich auch auf die Dynamik der Kontextvariablen aus (Kap. 2.2.1) und ist ggf. vom Kontextdienst zu berücksichtigen.

Quellenspezifisch

Die letzte Klassifikation [HI04], welche in dieser Arbeit betrachtet werden soll, ist von der inhaltlichen Klassifikation beeinflusst. Sie orientiert sich an den Quellen des Kontexts – Sensoren, Nutzern und Ableitungsverfahren – und legt zusätzlich Wert auf die Berücksichtigung der Persistenz des Kontexts. Folglich wird zwischen statischen und dynamischen Informationen unterschieden, die jeweils weiter unterteilt werden, sodass effektiv vier Klassen definiert sind (siehe Tab. 2.2 auf der nächsten Seite).

Tabelle 2.2.: Kontextklassen der quellenspezifischen Klassifikation nach [HI04]

Klasse	Beschreibung
Statischer Kontext	sämtliche Informationen, welche sich – ungeachtet ihres Quellentyps – nie ändern.
Profilierter Kontext	entsteht durch die Erfassung nutzerspezifischer Informationen. Die Erfassung kann dabei entweder direkt durch den Nutzer erfolgen, d. h. per Spezifikation, oder mittels Beobachtung des Nutzers. Zur Ableitung abstrakter Kontextinformationen aus den beobachteten primitiven Informationen werden häufig Ansätze aus dem Bereich der künstlichen Intelligenz bzw. des maschinellen Lernens eingesetzt – Neuronale Netze, Bayessche Netze, Hidden-Markov Modelle, etc. Obwohl der Beobachtungsgegenstand – der Nutzer – nicht statisch ist, ändert er sein Verhalten jedoch nur sehr langsam, sodass profilierter Kontext von den Autorinnen ebenfalls als statisch angesehen wird. Profilierter Kontext kann inaktuell und unvollständig sein.
Gemessener Kontext	wird mittels Sensoren erfasst. Er ändert sich häufig und kann ungenau oder veraltet sein.
Abgeleiteter Kontext	ist ebenfalls dynamisch, weil alle aus statischem Kontext ableitbaren Informationen schon a priori abgeleitet werden können und somit wieder statischen Kontext darstellen. Ihre Ableitung zur Laufzeit wäre unnötig, und würde die Effizienz des Systems verschlechtern. Somit bleibt als Basis für abgeleiteten Kontext nur dynamischer Kontext – , d. h. es wird nur dynamischer Kontext aus dynamischem Kontext abgeleitet.

Diese Klassifikation birgt einen Widerspruch, indem sie profilierten Kontext als statisch ansieht, aber gleichzeitig unterstellt, dass er inaktuell sein kann – statische Informationen können jedoch nicht inaktuell sein bzw. werden. Aus der quellenspezifischen Klassifikation kann gefolgert werden, dass die Bereitstellung statischen und profilierten Kontexts zeitunkritisch ist, und deshalb nicht gesondert vom Kontextdienst behandelt werden muss*. Alle anderen Klassen stellen dynamischen Kontext dar, welcher zeitnah auffindbar und zugreifbar verwaltet werden muss.

*Probleme beim mobilen Zugriff auf statischen/profilierten Kontext könnten z. B. durch Verteilung von redundanten Kopien wesentlich gemildert werden.

2.3.2. Spezielle Kontexttypen

Neben den zuvor beschriebenen Kontextklassen gibt es spezielle Kontexttypen, welche zwar diesen Klassen zugeordnet werden können, denen jedoch eine besondere Rolle zukommt. Sie können einerseits Relevanzfaktoren darstellen (siehe Kap. 2.1.2), andererseits die Ableitung vieler anderer Kontextinformationen ermöglichen. Sie sind somit in der Rolle eines primären Kontextes. Nicht, weil sie ggf. gemessen bzw. mit einfachen Mitteln erfasst werden können, sondern weil sie im übertragenen Sinne von primärer, d. h. herausragender, Bedeutung sind.

Schon Chen und Kotz [CK00] räumen der *Zeit* besondere Bedeutung ein. Schmidt und Gellersen [SG01] ergänzen sie um den *Ort*. Dey [Dey00] schließlich hebt die *Identität des Nutzers* hervor. Diese drei Kontexttypen sind von zentraler Bedeutung und sollten entsprechend berücksichtigt werden.

2.4. Kontextbezug und kontextbezogene Software

Die folgende Betrachtung kontextbezogener Systeme ist in zwei Aspekte unterteilt – die Bereitstellung der für ihren Betrieb erforderlichen Kontextinformationen (siehe nächstes Unterkapitel), und deren Nutzung (siehe Kap. 2.4.2). Aufgrund der Vielfalt und Menge von Kontextquellen besteht dabei zunächst das Kernproblem in der Spezifikation ihrer verteilten Verwaltung dieser ad hoc verfügbaren und genutzten Ressourcen.

2.4.1. Kontextbereitstellung

Die Bereitstellung bzw. Erfassung von Kontext erfolgt durch die Kontextquellen, welche unterschiedlicher Gestalt sein können. Sie werden entsprechend in den nächsten Abschnitten kurz vorgestellt*.

Sensoren

Sensoren können in Form von Hardware- oder Softwaresensoren auftreten. Erstere gewinnen ihre Informationen durch Messung von Umweltgrößen (z. B. physikalisch oder chemisch), wobei eine Zeit- und Wertdiskretisierung statt findet. Entsprechende Ungenauigkeiten sind folglich allen Sensoren inhärent. Softwaresensoren gewinnen Informationen meist durch Anwendung von Metriken auf Parameter und Indikatoren† von Systemkomponenten. So kann ein Softwaresensor beispielsweise Datenrate,

*Die Kontextquellen werden in Anlehnung an [HI04] in drei Klassen eingeteilt.

†Im einfachsten Fall entsprechen die Informationen den Ein- und Ausgaben von Systemaufrufen, oder sie spiegeln den Systemzustand wieder.

Jitter und Verzögerung einer Netzwerkverbindung ermitteln. Überwachen Softwaresensoren Teile der Benutzungsschnittstelle, so ist mit ihrer Hilfe auch die Beobachtung des Benutzers möglich. In Abhängigkeit von den beobachteten Vorgängen/-Systemkomponenten und der Mobilität des Benutzers bzw. seiner Geräte, sind von Sensoren gelieferte Informationen mehr oder weniger dynamisch. Sensoren können in großer Zahl* in einer Umgebung auftreten, welche sich über eine großen räumlichen Bereich erstreckt [MCP+02].

Es herrscht große Heterogenität im Bereich der Sensoren. Dies lässt sich einerseits auf die Vielfalt erfassbarer Informationstypen zurückführen, andererseits aber auch auf die unterschiedlichen Anforderungen denen die Sensoren im Einsatz genügen müssen. So ist es ohne weiteres möglich, das zwei Sensoren denselben Informationstyp – z. B. Temperatur – liefern, jedoch mit stark unterschiedlicher Qualität – z. B. hinsichtlich Messbereich, Genauigkeit, Aktualität, Präzision, etc. Nicht zuletzt gibt es technische Freiheiten bei der Realisierung von Sensoren. So gibt es sowohl Sensoren, welche aktiv abgefragt werden müssen, um deren Informationen zu erhalten, als auch solche, die selbstständig ihre Informationen regelmäßig oder bei Änderung übertragen. All diese Eigenschaften moderner Sensoren werden in Form von Parametern repräsentiert, welche für den jeweiligen Einsatzzweck konfiguriert werden müssen. Problematisch sind dabei nicht nur Vielfalt und Heterogenität der möglichen Parameter, sondern auch deren Zusammenhänge. So ist bei Sensoren mit konfigurierbarem Messbereich dessen Größe oft umgekehrt proportional zur Auflösung der gelieferten Messwerte†. Nicht immer sind diese Zusammenhänge dem Nutzer eines Sensors bekannt.

Ein weitere wesentliche Eigenschaft von Soft- und Hardwaresensoren liegt in ihrer schwankenden Verfügbarkeit. Wobei die Schwankung nicht nur durch Verbindungsunterbrechungen zu entfernt genutzten Sensoren (z. B. per Web Service oder Bluetooth) verursacht wird. Vielmehr kann sie auch lokal induziert sein: durch Plug-N-Play von Hardware, sowie De-/Installation von Software. Beides kann sowohl auf Initiative des Nutzers als auch durch Automatismen des Systems ausgelöst werden. Eine mögliche Motivation solcher Maßnahmen ist die Einsparung von Betriebsmitteln (z. B. Energie, Rechenzeit, Speicher) – speziell bei stark ressourcenbeschränkten, mobilen und integrierten, Geräten.

*Siehe auch Begriffe *Smart Dust* und *Wireless Sensor Network (WSN)* im Bereich des *Pervasive Computing*.

†Weil die Anzahl der Diskretisierungsintervalle meist gleich bleibt, geht eine Vergrößerung des Messbereiches auch mit einer Verringerung der Auflösung einher. Dabei beschreibt eine hohe Auflösung die Möglichkeit der Unterscheidung von zwei Messwerten mit sehr geringer Differenz, während bei einer niedrigen Auflösung die Differenz der Messwerte groß ist.

Datenbanken

Im Gegensatz zu Sensoren liefern Datenbanken keine dynamischen Informationen, sondern nur statische. Datenbanken sind klassische C/S-Systeme. Wegen des meist zentralen Charakters von Datenbanken existiert oft nur eine einzige Instanz dieser Kontextquellenklasse für einen konkreten Kontexttyp, was ihre Verwaltung wesentlich vereinfacht. In Verbindung mit der Lieferung statischer Informationen macht dieser Umstand Datenbanken für eine Betrachtung im Rahmen dieser Arbeit weniger interessant, sodass sie nachfolgend nicht näher betrachtet und unterstützt werden.

Ableitungsschemata

Vertreter der Klasse der Ableitungsschemata wenden unterschiedlichste Verfahren an, um aus Informationen weitere Informationen abzuleiten. Sie werden häufig in Form von Software realisiert. Als Verfahren kommen dabei Ansätze aus der Mathematik genauso zum Einsatz, wie solche aus den Bereichen Logik, Künstliche Intelligenz, maschinelles Lernen, usw. Ableitungen werden einerseits vorgenommen, um den Abstraktionsgrad von Informationen zu erhöhen, andererseits aber auch, um deren Qualität zu verbessern.

Weil die Ableitung aber von Kontextinformationen abhängig ist, welche dynamischen Charakter haben, sind die von ihr produzierten Informationen ebenfalls dynamisch. Jedoch spiegelt die Dynamik nicht nur die Dynamik der Eingangsinformationen, sondern auch den Ableitungsprozess wieder. Da dieser häufig eine Abstraktion vornimmt, reduziert sich die Dynamik z. T. nennenswert (siehe Kap. 2.2.1).

Die Ableitungsschemata sind von Kontextinformationen abhängig, welche von anderen Kontextquellen geliefert werden. Sind diese schwankender Verfügbarkeit unterworfen, dann wirkt sie sich ebenfalls auf die Verfügbarkeit der von den Ableitungsschemata produzierten Informationen aus.

Fazit

Die Menge potentiell anzutreffender Kontextquellen ist sehr heterogen. Dafür gibt es mehrere Gründe. Einerseits existieren ohnehin unterschiedliche Klassen von Kontextquellen. Andererseits sind diese Klassen selbst intern nicht homogen. Das ist sowohl darin begründet, dass diese Quellen trotz ihrer Zugehörigkeit zu einer konkreten Klasse (z. B. Sensoren) unterschiedliche Informationen bereitstellen (z. B. Temperatur und Helligkeit), welche ggf. unterschiedlich erfasst werden müssen. Weiterhin müssen Quellen ungeachtet ihrer Klasse unterschiedliche, anwendungsspezifische Anforderungen erfüllen, und sind deshalb unterschiedlich ausgelegt – z. B. hinsichtlich Wertebereich, Auflösung, Datenformat, Einheiten, etc. Nicht zuletzt gibt es bei der Realisierung von Quellen unterschiedliche technische Mittel zur Erreichung desselben Zieles.

A 6. *Die große Heterogenität der Kontextquellen ist vor den Kontextsenken zu verbergen, sodass der einheitliche Zugriff auf unterschiedlichste Kontextquellen möglich ist.*

Es ist festzustellen, dass ein signifikanter Teil von Informationsquellen nicht zum Zweck der Kontexterfassung in eine Umgebung eingebracht wird, sondern dort vorhanden ist, weil er einem anderen, primären Zweck dient – z. B. Überwachung von technischen Vorgängen. Somit ist die Gewinnung von Kontextinformationen nur eine sekundäre Nutzung der Informationsquellen in einer „zweckentfremdeten“ Form. Der Begriff *Kontextquelle* kann folglich auch als Rollenbezeichnung von Informationsquellen verstanden werden. Bei derart sekundär genutzten Einrichtungen kann nicht davon ausgegangen werden, dass sie aufgrund von externen Anforderungen, die aus der Kontextgewinnung erwachsen, geändert werden. Darüber hinaus ist eine gezielte, zweckgebundene Einbringung von Kontextquellen in die Umgebung in vielen Situation zu aufwendig oder zu teuer. Die Kontextgewinnung muss folglich angepasst an die vorhandenen Informationsquellen erfolgen, nicht umgekehrt.

A 7. *Die Integration der Kontextquellen in das Gesamtsystem hat derart zu erfolgen, dass die dafür notwendigen Maßnahmen geringen Aufwand mit sich bringen, und nur geringe Wechselwirkungen mit den Kontextquellen zur Folge haben.*

Aufgrund des integrativen Charakters des Kontextdienstes, sind unerwünschte Wechselwirkungen zwischen den Quellen und/oder Kontextsenken problematisch.

A 8. *Auswirkungen des Verhaltens (speziell des Fehlverhaltens) von Kontextquellen auf andere Kontextquellen, den Kontextdienst und die Kontextsenken sind nach Möglichkeit zu verhindern.*

Der dynamischen Verfügbarkeit von Quellen des lokalen Knotens ist Rechnung zu tragen (/A 9/), um Fehler der Senken beim Kontextzugriff zu vermeiden oder zu behandeln (/A 10/), d. h. wenn nötig auch alternative Quellen einzubinden. Das gilt auch für die Quellen entfernter Knoten (/A 11/).

A 9. *Die Menge der knotenlokalen Kontextquellen ist dauerhaft zu überwachen, um zu jedem Zeitpunkt deren Verfügbarkeit feststellen zu können.*

A 10. *Ist eine Quelle nicht mehr verfügbar, so sind alternative Quellen zu bestimmen und ggf. zu verwenden. Dies hat automatisch und für die Senke transparent zu erfolgen. Nur im Falle fehlender Alternativen sind eventuelle Kontextsenken zu benachrichtigen (siehe auch /A 3/ auf Seite 12).*

Unabhängig dessen, ob sie ursprünglich oder alternativ genutzt sind, können sich die von einer Senke genutzten Quellen auf entfernten* Knoten befinden, sodass sich

*Bezogen auf die Position der Senke, d. h. den lokalen Knoten.

die Erreichbarkeit des Knotens auf die Verfügbarkeit seiner lokalen Quellen für andere Knoten auswirkt.

A 11. *Die Erreichbarkeit entfernter Knoten, sowie die Verfügbarkeit ihrer lokalen Quellen ist ebenfalls zu überwachen.*

2.4.2. Kontextnutzung

Ausgangspunkt für die Betrachtung der Kontextnutzung soll eine Definition von Dey sein [Dey01], welcher kontextabhängige bzw. kontextbezogene Anwendungen wie folgt beschreibt:

> "A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task."

Erneut kommt der Relevanz signifikante Bedeutung zu, weil definiert wird, dass kontextbezogene Anwendungen relevante Informationen bereitstellen, oder relevante Dienste anbieten. In ihrer Überblicksarbeit untersuchen Chen und Kotz [CK00] eine Vielzahl vorangegangener Arbeiten und deren Definitionen von unterschiedlichen Klassen bzw. Eigenschaften kontextbezogener Anwendungen – u. a. Dey's Arbeiten. Jedoch fassen sie diese Definitionen so weit zusammen, dass sie für die Extraktion von Anforderungen in dieser Arbeit zu allgemein gefasst sind. Deshalb ist eine Betrachtung der einzelnen Arbeiten unumgänglich. Weil die untersuchten Arbeiten sich vorwiegend auf die Betrachtung der Verwendung der Kontextinformationen konzentrieren, soll zuerst im folgenden Unterkapitel dieser Aspekt dargestellt, und zwei wesentliche Formen des Zugriffs daraus abgeleitet werden. Danach werden mögliche Charakteristika des Kontextzugriffes hinsichtlich Raum, Zeit und Personen vorgestellt.

Zugriffsformen

Die Arbeit von Schilit et. al. [SAW94] beschreibt vier unterschiedliche Klassen kontextbezogener Anwendungen (Tab. 2.3 auf der nächsten Seite).

Die ersten drei Anwendungsklassen erfordern eine dauerhafte Überwachung der Umgebung, d. h. dauerhafte Ermittlung von Kontextinformationen. Andernfalls wäre es unmöglich:

- die Menge benachbarter Objekte zu ermitteln, welche bei der Präsentation hervorgehoben werden sollen. Dies gilt besonders, wenn der Nutzer mobil ist, d. h. die Menge benachbarter Objekte jederzeit Veränderungen unterworfen sein kann.

Tabelle 2.3.: Klassen kontextbezogener Anwendungen nach [SAW94]

Anwendungsklasse	Verhalten
Proximate Selection	Hervorhebung benachbarter Objekte, um die Auswahl durch den Nutzer zu erleichtern
Automatic Contextual Reconfiguration	Hinzufügen, Entfernen, Verbinden von Komponenten, d. h. Re-Strukturierung der Anwendung
Context-triggered Actions	Wenn-Dann-Regeln zur Steuerung der Adaption von Anwendungen in Abhängigkeit von Kontext
Contextual Information and Commands	in Abhängigkeit vom jeweils geltenden Kontext werden Informationen angepasst präsentiert oder Befehle angepasst ausgeführt.

- den Bedarf zur Rekonfiguration der Anwendung, und die dafür eventuell notwendigen Parameter festzustellen.
- die Möglichkeit/den Bedarf zur Ausführung adaptiver Aktionen festzustellen.

A 12. *Zur dauerhaften Überwachung ihrer Umgebung müssen Kontextsenken asynchron auf Kontextinformationen zugreifen können, d. h. sie müssen durch Benachrichtigungen über den aktuellen Zustand der Kontextvariablen unterrichtet werden.*

Asynchroner Informationszugriff wird häufig in Form des Publish/Subscribe- oder Push-Prinzips realisiert. Entsprechend abonnieren (engl. to subscribe) die Kontextsenken Meldungen über konkrete Kontextvariablen, welche bei deren Änderungen asynchron vom System erzeugt und an die Kontextsenken ausgeliefert werden.

Im Gegensatz dazu greift die vierte Anwendungsklasse lediglich im Moment der Darstellung der Informationen bzw. bei Auslösung des Kommandos auf die aktuellen Kontextinformationen zu, und geht davon aus, dass entweder der Kontext sich während der Darstellung bzw. Ausführung nicht ändert, oder Änderungen keine Auswirkungen auf die Darstellung bzw. Ausführung haben. Die Kontextsenken dieser Klasse stellen nach dem Request/Response- oder Pull-Prinzip Anfragen an das System, welche synchron mit Antworten bedient werden.

A 13. *Um die aktuellen Charakteristika ihrer Umgebung ermitteln zu können, müssen Kontextsenken synchron auf ihre Kontextinformationen zugreifen können. Nur wenn diese Informationen zeitnah zur Anfrage ermittelt werden, kann eine Basis für die konsistente, kontextbezogene Darstellung von Informationen oder Ausführung von Aktionen ermöglicht werden.*

Nicht nur beim Zugriff des Kontextdienstes auf die Kontextquellen sind seitens Ersterem in den Kontextquellen auftretende Fehler zu behandeln, sodass sie sich

nicht auf die übrigen Kontextquellen oder den Kontextdienst auswirken. Auch bei der Auslieferung der Kontextinformationen können innerhalb der Kontextsenken Fehler auftreten, deren Auswirkungen auf den Kontextdienst, sowie andere Kontextsenken und die Kontextquellen zu vermeiden sind.

A 14. *Bei der Auslieferung der Kontextinformationen innerhalb der Kontextsenken auftretende Fehler sind durch den Kontextdienst zu behandeln, sodass ihre Auswirkungen auf die betreffende Kontextsenke isoliert bleiben.*

2.4.3. Dimensionen des Kontextzugriffs

Es wurde bereits ersichtlich, dass beim Zugriff auf Kontext verschiedene (Relevanz-)Faktoren eine Rolle spielen können. Im Folgenden werden exemplarisch die drei Dimensionen Zeit, Raum und Nutzer/Person betrachtet. Darüber hinaus können jedoch viele weitere – oft domänen- oder anwendungsspezifische – Dimensionen existieren, deren Berücksichtigung beim Zugriff wünschenswert ist.

Zeitliche Dimension Die Heterogenität von Kontextquellen kann dazu führen, dass Quellen, welche den gleichen Kontexttyp liefern, dies mit stark abweichender Qualität realisieren. An dieser Stelle soll lediglich auf die zeitliche Auflösung eingegangen werden. Die Anforderungen von Kontextsenken hinsichtlich der Zeitauflösung von Kontextinformationen sind stark unterschiedlich, je nach Anwendungsdomäne und ggf. abhängig vom jeweiligen Nutzer. Folglich müssen Anwendungen ihre zeitlichen Anforderungen spezifizieren können. Andernfalls wäre es möglich, dass Kontextinformationen ausgeliefert werden, welche nicht die nötige Zeitauflösung besitzen. Darauf basierende Anwendungen wären nur begrenzt bzw. gar nicht nutzbar. Ebenso könnten – speziell im asynchronen Fall – Informationen unnötig oft an die Informationssenken geliefert werden, obwohl sie nicht so schnell verarbeiten werden können. Zusätzlich führt dieses Verhalten zu unnötiger hoher Systemlast hinsichtlich Rechenzeit und Übertragungskapazitäten.

Die Aktualität soll jedoch lediglich als Kriterium beim Zugriff auf Kontextinformationen verwendet werden. Es ist nicht Ziel des Systems, Kontextinformationen zu analysieren, um deren Änderungshäufigkeit zu ermitteln, welche ggf. sogar von der durch unterschiedliche Kontextsenken benötigten Wertdiskretisierung bzw. -auflösung abhängig wäre.

Räumliche Dimension Neben zeitlichen Einschränkungen kann es – speziell bei mobilen Anwendungen – auch räumliche Einschränkungen geben. Die Anwendungsklasse *Proximate Selection* zeigt, dass das System bzw. sein Nutzer eventuell nur Informationen als relevant betrachtet, deren Ursprungsort eine bestimmte Entfernung um die aktuelle Position nicht überschreitet. So könnte ein intelligentes Kfz-

Navigationssystem die aktuell verfügbaren Parkplatzkapazitäten in einem bestimmten Radius um das Fahrzeug herum ermitteln. Im Zuge proaktiver adaptiver Maßnahmen ist es ebenfalls vorstellbar, dass der als relevant betrachtete Bereich nicht bezogen auf die aktuelle Position definiert wird, sondern mit Bezug zu einer abweichenden, zukünftigen Position.

Wie das obige kurze Beispiel zeigt, hat der räumliche Zugriff eine Vielzahl möglicher Parameter. Wird eine Position anhand von logischen oder physischen Koordinaten definiert? Welche Form hat der Bereich? Selbst wenn Informationen benötigt werden, welche von einer konkreten Position stammen müssen, kann diese Position nur mit einer bestimmten Genauigkeit definiert werden. Folglich muss die Position/-der Bereich zusätzlich durch ein Toleranzmaß präzisiert bzw. fuzzifiziert werden. Wird der Ursprung von Informationen ohnehin auf einen mehr oder weniger großen Bereich beschränkt, so kann dieser durch die unterschiedlichsten geometrischen Formen begrenzt werden. Weiterhin kann er auf Basis des zwei- oder dreidimensionalen Raumes definiert sein. Darüber hinaus kann der Bereich relativ zur aktuellen Position einer Kontextsenke adressiert werden, oder eine absolute Position darstellen.

Nutzerdimension Neben Zeit und Raum ist ebenfalls Unterstützung für den letzten Relevanzfaktor – den menschlichen Nutzer – wünschenswert. Natürlich ist der Nutzer aufgrund des subjektiven Charakters von Kontext von zentraler Bedeutung beim Zugriff auf Kontextinformationen. Darüber hinaus ist es sinnvoll Zugriffe auf die persönlichen bzw. personenbezogenen Informationen eines Nutzers zu kontrollieren. Nur so kann die grundlegende Privatsphäre gesichert werden. Speziell für kollaborative Anwendungen, welche die Interaktion zwischen ihren menschlichen Nutzern fördern sollen, stellt dies eine wesentliche Forderung dar. Nichts wäre schlimmer, als persönliche Informationen in die Hände unberechtigter Nutzer oder Systeme gelangen zu lassen.

Die Zugriffskontrolle auf persönliche Informationen ist schwer zu realisieren, weil Informationen z. T. auch ohne Mithilfe des Nutzers feststellbar sind. So sind manche Systeme (z. B. Kameras mit Mustererkennung) in der Lage die Identität und somit eventuell auch den aktuellen Aufenthaltsort einer Person selbstständig festzustellen. Weiterhin entstehen viele persönliche Daten erst durch die Verknüpfung von Informationen mit der konkreten Identität einer Person. Es ist also wichtig, dass die Systeme derart gestaltet sind, dass die Mitwirkung des Nutzers oder seiner Geräte bei der Identifikation erforderlich ist, um so die Zustimmung zur Verarbeitung seiner persönlichen Daten zu erhalten. Eine umfassende Betrachtung der notwendigen Konstruktionsprinzipien für solche Systeme geht weit über den Rahmen dieser Arbeit hinaus.

2.5. Anforderungskatalog

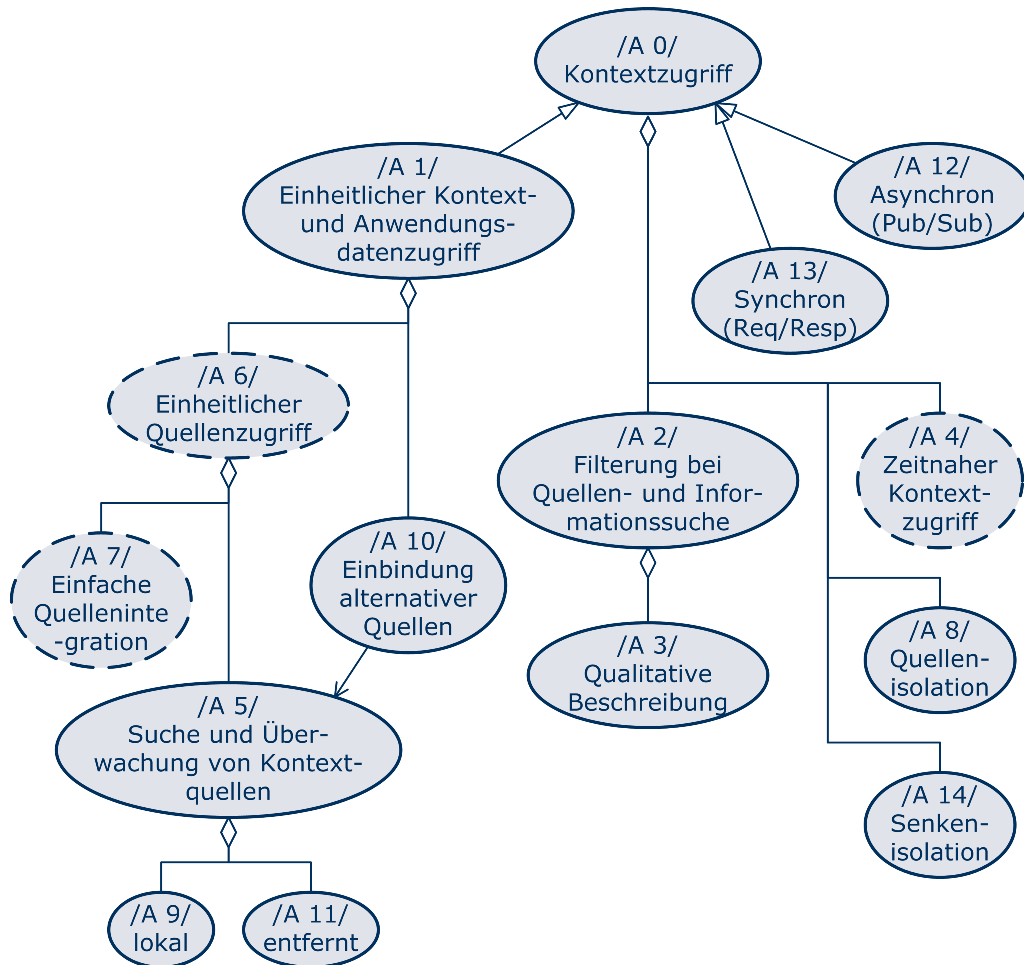

Abbildung 2.3.: Zusammenhänge der Anforderungen, welche durch die Eigenschaften von Kontext, sowie seiner Bereitstellung und Nutzung induziert sind

Die vorangegangene Untersuchung der Charakteristika von Kontext, sowie seiner Bereitstellung und Nutzung hat eine Vielzahl grundlegender Anforderungen aufgezeigt (siehe Abb. 2.3), welche in der Mehrzahl funktionaler Art sind (/A 1/–/A 3/, /A 5/, /A 8/–/A 14/). Funktionale Anforderungen sind mit geschlossenem Rand dargestellt – nicht-funktionale mit durchbrochenem. Geschlossene Pfeile repräsentieren eine Spezialisierungsbeziehung, welche von der spezialisierten zur allgemeineren

Anforderung verläuft. Ein offener Pfeil stellt eine Nutzungsbeziehung dar, und verweist auf die genutzte Anforderung bzw. ihre Realisierung. Die Raute markiert das Kompositum einer Kompositionsbeziehung.

Die zentrale Anforderung /A 0/ ist der Kontextzugriff für adaptive Software. Sie ist zugleich die grundlegende Motivation für diese Arbeit, und wurde im vorangegangenen Text bereits mehrfach implizit definiert, ohne je explizit aufgeführt zu werden. /A 0/ besitzt drei Facetten – /A 1/, /A 12/ und /A 13/ – welche Spezialisierungen dieser allgemeinen Anforderung darstellen.

Der einheitliche Zugriff auf Anwendungs- und Kontextinformationen /A 1/ setzt zuerst den einheitlichen Zugriff auf alle Informationsquellen voraus /A 6/, welcher die Transparenz der Heterogenität der Quellen zum Ziel hat. Um die Informationen der Quellen den Anwendungen zur Verfügung stellen zu können, sind diese zuerst zu suchen und zu überwachen /A 5/. Das gilt sowohl für die Quellen des lokalen Gerätes /A 9/, als auch für diejenigen, welche sich auf entfernten Geräten befinden /A 11/. Für die Integration der Quellen für deren einheitlichen Zugriff sollte wenig Aufwand seitens der Entwickler notwendig sein /A 7/. Um die Unterschiede zwischen Anwendungs- und Kontextquellen verbergen zu können, muss vor allem die dynamische Verfügbarkeit der Kontextquellen gekapselt und behandelt werden. Im Zuge dessen sind ggf. alternative Kontextquellen zu selektieren /A 10/. Diese Funktion basiert ihrerseits auf der Überwachung und Suche der Kontextquellen /A 5/.

Die Anforderungen /A 12/ und /A 13/ repräsentieren die beiden Formen des Kontextzugriffs – asynchron und synchron – welche je nach Art der Anwendungen notwendig sind.

Unabhängig von der Zugriffsform sind die genutzten Kontextquellen und die von ihnen bezogenen Kontextinformationen auf die benötigte Menge zu reduzieren /A 2/, dass die Funktion der kontextbezogenen Anwendungen auf die Nutzung der Kontextinformationen konzentriert werden kann. Weil die Anwendungen nicht nur Anforderungen hinsichtlich des semantischen Typs der benötigten Informationen haben, sind die Kontextquellen und ihre Informationen entsprechend mit qualitativen Merkmalen zu beschreiben. Diese Beschreibung ist vom System zu verarbeiten und auch an die Anwendungen weiter zu leiten /A 3/. Nur so sind die Anwendungen in der Lage, die Nicht-Verfügbarkeit von Informationen zu erkennen und zu behandeln.

Weiterhin muss der Kontextzugriff zeitnah erfolgen /A 4/. Andernfalls könnte die Volatilität der Kontextinformationen zur Nutzung veralteter Informationen durch die Anwendungen führen. Aufgrund der Vielzahl von Kontextquellen und der parallelen Nutzung ihrer Kontextinformationen durch mehrere Anwendungen sind Fehler in den Quellen und den Senken auf diese zu beschränken, dass die übrigen Komponenten vor eventuellen Effekten isoliert werden – /A 8/ und /A 14/.

2.5.1. Einschätzung

Aufgrund ihres allgemeinen Charakters sind die Anforderungen /A 1/ und /A 2/ am problematischsten zu beurteilen. Einerseits hat die theoretische Betrachtung gezeigt, dass die Unterschiede zwischen Kontext und klassischen Anwendungsdaten eine einheitliche Nutzung beider Informationsklassen nur in äußerst eingeschränkten Szenarien gestatten. In den meisten Fällen ist aufgrund der Dynamik und Imperfektion von Kontext eine besondere Behandlung erforderlich, welche für Anwendungsdaten nicht notwendig ist, und somit den transparenten Zugriff auf Kontextinformationen unrealistisch erscheinen lässt. Andererseits ist die Filterung von Kontextinformationen anhand von Kriterien notwendig, welche entweder rein domänenspezifisch sind (z. B. Nutzer, Fahrzeug, Maschine), oder die domänenspezifische Formen haben – z. B. Koordinatensysteme für verschiedene Einsatzzwecke. Die Erfüllung dieser Anforderungen ist folglich einsatzspezifisch und komplex, sodass vorwiegend spezifische und aufwändige Lösungen zu erwarten sind. Allgemein gültige, generische Lösungen sind vor diesem Hintergrund wenig wahrscheinlich.

3. Anwendungsszenario

Um die theoretisch begründeten Anforderungen aus dem vorangegangenen Kapitel zu vertiefen, soll im Folgenden der Einsatz eines kontextbezogenen, adaptiven Systems in einem Anwendungsszenario untersucht werden.

3.1. Einführung in das Szenario

In zahlreichen produzierenden Betrieben muss die Funktionsfähigkeit der eingesetzten Werkzeugmaschinen und Fertigungssysteme sicher gestellt sein. Auf die damit verbundenen Aufgaben sind eigenständige Wartungsfirmen spezialisiert. Ihre technischen Angestellten (nachfolgend auch Techniker oder Ingenieure genannt), die mit der Betreuung der Anlagen betraut sind, benötigen zur Erledigung bzw. Erleichterung ihrer Tätigkeiten elektronische Unterstützung.

Im Folgenden werden die Aufgaben der Techniker betrachtet, sowie die dafür erforderliche Unterstützung durch elektronische Systeme. Letztere wird analysiert, um die grundlegenden Anforderungen an einen Kontextdienst zu identifizieren, welcher durch die Bereitstellung von Kontextinformationen die Realisierung solcher Systeme ermöglicht.

3.1.1. Aufgaben von Wartungsfirmen

Im Rahmen der Betreuung von Anlagen müssen verschiedene Aufgaben bewältigt werden, einerseits von den Technikern, andererseits von weiteren Mitarbeitern der Wartungsfirmen.

Technische Beratung

Ein Aspekt der Dienstleistung besteht in der Bereitstellung technischer Beratung für die Kunden. Treten während der Nutzung der Anlagen Fragen oder Probleme auf – z. B. bezüglich der Bedienung oder des Verhaltens der Anlage – so können diese mit dem technischen Kundendienst per Sprach- oder Videotelefonie geklärt werden.

Beim Eingang einer Anfrage greift der Kundendienst auf die Kundendaten zu, um die Vertragskonditionen des Kunden zu ermitteln und somit festzustellen, in welchem Umfang technische Beratung geleistet werden kann. Anschließend wird ermittelt, auf welche der Anlagen des Kunden sich die Anfrage bezieht. Die dafür erforderlichen Informationen über den Anlagenpark des Kunden sind ebenfalls in der Kundendatenbank enthalten. Anfragen, die Störungsmeldungen darstellen, werden sofort zur Koordination weiter geleitet, um entsprechende Instandhaltungsaufträge auszulösen. Handelt es sich lediglich um Fragen, so wird versucht, diese mit Hilfe der technischen Dokumentation der Anlage zu beantworten.

Sind die Kundendienstmitarbeiter – meist ebenfalls Techniker – nicht in der Lage, die Fragen zu beantworten, so muss ein geeigneter Techniker gefunden werden, der konsultiert werden kann. Diese Aufgabe wird durch die Koordinatoren gelöst. Kundenanfragen können jederzeit auftreten und bedürfen je nach Dringlichkeit kurz- oder mittelfristiger Beantwortung (z. B. durch Rückruf).

Überwachung

Um die Funktionsfähigkeit der Anlagen sicherzustellen, werden sie dauerhaft überwacht. Das erlaubt schnellstmögliche Erkennung von Fehlern und sogar deren Prognose.

Die Überwachung wird durch elektronische Systeme realisiert, welche den Anlagenzustand bewerten – z. B. die Einhaltung des Arbeitsbereiches bestimmter Parameter. Für die maschinelle Bewertung benötigt das System nicht nur Zugriff auf den Anlagenzustand und ggf. auch die Umgebung, sondern idealerweise auch technologische oder physikalische Modelle* der Anlage.

Werden während der Bewertung Fehler prognostiziert oder erkannt, so sind Meldungen an die Koordinatoren auszugeben, auf deren Basis Instandhaltungsaufträge

*Oder andere Maschinen-lesbare Modelle, welche zulässige (und ggf. unzulässige) Zustände der Anlage beschreiben.

ausgelöst werden können. Kann die Bewertung aufgrund fehlender oder unzureichender Informationen nicht erfolgen, so resultiert das in einer Meldung, die einen Instandhaltungsauftrag für eine Inspektion auslöst.

Instandhaltung

Zur Vermeidung von Ausfällen werden die Anlagen vorbeugend Instand gehalten [WB06a]. Ist ein Defekt bereits eingetreten, muss hingegen eine ausfallbedingte Instandhaltung vorgenommen werden. Beide Aufgaben sind sich sehr ähnlich, und werden deshalb zusammengefasst betrachtet.

Instandhaltungen bestehen aus vielfältigen Phasen bzw. Maßnahmen – z. B.:

- Kontrolle der Funktionsfähigkeit bzw. Feststellung des Ausmaßes von Schäden
- Überprüfung von automatisch erstellten Protokollen des Prozessrechners und der Diagnosesensorik der Anlage – ggf. auch Zugriff auf die Prozessleitrechner des Betriebes
- Versorgung mit Betriebsmitteln
- Austausch mechanischer und elektronischer Komponenten
- Installation von Software und Konfiguration der Anlage

Für diese Tätigkeiten benötigt der Techniker die technische Dokumentation der Anlage, sowie die Wartungsdokumentation. Aufgrund der Integration der Anlagen miteinander – durch Prozessleittechnik und Transportsysteme – kann es erforderlich sein, den Zustand mehrerer Anlagen zu erfassen. Wegen möglicher Wechselwirkungen ist darüber hinaus die Umgebung der Anlage zu beobachten. Des weiteren muss der Techniker ggf. mit den Koordinatoren oder dem Anlagenhersteller interagieren, um fehlende Ersatzteile, Betriebsmittel oder Hilfe durch weitere Kollegen zu organisieren. Werden drohende Ausfälle prognostiziert, deren Ursachen jedoch nicht behoben werden können, so müssen Meldungen an die Koordinatoren zur Auslösung zukünftiger Instandhaltungsaufträge übermittelt werden. Beschreibungen und zusätzliche Informationen (z. B. Messprotokolle) der durchgeführten Tätigkeiten werden teil- oder vollautomatisch in der Wartungsdokumentation abgelegt.

Während der Arbeiten an der Anlage trägt der Techniker oft sein Augmented Reality Headset. Diese Kombination aus Kamera, Kopfhörer, Mikrofon und Head-Mounted Display ermöglicht sowohl die Einblendung von Informationen in das Sichtfeld des Technikers, als auch die Interaktion mittels Sprachein- und -ausgabe, sowie Gesten. Somit sind die Hände für die Durchführung der Arbeiten frei, und wichtige Hinweise für Instandhaltungsprozeduren können an passender Stelle präsentiert werden.

Instandhaltungen finden entweder zeitabhängig, zustandsorientiert oder ausfallbedingt statt. Dem entsprechend sind sie lang-, mittelfristig oder gar nicht planbar.

Fernwartung

Ein Teil der Instandhaltungsmaßnahmen erfordert keine Anwesenheit der Techniker vor Ort, sodass sie als Fernwartungen ortsunabhängig vorgenommen werden können. Befindet sich ein Techniker während einer Fernwartung sowohl außerhalb der Wartungsfirma, als auch außerhalb der Kundenbetriebe, so hat er lediglich seine mobilen Geräte zur Verfügung. Ob sich dieser Umstand auf die Anforderungen an das Unterstützungssystem auswirkt, wird nachfolgend festgestellt.

Koordination

In den zuvor betrachteten Aufgaben wurde bereits mehrfach auf die Arbeit der Koordinatoren hingewiesen. Zu ihren Tätigkeiten gehört:

- Ermittlung geeigneter, verfügbarer Techniker
- Weiterleitung von Anfragen für technische Beratung
- Planung und Zuordnung von Instandhaltungsaufträgen
- Auftrags-abhängige Bereitstellung von Betriebsmitteln, Ersatzteilen, Werkzeugen, etc.
- Erfassung der Arbeiten für Rechnungslegung oder zur Ermittlung der Abdeckung durch Dienstleistungsverträge

Um festzustellen, ob ein Techniker fachlich geeignet ist, muss ein inhaltlicher Vergleich zwischen seinem Profil (in der Mitarbeiterdatenbank) und den fachlichen Anforderungen vorgenommen werden. Seine Verfügbarkeit wird jedoch durch den Aufenthaltsort, die aktuelle Tätigkeit, nutzbare mobile Geräte (und deren Eigenschaften), und vieles mehr beeinflusst. Verfügen die Geräte eines Technikers an seinem Aufenthaltsort nur über schmalbandige, langsame Netzwerkverbindung, so ist es ihm nicht oder nur schwer möglich, auf die umfangreiche technische Dokumentation zuzugreifen und Kundenanfragen zu beantworten. Hat der Techniker nicht die notwendigen Ersatzteile oder Werkzeuge für einen dringenden Instandhaltungsauftrag, so kann er nicht für diesen eingesetzt werden bzw. muss zuvor bei der Wartungsfirma oder dem Hersteller seine Ausrüstung vervollständigen. Derart detaillierte Informationen stehen den Koordinatoren nur dann zur Verfügung, wenn die Techniker umfassend überwacht werden können.

Für die zeitliche Planung von Aufträgen sind technische und Wartungsdokumentation erforderlich, die Aufschluss über den theoretisch veranschlagten und praktisch

aufgetretenen Arbeitszeitaufwand verschiedener Tätigkeiten geben. Des weiteren gibt die technische Dokumentation Aufschluss über benötigte Arbeitsmittel, die den Technikern für ihre Aufträge bereitgestellt werden müssen.

Die Koordinatoren stellen naturgemäß das Bindeglied zwischen allen übrigen Beteiligten dar. Um ihre Arbeit erledigen zu können bzw. um sie zu vereinfachen, werden sie durch ein so genanntes Computerized Maintenance Management System (CMMS) unterstützt. In dieses System finden sämtliche zuvor beschriebenen Informationen Eingang.

Koordination muss jederzeit gewährleistet sein – wie auch technische Beratung und Überwachung – weil sie das Bindeglied zwischen allen Tätigkeiten der Techniker darstellt. Auf Meldungen von Kunden, Kundendienst, Anlagenüberwachung und Technikern muss kurzfristig reagiert werden.

3.1.2. Zeitaufwand

Die in diesem Kapitel identifizierten Aufgaben sind von unterschiedlichen Akteuren zu erledigen, und sind mit unterschiedlichem Zeitaufwand verbunden. Die Überwachung der Anlagen wird vorwiegend automatisch durch das System vorgenommen, und hat während ihres Betriebes dauerhaft zu erfolgen. Parallel dazu muss das System die Techniker während ihrer Arbeitszeit überwachen, um jederzeit die notwendigen Informationen für ihre Koordination bereitstellen zu können. Der Arbeitsaufwand der Techniker verteilt sich aufgrund ihrer Mitwirkung auf die zuvor beschriebenen Aufgaben, wobei der Schwerpunkt auf der Durchführung von Instandhaltungen liegt. In den meisten Fällen ist die Anwesenheit des Technikers am Einsatzort der Anlage erforderlich, weil trotz fortschreitender Tendenz zur Fernwartung/-diagnose nicht alle Arbeiten entfernt durchgeführt werden können oder dürfen.

- Die *Technische Beratung* wird vorwiegend durch den telefonischen Kundendienst gewährleistet. Somit spielt sie eine untergeordnete Rolle, und beansprucht nur wenig Arbeitszeit eines Technikers.

- *Überwachung* findet dauerhaft und weitestgehend automatisch statt. Der Techniker muss nur selten bei unklaren Ergebnissen die Situation einer Anlage durch eine Inspektion vor Ort beurteilen. Es wird auch hier ein geringer Zeitaufwand erwartet.

- Ausfallbedingte Instandhaltungen – *Reparaturen* – sind aufgrund der hohen Qualität der Maschinen und angemessener Planung präventiver Instandhaltungen selten durchzuführen. Da sie jedoch zeitaufwendig sind, verbringt der Techniker einen nennenswerten Teil seiner Arbeitszeit damit.

- Zeitabhängige oder zustandsorientierte Instandhaltungen, welche präventiv statt finden, nehmen den größten Teil der Arbeitszeit eines Technikers in An-

spruch. Allerdings kann ein Teil davon als Fernwartung erfolgen. Trotzdem ist der überwiegende Anteil der Arbeiten vor Ort durchzuführen.

3.1.3. Notwendige Systemunterstützung

Um die beschriebenen Aufgaben zu unterstützen, muss ein entsprechendes System in erster Linie über die notwendigen Informationen verfügen. Folglich besteht die Kernfunktion des Systems im *Informationszugriff*. Handelt es sich bei den Anwendern um Techniker, welche abseits ihrer Firmensitze über mobile Geräte auf Informationen zugreifen, so wird eine weitere Systemfunktion erforderlich – die *Anpassung* der Benutzungsschnittstellen ihrer Anwendungen an die beschränkten Ressourcen (speziell die Präsentationsmöglichkeiten) der Geräte und an die Arbeitssituation der Techniker. Dies schließt auch die präsentierten Dokumente und Informationen ein, welche ggf. umfangreich und komplex sein können. Weil diese Funktion nur dann erbracht werden kann, wenn wiederum Informationen über die Geräte und die Situationen der Techniker verfügbar sind, stellt der Informationszugriff nach wie vor die zentrale Aufgabe dar. Die Nutzung der Informationen für die Adaption der Benutzungsschnittstellen soll jedoch nicht näher betrachtet werden. Die damit verbundene eingehende Diskussion von Problemen der Mensch-Maschine-Interaktion würde den Rahmen dieser Arbeit sprengen.

3.2. Informationen

Das skizzierte Szenario benennt eine Vielzahl von Informationen, auf welche die Angestellten und die von ihnen benutzten Anwendungen zugreifen müssen. Sie werden nachfolgend hinsichtlich ihrer Eigenschaften untersucht. Die genannten Zahlen sind als Orientierungswerte zu betrachten, weil sie in Abhängigkeit des betrachteten Industriezweiges, der Betriebsgröße, sowie innerbetrieblicher Abläufe usw. sehr stark von den tatsächlichen Werten abweichen können.

3.2.1. Technische Dokumentation

Die Anlagenproduzenten – oder Hersteller – erstellen die technische Dokumentation der Anlagen. Sie sind es auch, die den Zugriff darauf über entsprechende Server ermöglichen. Die technische Dokumentation beschreibt die Eigenschaften und den Aufbau einer Anlage. Das beinhaltet vor allem Spezifikationen, Bau- und Schaltpläne, Betriebs- und Instandhaltungsanleitungen, sowie u. U. sogar technologische und physikalische Modelle der Anlage. Letztere sind jedoch meist nicht in elektronischer Form vorhanden oder stellen hoch sensibles Wissen der Hersteller dar, welches entsprechend geheim ist. Somit beschreiben sie ggf. nur einzelne Subsysteme. Darüber

hinaus basieren diese Teilmodelle oft auf unterschiedlichen, meist proprietären Metamodellen, weil es keine einheitlichen Standards gibt, auf welche die Hersteller und ihre Zulieferer zurückgreifen können.

Nach dem Zeitpunkt der Auslieferung oder Inbetriebnahme ändern sich diese Informationen kaum noch – z. B. zur Dokumentation neuer Software. Die technische Dokumentation ist sehr umfangreich – pro Anlage mehrere Hundert Megabyte Datenvolumen (Annahme: 500 MB* pro Anlage). Weil sich diese Informationen nicht ändern, ist oft eine einmalige Übertragung beim Zugriff darauf ausreichend. Nachfolgende Übertragungen von Aktualisierungen müssen nicht statt finden. Die technische Dokumentation ist Anlagen-spezifisch, d. h. bei der Suche nach benötigten Informationen wird die verfügbare Informationsmenge primär auf die betreffende Anlage eingeschränkt.

Die Geräte der Mitarbeiter an den Standorten der Wartungsfirmen unterliegen keinerlei Einschränkungen bzgl. Netzwerkressourcen oder Präsentation der Dokumentation. Techniker die jedoch Instandhaltungsaufträge bei Kunden durchführen, können nur ihre mobilen Geräte nutzen. Sie sind eingeschränkt hinsichtlich Präsentationsfähigkeiten, Speicherplatz und Netzwerkverbindungen. Obwohl ihnen im Außeneinsatz die besondere Präsentationsfunktionalität von Augmented Reality Headsets zur Verfügung steht, stellt die situationsabhängige Präsentation der komplexen Dokumentation eine große Herausforderung für das System dar.

3.2.2. Wartungsdokumentation

Die Wartungsfirmen dokumentieren sämtliche durchgeführten und geplanten Instandhaltungsarbeiten, sowie damit verbundene Informationen (z. B. Prüfprotokolle). Im Allgemeinen sind diese Informationen für die Mitarbeiter am Firmensitz und im Außeneinsatz über die Server der Wartungsfirma zugreifbar. Während ein Techniker eine Instandhaltung durchführt, sind deren Informationen nur über seine mobilen Geräte verfügbar.

Obwohl die Wartungsdokumentation jeder einzelnen Anlage nie einen endgültigen Stand erreicht, ändert sie sich nur selten, weil nur wenige Instandhaltungen pro Jahr und Anlage erforderlich sind. Man kann folglich von semi-statischen Informationen ausgehen. Analog zur technischen Dokumentation wird auf die Wartungsdokumentation einmalig zugegriffen. Dabei ist aufgrund des Anlagenbezuges auch hier die Identität der konkreten Anlage das primäre Suchkriterium.

Es wird davon ausgegangen, dass die Wartungsdokumentation einer Anlage ein geringeres Datenvolumen besitzt, als deren technische Dokumentation – schätzungsweise 50 MB pro Anlage. Somit sind die Effekte beim Zugriff eingeschränkter, mobiler Geräte auf die Wartungsdokumentation weniger gravierend. Auch ist die Komple-

*Megabyte (MB)

xität dieser Daten geringer, sodass weniger umfangreiche Präsentationsfähigkeiten der Geräte vorausgesetzt werden müssen.

3.2.3. Kundendaten

Neben der Wartungsdokumentation pflegen die Wartungsfirmen jeweils ihre eigenen Kundendatenbanken, welche natürlich auch außerhalb der Firmenstandorte zugreifbar sein müssen. Inhalt dieser Datenbestände sind u.a. die Konditionen der Dienstleistungsverträge mit den Kunden, Informationen über die Kundenstandorte und den jeweils dort vorhandenen Anlagenpark.

Ähnlich wie die Wartungsdokumentation sind auch die Kundendatenbanken ständigen Änderungen unterworfen. Jedoch sind auch hier die Intervalle zwischen den Änderungen vergleichsweise hoch, d. h. meist Tage bis Monate. Deshalb können diese Daten ebenfalls als semi-statisch betrachtet werden, d. h. einmalige Zugriffe sind ausreichend. Werden die Kundendatenbanken von den Mitarbeitern genutzt, was meist navigierend der Fall ist, dann wird häufig zuerst ein konkreter Kunde selektiert, bevor die Daten detaillierter gefiltert werden.

Die Informationen in der Kundendatenbank sind voraussichtlich meist einfach strukturiert und wenig umfangreich (Annahme: 20 MB pro Kunde).

3.2.4. Anlagenzustand

Im Gegensatz zu allen zuvor betrachteten Informationen werden diejenigen, welche den Zustand der Anlagen repräsentieren, nicht manuell durch die Mitarbeiter der Wartungsfirmen, sondern automatisch durch Sensoren erfasst. Sie sind Teil der Anlage, einer anlagennahen Diagnoseelektronik oder spezielle Mess- und Prüfgeräte, welche von den Technikern bedarfsabhängig eingesetzt werden. Meist handelt es sich dabei um Hardwaresensoren, aber auch die Software der Prozessrechner der Anlagen und der Prozessleitrechner kann als Informationsquelle dienen. Zum Zustand der Anlage gehören auch die Zustände ihrer Subsysteme (z. B. Energieversorgung, Antrieb, Werkzeuge, etc.), sowie Modelle der Prozesse der Anlage (z. B. Workflows, Arbeitsabläufe, Fertigungs- und CAD/CAM*-Modelle). Dementsprechend kann der Zustand vielfältige Informationen enthalten:

- Elektrik (Spannung, Strom, Phasenverschiebung)
- Mechanik (Druck, Drehzahl, Geschwindigkeit, Kraft, Beschleunigung)
- Akustik (Vibration, Geräusch)
- Temperatur

*Computer-Aided Design (CAD), Computer-Aided Manufacturing (CAM)

- Betriebsmittel (Füllstand bzw. Vorrat)
- Werkzeuge
- Software (Prozess/Arbeitsschritt, Daten, Konfiguration, installierte Software/-Firmware, produziertes Werkstück)

Die einzelnen Informationen des Anlagenzustandes können sich sehr schnell ändern, vor allem, weil es sich dabei um physikalische und technische Größen handelt. Je höher die vom Konsumenten – dem System – geforderte Auflösung solcher Größen ist, desto höher ist meist auch die Rate relevanter Änderungen der Informationen.

Die Diagnose des Anlagenzustandes kann auf zwei Wegen erfolgen. Einerseits können die Veränderungen des Anlagenzustandes dauerhaft beobachtet werden. Das ist z. B. bei regelnden Eingriffen notwendig, deren Effekte nicht (vollständig) bekannt sind. Dabei muss der Informationszugriff asynchron erfolgen, d. h. die benötigten Informationen werden beobachtet, sodass jede Veränderung sofort erkennbar ist. Andererseits kann die Feststellung des Anlagenzustandes zu einem konkreten Zeitpunkt erforderlich sein. Dies ist vollkommen ausreichend, wenn ermittelt werden soll, ob durchgeführte Maßnahmen den gewünschten/erwarteten Effekt zeigen. Für die momentane Feststellung des Anlagenzustandes ist der zuvor schon erwähnte synchrone Informationszugriff notwendig. Aufgrund des Anlagenbezuges der Informationen stellt die Anlage auch das primäre Suchkriterium beim Zugriff dar. Bei der Untersuchung konkreter Aspekte oder Bereiche sind darüber hinaus weitere Suchkriterien wie der Ursprungsort oder der Typ einer Information erforderlich.

Die Informationen des Anlagenzustandes sind vielfältig und häufig vorhanden, jedoch von geringem Volumen. Ein einzelner Anlagenkennwert besteht oft nur aus wenigen Byte oder sogar Bits. Folglich sind potentielle technische Zugriffsprobleme vorwiegend bei der Auswahl der benötigten Informationen zu erwarten, nicht jedoch bei deren Speicherung, Verarbeitung oder Präsentation. Für den asynchronen Zugriff gilt einschränkend, dass die potentiell hohe Änderungsrate in einer großen Zahl von Benachrichtigungen resultiert, sodass deren zeitnahe Übertragung problematisch werden kann.

Zusätzlich zu den technischen Problemen sind in vielen Betrieben administrative Beschränkungen des Zugriffs auf den Anlagenzustand üblich. Weil die Anlagen von existenzieller Bedeutung für den Betrieb sind, werden sensible Informationen – und dazu gehört der Anlagenzustand – gar nicht, oder nur unter strengen technischen Auflagen für Personen außerhalb des Betriebes zugreifbar gemacht. Das bedeutet, dass häufig das Netzwerk, welches die Anlagen und die Prozessleittechnik verbindet, keine Verbindung zum Internet hat. Wenn dies doch der Fall ist, werden Sicherheitsmaßnahmen wie Firewalls eingesetzt, um Art und Menge der ausgetauschten Informationen zu kontrollieren. In der Konsequenz kann dies bedeuten, dass Fernwartungen gar nicht, oder nur in eingeschränktem Maße möglich sind.

3.2.5. Umgebungsbeschreibung

Wie der Anlagenzustand wird auch die Umgebung der Techniker und der Anlagen durch Sensoren erfasst. Befinden sie sich in einem Gebäude, so können die Sensoren der Gebäudeautomation beispielsweise folgende Informationen liefern:

- Temperatur
- Helligkeit/Beleuchtung
- Geräuschpegel
- Luftqualität (z. B. Druck, Feuchte, Reinheit, Zusammensetzung)
- Sicherheitswarnung (z. B. Feueralarm)

Unabhängig davon können die mobilen Geräte des Technikers ebenfalls seine Umgebung beobachten. Neben Sensoren für Temperatur, Helligkeit oder Geräuschpegel verfügen manche Geräte sogar über einen Beschleunigungssensor – z. B. Notebooks zum aktiven Schutz ihrer Festplatte – oder über einen GPS-Empfänger. Die Geräte weiterer Personen, sowie ggf. einzelne, unabhängige Sensoren in der Umgebung können zusätzliche Informationsquellen sein.

Aufgrund der Mobilität der Techniker kann sich ihre aktuelle Umgebung schnell ändern. Wie beim Anlagenzustand sind auch die Informationen der Umgebung wenig komplex und kompakt. Die Umgebung wird vorwiegend beobachtet, um störende Umgebungseinflüsse durch Anpassung der Benutzungsschnittstelle der Geräte/des Systems zu kompensieren. Dies setzt vorwiegend asynchronen Informationszugriff voraus. Außerdem ist eine räumliche Eingrenzung der Informationen auf diejenigen in der direkten Umgebung des Technikers erforderlich. Diese Beschränkung ist auch notwendig, wenn – im Rahmen der Anlagendiagnose – mögliche Wechselwirkungen zwischen Umgebung und Anlage untersucht werden sollen. Zusätzlich kann eine Analyse der Änderungen hinsichtlich Häufigkeit, Schwankungsbreite, etc. für die grob granulare Steuerung der Adaption nützlich sein, d. h. beispielsweise: Deaktivierung der automatischen Adaption der Benutzungsschnittstelle der mobilen Geräte bei zu hoher Dynamik der Umgebung. Deshalb ist das Interesse an zeitlich hoch aufgelösten Messwerten geringer, d. h. es werden tendenziell weniger häufig Änderungsnachrichten kommuniziert. Dementsprechend sind die Anforderungen an die Ressourcen der mobilen Geräte – speziell an deren Netzwerkverbindung – geringer, als beim Zugriff auf den Anlagenzustand.

3.2.6. Technikersituation

Die Situationen in denen sich die Techniker während ihrer Arbeit befinden, sind vorwiegend durch ihre Tätigkeiten – aktuell und zukünftig bzw. geplant – charakterisiert. Sind sie nicht explizit bekannt, kann der Aufenthaltsort bzw. die Umgebung

eines Technikers weitere Hinweise geben, oder den Zugriff auf weitere Informationen (z. B. die Umgebung) erlauben, welche Rückschlüsse auf seine Tätigkeit gestatten. Neben den Situationen der Techniker könnten theoretisch auch die anderer Akteure (z. B. Koordinatoren, Kunden) beobachtet werden. Jedoch sind diese entweder nicht mobil, oder benötigen keine Unterstützung durch das System, sodass sie nachfolgend nicht weiter betrachtet werden.

Nur durch die Menge der Geräte in seiner Umgebung ist die Beobachtung der Arbeitssituation eines Technikers möglich. Folglich besteht diese Menge aus zwei Gruppen – den mobilen Geräten, welche ein Techniker mit sich führt, und den Geräten, welche unabhängig von ihm (stationär oder mobil) in seiner Umgebung existieren.

Betrachtet man die Arbeitssituation lediglich grob granular, so ändert sie sich im Vergleich zum Anlagenzustand nur langsam – meist nur innerhalb von mehreren Minuten. Für die Überwachung im Rahmen der Koordination durch die Mitarbeiter am Firmensitz ist diese Genauigkeit voraussichtlich ausreichend. Für die Anpassung der Benutzungsschnittstellen der Geräte gilt dies nicht. Beispielsweise kann die Darstellung von Bauteilen oder komplexen Instandhaltungsanweisungen im Display des Augmented Reality Headsets nur dann mit den vom Techniker betrachteten Objekten synchron erfolgen, wenn Sensorinformationen über seine Blickrichtung mit der nötigen zeitlichen und räumlichen Auflösung verfügbar sind. In diesen Fällen ist eine kontinuierliche Überwachung, d. h. asynchroner Zugriff, notwendig.

Die qualitativen Anforderungen an die Informationen sind folglich abhängig vom Anpassungsgrad des Systems an die Arbeitssituation. Die begrenzten Kommunikationsmittel – speziell für WAN*-Kommunikation – der mobilen Geräte können einen Engpass darstellen, sofern detaillierte Informationen mit entfernten Teilen des Systems ausgetauscht werden müssen. Für die Aufbereitung komplexer Instandhaltungsdokumente entsteht somit die Problematik: entweder (1) Übertragung der vollständigen Dokumente an die mobilen Geräte und anschließende lokale Anpassung, oder (2) Übertragung komplexer und volatiler Umgebungsinformationen für entfernte Anpassung der Dokumente und anschließende Übertragung der angepassten Dokumente zu den mobilen Geräten. Lokale Kommunikation hingegen wird als weniger problematisch betrachtet.

3.2.7. Heterogene Informationen

Es zeigt zeigt sich eine hohe Heterogenität der Informationen in mehreren Aspekten. Ihre Struktur reicht von einfachen Messwerten bis hin zu komplexen Situationsbeschreibungen oder technischen Dokumenten. Damit verbunden schwankt auch das Datenvolumen zwischen wenigen Byte und mehreren Hundert Megabyte. Weiter-

*Wide Area Network (WAN)

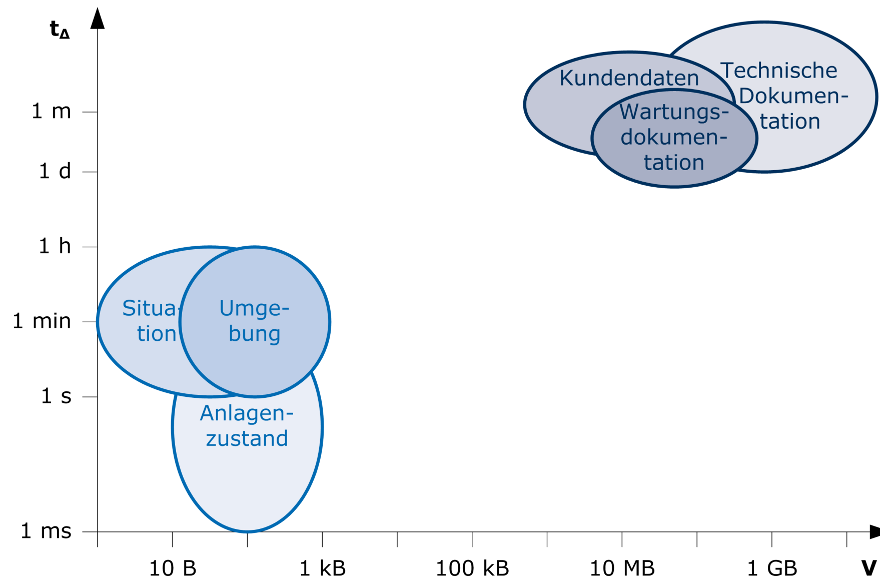

Abbildung 3.1.: Heterogenität von Informationen – mittlere Volumina V und mittlere Änderungszeiten t_Δ

hin besitzen die Informationen unterschiedliche Volatilität. Kombiniert man diese beiden Aspekte, so wird erkennbar (Abb. 3.1), dass zwei Klassen von Informationen existieren – (1) großvolumige, semi-/statische und (2) kompakte, dynamische Informationen.

Grundsätzlich ist zu bemerken, dass die Darstellung in Abbildung 3.1 stark vereinfacht ist. In realen Szenarien ist davon auszugehen, dass es keine klaren Grenzen zwischen den beiden Informationsklassen gibt. Im Rahmen dieser Arbeit wurde jedoch eine derartige Annahme getroffen, um eine Fokussierung auf die Behandlung der, als problematisch einzuschätzenden, dynamischen Informationen zu gestatten. In Tabelle 3.1 auf der nächsten Seite sind die wesentlichen Eigenschaften der zuvor untersuchten Informationen zusammengefasst. Die aufgeführten Volumina stellen den Speicherplatz* dar, welcher zu einem konkreten Zeitpunkt erforderlich ist, um die genannten Informationen zu repräsentieren. Sie enthalten keinerlei Aussage über die Änderungsrate der Informationen und somit auch nicht über die innerhalb eines konkreten Zeitraumes kumulierten Volumina V_k. Eine entsprechende Abschätzung ist jedoch leicht möglich (Gl. 3.1).

$$V_k = \int \frac{V}{t_\Delta} \, dt \tag{3.1}$$

*angegeben in Byte (B), Kilobyte (kB) oder Megabyte (MB)

Tabelle 3.1.: Struktur, Volatilität und Volumen von Informationen

Information	**Struktur**	**Volatilität t_Δ**	**Volumen V**
Technische Dokumentation	sehr komplex	statisch / sehr gering (Monate - Jahre)	sehr hoch (500 MB / Anlage)
Wartungsdokumentation	komplex	semi-statisch / gering (Wochen - Monate)	mittel (50 MB / Anlage)
Kundendaten	komplex	semi-statisch / gering (Wochen - Monate)	mittel (20 MB / Kunde)
Anlagenzustand	einfach	dynamisch / sehr hoch (Milli- - Sekunden)	sehr gering (10 B / Wert – 1 kB / Anlage)
Umgebung	einfach	dynamisch / hoch (Sekunden - Minuten)	sehr gering (10 B / Wert – 1 kB / Umgebung)
Situation	einfach	dynamisch / hoch (Sekunden - Minuten)	sehr gering bis gering (10 B / Wert – 1 kB / Situation)

Weiterhin sind die meisten Informationen nicht isoliert/allgemein gültig, sondern haben einen konkreten Bezug, ohne den sie ungültig bzw. irrelevant sind. Sie besitzen, somit neben ihrem Typ, oft auch einen Bezugsort (meist den ihrer Entstehung), eine Bezugsperson oder sie sind, wie im geschilderten Szenario, auf eine konkrete Anlage bezogen. Bei seiner Arbeit greift ein Techniker vorwiegend auf Informationen aus seiner direkten räumlichen Umgebung zu. Sei dies bei der Beobachtung der Umgebung oder bei der Überwachung des Anlagenzustandes. Die Informationen der übrigen Informationsquellen bleiben ungenutzt. Folglich darf die Verwaltung der Informationsquellen keinen unnötigen Aufwand mit sich bringen.

A 15. *Die Verwaltung der Informationsquellen muss derart effizient sein, dass der daraus entstehende Aufwand sehr gering ist – selbst wenn keine oder nur wenige der verwalteten Quellen genutzt werden. Idealerweise ist der Verwaltungsaufwand unabhängig von der Menge der verwalteten Quellen.*

3.3. Akteure und Informationsquellen

Die Quellen, welche die betrachteten Informationen liefern, wurden im Szenario bisher lediglich erwähnt. Nun soll deren noch ausstehende Untersuchung vorgenommen werden. Dazu werden die verschiedenen Orte bzw. Umgebungen der Beteiligten des Szenarios betrachtet. Schwerpunkte dabei sind das Vorkommen von Quellen und der Zugriff auf sie.

3.3.1. Beim Anlagenhersteller

Die Hersteller der Anlagen erstellen die technische Dokumentation der Anlagen und stellen Ersatzteile, Zubehör und Software für die Anlagen bereit. Die Anwendungen auf den Servern der Hersteller erlauben sowohl den Zugriff auf die Dokumente, als auch die Abwicklung von Bestellungen. Diese Server sind stationär und verfügen über umfangreiche Ressourcen bezüglich Verarbeitungsleistung (Speicher und Prozessoren) und Kommunikationsfähigkeit (Netzanbindung). Verschiedene Akteure – Techniker, Betriebe, etc. – greifen bei unterschiedlichen Arbeiten – Beratung, Instandhaltung, Nutzung – auf die Server zu, d. h. jederzeit und von überall.

3.3.2. In der Wartungsfirma

Durch die Wartungsfirmen werden ebenfalls Server betrieben, welche ihren Mitarbeitern den Zugriff auf die Kundendaten und die Instandhaltungsdokumentation, sowie den Betrieb des CMMS ermöglichen. Der Zugriff auf die Server der Wartungsfirmen ähnelt dem der Herstellerserver. Zusätzlich sind in den Wartungsfirmen noch die Bürocomputer der Mitarbeiter und ggf. Rechner der Gebäudeautomation vorhanden. Beide können Informationen über die Umgebung und die Situation der Nutzer liefern. Auf diese Daten wird in zwei Formen zugegriffen: (1) von den Koordinatoren am Sitz der Wartungsfirma – bzw. vom CMMS – zur Überwachung und (2) von den Geräten der Nutzer zur Adaption ihrer Benutzungsschnittstellen. Im ersten Fall findet die Kommunikation zwischen dem Firmensitz und den mobilen Technikern statt, und es werden kompakte Informationen (z. B. Ort, Aktivität) ausgetauscht. Im zweiten Fall kommunizieren die Geräte der Techniker mit den Geräten in ihrer direkten Umgebung, um diese zu beobachten, und tauschen dabei ggf. auch komplexere Daten aus.

3.3.3. Im Produktionsbetrieb

In den Betrieben selbst sind diverse Quellen im Einsatz. Dazu gehören nicht nur die von den Technikern betreuten Anlagen – Werkzeugmaschinen und Fertigungssysteme – sondern auch Transport- und Lagersysteme. Teil dieser Systeme sind heutzutage meist Rechner, wie: Prozess-, Zellen-, Fertigungs- und Betriebsrechner [WB06b]. Weil es sich bei den Sensoren der Anlagen um hoch spezialisierte Komponenten handelt, können sie nicht selbst als Informationsquellen eingebunden werden, sondern sind über die soeben genannten Rechner nutzbar.

Neben der eigentlichen Fertigung existiert auch elektronische Unterstützung für andere Aspekte der Produktion – wie Entwicklung, Planung, Qualitätssicherung, etc. Für die Arbeit der Techniker sind diese Aspekte jedoch ohne Belang, weil sie nicht über die fachliche Kompetenz verfügen, die Wechselwirkungen – speziell Fehler – dieser Aspekte mit den Anlagen zu untersuchen.

Wie die Gebäude der Wartungsfirmen, so sind auch die der Produktionsbetriebe automatisiert und können entsprechend überwacht werden. Wenn administrativ möglich (siehe Kap. 3.2.4), findet im Rahmen der Überwachung zwischen den Wartungsfirmen und den Anlagen, d. h. im Weitbereich (WAN), Kommunikation statt, die meist nur eine grob granulare Einschätzung (z. B. Warnmeldungen) des Anlagenzustandes erlaubt. Wenn notwendige Instandhaltungen in Form von Fernwartungen durchgeführt werden, dann ist die Kommunikation bzgl. Inhalten oder Details oft ebenfalls derart beschränkt. Finden Instandhaltungen vor Ort im Betrieb statt, dann konzentriert sich die Kommunikation meist auf den Nahbereich (Local Area Network (LAN)) der Anlage und der Techniker. Die Produktionsstätten sind, wie die Wartungsfirmen, mit Gebäudeautomation ausgestattet, sodass auch hier eine Beobachtung der Umgebung der Techniker möglich ist. Zusätzlich zur Anpassung der Benutzungsschnittstelle, können diese Informationen auch z. B. zur Diagnose der Anlagen genutzt werden.

3.3.4. Beim Techniker

Entsprechend seiner Arbeitsaufgaben befindet sich ein Techniker meist in einem der Produktionsbetriebe oder in seiner Wartungsfirma. Diese beiden Umgebungen und die dort vorhandenen Informationsquellen wurden bereits betrachtet. Darüber hinaus, verfügt ein Techniker meist über weitere Informationsquellen, welche er in Form mobiler, persönlicher Geräte jederzeit mit sich führt – z. B. Smart Phone, Personal Digital Assistant (PDA), Notebook. auftragsabhängig kann diese Menge um Spezialgeräte ergänzt werden – z. B. Diagnosegeräte, elektronische Werkzeuge, etc.

Letztere Geräte besitzen aufgrund ihrer Zweckbindung häufig wenig freie Ressourcen, welche die Erbringung zusätzlicher Funktionalität – z. B. Betrieb eines Kontextdienstes – gestatten. Anders ist dies bei den persönlichen Geräten. Sie sind häufig generischer Natur und können für vielfältige Zwecke eingesetzt werden – je nachdem, welche Software sie gerade ausführen. Die Ermittlung des Aufenthaltsortes eines Technikers wird oft erst durch seine persönlichen Geräte ermöglicht, da ihre Position von ihnen selbst oder extern feststellbar ist, und sie fast immer in seiner Nähe sind. Zusätzlich erlauben sie die Beobachtung genutzter Software und weiterer Geräte, sowie der direkten Umgebung des Technikers, und gestatten somit Rückschlüsse auf seine gegenwärtige Aktivität.

Bei der Koordination durch die Kollegen in der Wartungsfirma werden einerseits diese grob granularen Informationen dauerhaft beobachtet, und anderseits im Bedarfsfall fein granulare Informationen – wie vorhandene Geräte, Ersatzteile und Betriebsmittel ermittelt. Für die Anpassung der Benutzungsschnittstellen der Geräte und ihrer Anwendungen ist eine fein granulare Beobachtung der Umgebung erforderlich. Findet die Anpassung auf den mobilen Geräten des Technikers statt, so

stellen deren Netzwerkverbindungen keinen Engpass dar, weil breitbandige, lokale Kommunikation möglich ist. Bei einer Anpassung außerhalb der Technikerumgebung kann der Zugriff auf eine fein granulare Beschreibung der Umgebung aufgrund der größeren Verzögerung und niedrigeren Bandbreite der in diesem Fall genutzten MAN*/WAN-Verbindungen problematisch werden.

A 16. *Den begrenzten Ressourcen der mobilen und spezialisierten Geräte ist Rechnung zu tragen. Dies gilt speziell für deren meist schmalbandige und langsame Kommunikationsschnittstellen.*

3.3.5. Gerätelandschaft

Ein Überblick über die Informationsquellen (Tab. 3.2) in Verbindung mit den Charakteristika ihrer Informationen (Kap. 3.2.7) zeigt, dass die Klasse der voluminösen, statischen Informationen vorwiegend von leistungsfähigen, stationären Geräten bereitgestellt wird. Die kompakteren, dynamischen Informationen hingegen, müssen in erster Linie durch Nutzung der Quellen leistungsschwächerer – z. T. auch mobiler – Geräte gewonnen werden. Diese zweite Gruppe ist folglich nicht nur wegen ihrer hohen Volatilität problematisch, sondern auch wegen der Ressourcenbeschränkungen der Geräte, welche die Informationen liefern, und der z. T. vorhandenen Mobilität.

Tabelle 3.2.: Umgebungen und ihre Informationsquellen

<table>
<tr><th>Umgebung</th><th>Quellen</th><th>Informationen</th></tr>
<tr><td>Anlagenhersteller</td><td>Server</td><td>Technische Dokumentation</td></tr>
<tr><td rowspan="5">Wartungsfirma</td><td rowspan="2">Server</td><td>Wartungsdokumentation</td></tr>
<tr><td>Kundendaten</td></tr>
<tr><td rowspan="2">Arbeitsplatzrechner</td><td>Situation</td></tr>
<tr><td rowspan="2">Umgebung</td></tr>
<tr><td>Gebäudeautomation</td></tr>
<tr><td rowspan="3">Produktionsbetrieb</td><td>Anlagensensoren</td><td rowspan="2">Anlagenzustand</td></tr>
<tr><td>Diagnoseeinrichtungen</td></tr>
<tr><td>Gebäudeautomation</td><td>Umgebung</td></tr>
<tr><td rowspan="5">Techniker – überall, mobil</td><td rowspan="3">persönliche Geräte</td><td>Wartungsdokumentation</td></tr>
<tr><td>Situation</td></tr>
<tr><td rowspan="2">Umgebung</td></tr>
<tr><td rowspan="2">Spezialgeräte</td></tr>
<tr><td>Anlagenzustand</td></tr>
</table>

*Metropolitan Area Network (MAN)

Weiterhin sind die Informationen der Quellen aus der zweiten Gruppe als Kontextinformationen zu betrachten. Dazu zählen der Anlagenzustand, die Umgebung von Anlagen und Technikern, sowie die Situation der Techniker. Dementsprechend können Gebäudeautomation, Anlagensensoren, Diagnoseeinrichtungen, Bürocomputer, persönliche Geräte und Spezialgeräte neben ihrer primären Funktion auch als Kontextquellen genutzt werden.

3.4. Informationssenken und -zugriff

Nachdem die Quellen und die von ihnen gelieferten Informationen betrachtet wurden, rückt nun der Zugriff auf die Informationen bzw. ihre Quellen in den Mittelpunkt. Weil der Zugriff für unterschiedliche Zwecke erfolgt, beginnt die Analyse ausgehend von den bereits geschilderten Aufgaben.

3.4.1. Aufgaben

Technische Beratung

Bei der Technischen Beratung greifen die *Kundendienstmitarbeiter* der Wartungsfirmen auf die *Kundendatenbank*, sowie auf die *technische Dokumentation* und die *Wartungsdokumentation* der Anlagen zu. Weil diese Informationen *kunden- bzw. anlagenbezogen* sind, stellen Kunde und Anlage auch jeweils ein wesentliches Such- oder Zugriffskriterium dar. Die Informationen werden direkt von den Mitarbeitern genutzt, d. h. nicht durch ein Softwaresystem verarbeitet. Dem entsprechend wird die Informationsmenge einerseits durch iterative Suchanfragen und andererseits durch systematische Navigation eingegrenzt.

Überwachung

Die Überwachung der Anlagen wird durch automatische Systeme der Wartungsfirmen vorgenommen. Es ist dauerhafter Zugriff auf den Zustand und die Umgebung der Anlagen notwendig, um Änderungen sofort zu erkennen. Zur Validierung der erfassten Informationen wird bedarfsgesteuert auf die technischen Dokumentationen der Anlagen zugegriffen. Beim Zugriff auf den Anlagenzustand stellt die Anlage das primäre Suchkriterium dar. Ausgehend von der Position der Anlage wird ihre Umgebung beobachtet. Folglich werden Umgebungsinformationen primär anhand ihres Bezugs- oder Ursprungsortes selektiert.

Durch bestehende administrative Einschränkungen (Kap. 3.2.4) kann die externe Überwachung für ganze Betriebe gar nicht oder nur eingeschränkt möglich sein.

Instandhaltung und Fernwartung

Die vornehmliche Aufgabe der Techniker besteht in der Instandhaltung der Anlagen, welche sie entweder vor Ort im Produktionsbetrieb oder als Fernwartungen durchführen. In beiden Fällen benötigen sie die technischen Dokumentationen, Wartungsdokumentationen und Kundendaten – analog zur technische Beratung. Der Zugriff auf diese Informationen findet wie bei der technischen Beratung statt. Während der einzelnen Arbeitsschritte greifen die Techniker auf den Zustand der Anlagen und der Umgebung zu. In Abhängigkeit des Arbeitsschrittes dauert der Zugriff bzw. die Beobachtung nur einen Moment, oder erstreckt sich über eine längere Periode (meist mehrere Minuten). Um sich an die Arbeitsbedingungen der Techniker anzupassen, beobachten ihre persönlichen Geräte dauerhaft die Umgebung.

Befindet sich ein Techniker vor Ort in einem Betrieb, können sich die administrativen Regelungen, welche bei der Anbindung der Anlagen (kein Zugang zum Internet) bestehen, auch auf andere Informationen auswirken. Dann ist der Zugriff darauf lediglich über die limitierten Kommunikationsschnittstellen der mobilen Geräte des Technikers möglich.

Koordination

Die Koordinatoren bzw. das CMMS der Wartungsfirma muss jederzeit feststellen können, wo sich die Techniker befinden, und welche Arbeiten sie gerade erledigen. Folglich kann bei eingehenden Aufträgen die grundsätzliche zeitliche und örtliche Verfügbarkeit der Techniker nur über deren Situation ermittelt werden. Neben der Verfügbarkeit muss auch die Eignung der Techniker anhand ihres Mitarbeiterprofils geprüft werden. Nicht zuletzt sind die benötigten Werkzeuge, Geräte und Betriebsmittel zu identifizieren, sowie die Ausstattung des Technikers mit diesen sicher zu stellen. Dies ist nicht ohne Kenntnis der technischen Dokumentation und ohne Erfassung der Umgebung des Technikers möglich.

3.4.2. Senken

Die Akteure, welche an den verschiedenen Aufgaben beteiligt sind, betreiben gleichzeitig auch die Informationssenken. Zu ihnen gehören:

- die Wartungsfirmen mit:
  - den Kundendienstmitarbeitern,
  - dem System zur Anlagenüberwachung,
  - den Technikern (inkl. ihrer persönlichen Geräte) und
  - der Anwendung (für die Koordinatoren) zur Überwachung der Techniker, sowie

- die Hersteller.

Tabelle 3.3 auf der nächsten Seite fasst die Charakteristika des Zugriffs der verschiedenen Senken auf die unterschiedlichen Informationen zusammen. Jeder Eintrag beschreibt die primären Zugriffskriterien und den zeitlichen Aspekt des Zugriffs (synchron oder asynchron). Dabei ist zu beachten, dass der Typ einer Information in allen Fällen zu den primären Zugriffskriterien bzw. Indizes gehört. Er wird jedoch aus Gründen der Übersichtlichkeit nicht in der Tabelle aufgeführt.

Die Senken lassen sich nicht in klare Gruppen einteilen. Meist betreibt ein Gerät mehrere Senken, welche auf mehrere Informationen zugreifen. Eine besondere Rolle kommt dabei den persönlichen Geräten der Techniker zu. Sie können mobil sein, was ggf. zu Einschränkungen der Netzanbindung führen kann. Weil sie gleichzeitig Informationssenken und -quellen sind, wirken sich die Einschränkungen nicht nur auf sie selbst, sondern auch auf die Informationssenken anderer Akteure aus. Zu den Einschränkungen gehören: geringe Nutzdatenrate, hohe Verzögerung und schwankende Konnektivität. Dieser Umstand ist in Tabelle 3.4 auf Seite 51 dargestellt. Sie zeigt die Erreichbarkeit der Quellen konkreter Informationen für die bereits bekannten Senken. Dabei wird lokal oder entfernt, d. h. über LAN oder MAN bzw. WAN, auf die Quellen zugegriffen. Die möglichen Verbindungseigenschaften sind somit entweder stabil (inkl. hoher Bandbreite und geringer Latenz) oder schwankend (inkl. niedriger Bandbreite und hoher Latenz). Weil Techniker die zentralen Akteure des Szenarios darstellen, kommt der Unterstützung ihrer Mobilität folglich zentrale Bedeutung zu. Administrative Probleme beim Zugriff auf Betriebsinterne Informationen von außerhalb, d. h. Anlagenzustand, -umgebung, sowie Nutzersituation (wenn vor Ort), gelten natürlich nach wie vor. Darüber hinaus wirken sich auch dynamische Softwareinstallation und Systemkonfiguration auf die Erreichbarkeit der Quellen aus.

3.5. Quantitative Abschätzungen

Um nähere Aussagen über die technischen Probleme im Rahmen des beschriebenen Szenarios machen zu können, ist eine quantitative Abschätzung folgender Aspekte unerlässlich:

- mit dem System interagierende Akteure (z. B. Nutzer)
- vorhandene Kontextquellen
- vernetzte Knoten, welche die vorgenannten Kontextquellen verwalten
- vernetzte Knoten, die Kontextsenken betreiben
- Häufigkeit des Informationsaustauschs

Tabelle 3.3.: Kriterien und Formen des Informationszugriffs

Senke	**Informationen**						
	Technische Dokumentation	**Wartungsdokumentation**	**Kundendaten**	**Mitarbeiterprofil**	**Anlagenzustand**	**Umgebung**	**Situation**
Kundendienst	Anlage; synchron		Kunde; synchron				
Anlagenüberwachung	Anlage; synchron				Anlage; asynchron	Ort, Anlage; synchron	
Techniker	Anlage; synchron	Anlage; synchron	Kunde; synchron		Anlage; asynchron und synchron	Ort, Anlage; asynchron und synchron	
Persönliche Geräte						Ort, Person; asynchron und synchron	Person; asynchron und synchron
Koordinatoren	Anlage; synchron			Person, Anlage; synchron		Ort, Person; synchron	Person, Ort; synchron
Hersteller	Anlage; synchron					Ort, Person; asynchron und synchron	

Tabelle 3.4.: Zugriff und Erreichbarkeit der Informationen bzw. ihrer Quellen

<table>
<tr><th rowspan="2">Senke</th><th colspan="7">Informationen</th></tr>
<tr><th>Technische Dokumentation</th><th>Wartungsdokumentation</th><th>Kundendaten</th><th>Mitarbeiterprofil</th><th>Anlagenzustand</th><th>Umgebung</th><th>Situation</th></tr>
<tr><td>Kundendienst</td><td>entfernt & stabil</td><td></td><td>lokal & stabil</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Anlagenüberwachung</td><td>entfernt & stabil</td><td></td><td></td><td></td><td colspan="2">entfernt & stabil</td><td></td></tr>
<tr><td>Techniker</td><td>entfernt & schwankend</td><td colspan="2">lokal/entfernt & schwankend</td><td></td><td colspan="2">lokal/entfernt & schwankend</td><td></td></tr>
<tr><td>Persönliche Geräte</td><td></td><td></td><td></td><td></td><td></td><td colspan="2">lokal & schwankend</td></tr>
<tr><td>Koordinatoren</td><td>entfernt & stabil</td><td></td><td></td><td>lokal & stabil</td><td></td><td colspan="2">lokal/entfernt & schwankend</td></tr>
<tr><td>Hersteller</td><td>lokal & stabil</td><td></td><td></td><td></td><td></td><td colspan="2">entfernt & schwankend</td></tr>
</table>

Die nachfolgend präsentierten Zahlen stellen – analog zur Betrachtung der Informationen (Kap. 3.2) – grobe Schätzungen dar und beziehen sich, aufgrund der Verfügbarkeit entsprechenden Zahlenmaterials [Bun06, VDW08], auf Deutschland. Ohne sie ist jedoch eine Einschätzung der Skalierbarkeitsanforderungen an den Kontextdienst gänzlich unmöglich.

3.5.1. Akteure

In der Darstellung des Szenarios wurden bereits dessen wesentliche Akteure genannt – Betriebe als Kunden der dienstleistenden Wartungsfirmen und deren Techniker und Ingenieure als ausführende Organe, welche die Betreuung der Anlagen des Herstellers übernehmen. Dieser Abschnitt ermöglicht durch die Feststellung quantitativer Aussagen über die Akteure, nachfolgende quantitative Annahmen über die Informationsquellen und -senken, sowie die zwischen ihnen ausgetauschten Informationen.

Im Jahr 2004 gab es allein in Deutschland ca. 17.000 Betriebe des verarbeitenden Gewerbes*. Es muss davon ausgegangen werden, dass die Anzahl der Anlagen pro Betrieb stark schwankt. Während kleine, mittelständische Unternehmen nur wenige Anlagen einsetzen, werden in den Betrieben der Fahrzeugindustrie signifikant mehr Anlagen betrieben. Im Rahmen dieser Arbeit wird eine mittlere Ausstattung von 20 Anlagen pro Betrieb unterstellt. Auf alle Betriebe bezogen, resultiert dies in 340.000 Anlagen – allein in Deutschland.

Nimmt man den für die Instandhaltungen erforderlichen Zeitaufwand mit jährlich 3 Personentagen pro Anlage an, so führt das zu einem jährlichen Gesamtaufwand von 4 Personentagen pro Anlage†. bei ca. 44 Arbeitswochen pro Jahr, zu einem Personalbedarf von ca. 6.200 technischen Angestellten. Natürlich ist nicht davon auszugehen, dass die Anlagen aller deutschen Betriebe von einer einzigen Wartungsfirma betreut werden. Geht man von einer mittleren Belegschaftsgröße von 10 Angestellten aus, dann konkurrieren bis zu 620 Wartungsfirmen um Aufträge von Produktionsbetrieben. Es gibt aber mehrere Faktoren, welche ein einheitliches System motivieren, das in der Lage ist mit derartigen Größenordnungen zu arbeiten:

- Einsatz von offenen Standards, um sowohl Austauschbarkeit als auch Kompatibilität von Teillösungen zu erreichen.

- Der ökonomische Wettkampf der Dienstleister führt zu häufigen Umstrukturierungen der Firmenlandschaft. Dementsprechend vereinfachen einheitliche, offene Systeme die Zusammenführung unterschiedlicher Firmen.

*Laut [Bun06] Tabelle 3, Positionen D29 bis D36.

†Annahme: Instandhaltungen vor Ort und Fernwartungen beanspruchen nur 3/4 der Arbeitszeit, d.h. es gilt ein Faktor von 4/3.

Nach Angaben des Vereins Deutscher Werkzeugmaschinenfabriken e.V. [VDW08] gab es im Jahr 2006 ca. 350 Unternehmen, welche Werkzeugmaschinen und Fertigungssysteme herstellten.

Tabelle 3.5 fasst die Akteure zahlenmäßig zusammen. Es ist zu berücksichtigen, dass sich die gemachten quantitativen Annahmen auf die Bundesrepublik Deutschland beziehen. Im weltweiten Maßstab ist folglich von einer Erhöhung der Zahlen um mehrere Größenordnungen zu rechnen. Aufgrund der unsicheren Annahmen für ein weltweites Szenario werden jedoch sämtliche Betrachtungen nachfolgend auf die Bundesrepublik Deutschland beschränkt.

Tabelle 3.5.: Vorkommen der Akteure in den verschiedenen Umgebungen

Akteur	**Anzahl**
Betriebe	17.000
Wartungsfirmen	620
Techniker & Ingenieure	6.200
Hersteller	350
Summe	*24.170*

3.5.2. Informationsquellen, -knoten und -senken

Die qualitative Betrachtung der vorangegangenen Abschnitte zeigte bereits, dass die Informationsquellen aufgrund ihres spezialisierten und integrierten Charakters meist nicht leistungsfähig genug sind, selbst einen Kontextdienst zu betreiben, welche Softwaresystemen den einheitlichen Zugriff auf ihre Kontextinformationen gestattet. Deshalb schließt die Betrachtung der Quellen notwendigerweise deren Zugriffspunkte bzw. Verwaltungseinheiten ein. Diese werden nachfolgend als Informationsknoten bezeichnet. Ausgehend von der qualitativen Betrachtung der Akteure, werden in diesem Abschnitt grundlegende Annahmen über die in ihrer Umgebung existierenden Informationsquellen gemacht.

Alle betrachteten Anlagen sind Computer-gesteuert, d. h. ein mehr oder weniger leistungsfähiger Rechner überwacht anhand von Sensoren den Zustand der eigentlichen Maschine und steuert ihre Aktoren. Aufgrund der Vielfalt möglicher Informationen und der Komplexität der Anlagen, wird von durchschnittlich 50 Sensoren pro Anlage ausgegangen. Als Teil der Diagnoseelektronik, welche z. T. auf die Sensoren der Anlagensteuerung zugreift, existieren schätzungsweise weitere 10 Informationsquellen pro Anlage. Diese Quellen werden durch die Prozessrechner der Anlagen und ihre Diagnoseeinheiten verwaltet, sowie analog des Gateways eines WSNs [KW05] zugreifbar gemacht. Sind diese Rechner nicht leistungsfähig genug, muss die Ver-

waltung durch Prozessleitrechner oder andere, übergeordnete und leistungsfähigere Systeme erfüllt werden.

Techniker halten sich vorwiegend in den Betrieben oder Wartungsfirmen auf. Bei Ersteren wird eine Verteilung der 20 Anlagen auf 2 Gebäude (z.B Hallen) unterstellt. Bei Letzteren wird hingegen von der Nutzung eines einzelnen Bürogebäudes oder eine Büroetage ausgegangen. Aufgrund der vorhandenen Sensoren, kann der Steuerungsrechner eines jeden Gebäudes schätzungsweise auf durchschnittlich 20 Informationsquellen zugreifen.

In jeder Situation führt ein Techniker seine persönlichen Geräte mit sich, und nutzt ihre Dienste. Er verfügt voraussichtlich über drei solcher Geräte, welche jeweils im Mittel 10 eigene Informationsquellen (Anwendungen, Datenbanken, Sensoren, etc.) besitzen und verwalten. Über diese Geräte ist jedem Techniker der Zugriff auf weitere Informationsquellen möglich – z. B. die Spezialgeräte (ca. 5 Geräte pro Techniker).

Die Bürocomputer der Techniker in den Wartungsfirmen sind leistungsfähiger als ihre persönlichen Geräte. Folglich können auf ihnen auch mehr Anwendungen gleichzeitig (z. T. im Hintergrund) ausgeführt werden. Entsprechend ist die Zahl der Informationsquellen mit ca. 10 Stück pro Rechner höher. Im Gegensatz zu den Bürocomputern werden auf den Servern der Hersteller und Wartungsfirmen nur wenige Anwendungen betrieben, sodass die Server nur wenige potentielle Informationsquellen – ca. 5 pro Server – bereitstellen. Diese liefern jedoch eine große Menge an Informationen mit z. T. erheblicher Komplexität.

Insgesamt (Tab. 3.6 auf der nächsten Seite) müsste ein verteilter Kontextdienst, welcher von allen Akteuren betrieben wird, die Verwaltung von über 21 Millionen Informationsquellen erlauben. Dabei stünden über 740.000 Informationsknoten für den Betrieb des Kontextdienstes zur Verfügung. Wobei jeder Knoten durchschnittlich 30 Quellen verwaltet – zwischen 5 und 50 schwankend. Selbst, wenn diejenigen Quellen ignoriert werden, welche vorwiegend statische Informationen liefern, d. h. die Server der Hersteller und Wartungsfirmen, so hätte dies keine nennenswerte Auswirkung auf die Größenordnungen der verwalteten Informationsquellen und -knoten.

A 17. *Das System muss eine hohe Zahl von Informationsknoten ($> 10^5$) und -quellen ($> 10^7$) verwalten können.*

Potentielle Informationssenken werden in diesem Szenario nur von den Servern der Hersteller und Wartungsfirmen, sowie den Bürocomputern und persönlichen Geräten der Techniker betrieben. Denn diese Geräte passen sich oder ihre Daten an die Nutzungssituation an, oder erlauben ihren Nutzern die Beobachtung von Technikern oder Anlagen. Somit greift jeder der über 25.000 Knoten durchschnittlich auf über 800 Informationsquellen zu. Geht man davon aus, dass jeder Knoten im Mittel 5 Kontextsenken, d. h. adaptive System- oder Anwendungskomponenten, betreibt, dann ist mit einer Gesamtmenge von über 125.000 Kontextsenken zu rechnen.

Tabelle 3.6.: Häufigkeit von Informationsquellen und ihren Knoten

Quellen		Knoten	
Typ	**Anzahl**	**Typ**	**Anzahl**
Anlagensensoren und Diagnoseeinrichtungen	20.400.000	Prozessrechner und Diagnoseeinheit	680.000
Gebäudesensoren	692.400	Steuerungsrechner	34.620
Spezialgeräte	31.000	persönliches Gerät	18.600
Software und Sensoren der persönlichen Geräte	186.000		
Software und Sensoren der Bürocomputer	62.000	Bürocomputer	6.200
Software der Server	4.850	Server bzw. Cluster	970
Summe	*21.376.250*		*740.390*

3.5.3. Informationen und ihr Zugriff

Bevor der dynamische Datenaustausch untersucht wird, sollen zunächst die statischen Aspekte betrachtet werden, d. h. welche Datenvolumina der unterschiedlichen Informationen existieren, und ggf. zugegriffen werden können.

Datenmengen

Ausgehend von der Annahme, dass die gesamte technische Dokumentation einer Anlage 500 MB umfasst, kann das entsprechende Volumen für alle installierten Anlagen mit 170 TB* angenommen werden. Jedoch kann weder von einem Monopol auf Seiten der Hersteller, noch auf Seiten der Wartungsfirmen ausgegangen werden. Folglich lagern auf den Servern jedes Herstellers durchschnittlich fast 500 GB†, während jede Wartungsfirma auf durchschnittlich ca. 300 GB zugreift. Das resultiert in einem durchschnittlichen Datenvolumen der technischen Dokumentation für die von einem Techniker im Laufe eines Jahres bei allen Kunden betreuten Anlagen von 30 GB. Jedoch werden während einer Instandhaltung oder Fernwartung voraussichtlich wesentlich kleinere Datenbestände genutzt – z. B. die Dokumentation einer einzelnen Anlage oder der Anlagen eines Betriebes (10 GB).

Schätzt man das Volumen der Wartungsdokumentation einer Anlage auf 50 MB, so führt dass zu einem Gesamtvolumen von fast 20 TB für alle Anlagen. Sowohl bei technischen Beratungen als auch bei Instandhaltungen greifen Kundendienst bzw. Techniker auf die Wartungsdokumentation einer einzelnen Anlage oder der Anlagen

*Terabyte (TB)
†Gigabyte (GB)

eines einzigen Betriebes zu. Deshalb beschränkt sich das Volumen der potentiell notwendigen Informationen während einer einzelnen solchen Tätigkeit auf 50 MB bis 1 GB.

Bei der technischen Beratung greift der Kundendienst einer Wartungsfirma auf die Daten seiner rund 30 Kunden zu. Bei 20 MB pro Kundendatensatz, sind dies insgesamt 550 MB. Techniker greifen während einer Instandhaltung vor Ort jedoch nur auf die Daten dieses Kunden zu, d. h. auf 20 MB.

Auf den Anlagenzustand wird sowohl bei der Überwachung als auch bei der Instandhaltung zugegriffen. Im ersteren Fall überwacht eine Wartungsfirma 20 Anlagen pro Kunde bzw. 540 Anlagen insgesamt. Im zweiten Fall untersucht ein Techniker nur eine einzige Anlage. Nur selten wird ein Zugriff auf alle 20 Anlagen eines Kunden notwendig sein. Mit ca. 60 Informationsquellen pro Anlage und 10 B pro Information können pro Anlage und Zugriff 600 B Nutzdaten entstehen. Ein Zugriff bei der Instandhaltung betrifft folglich nur wenige Kilobyte. Bei der Überwachung aller Anlagen ihrer Kunden kann eine Wartungsfirma zu jedem Zeitpunkt auf über 300 kB zugreifen. Jedoch handelt es sich dabei um dynamische Daten, sodass das mittel- oder langfristig zu übertragende Datenvolumen wesentlich, d. h. um mehrere Größenordnungen, höher sein kann.

Die Überwachung der Umgebung von Anlagen ist nur durch die Gebäudeautomation und durch die persönlichen Geräte ggf. vor Ort anwesender Techniker möglich. Somit sind pro Kunde ca. 70 Informationsquellen verfügbar – 40 in den zwei Gebäuden, und 30 auf den persönlichen Geräten des Technikers. Das sind pro Kundenbetrieb entsprechend Nutzdaten von 700 B. Bei der Überwachung aller ca. 30 Kunden einer Wartungsfirma werden ca. 20 kB Nutzdaten ausgetauscht. Bei Instandhaltungen wird die Umgebung der Techniker durch deren persönliche Geräte beobachtet. Weil sie dabei nur auf ihre eigenen Informationsquellen und die der Umgebung – der Gebäude des Kunden – zugreifen können, sind in diesem Fall ebenfalls 70 Informationsquellen verfügbar. Für die Anpassung der Benutzungsschnittstellen der persönlichen Geräte ist diese lokale Beobachtung ausreichend. Müssen jedoch Dokumente auf den Servern der Hersteller oder der Wartungsfirmen vor ihrer Übertragung angepasst werden, dann wird auf die Umgebung aller 6.200 Techniker zugegriffen. Das sind ca. 4 MB Nutzdaten. Das CMMS einer Wartungsfirma beobachtet nur zehn Techniker.

Für die Koordination und somit auch während aller Arbeiten besteht Interesse an der Situation der Techniker, um jederzeit deren Verfügbarkeit ermitteln zu können. Weil sie im Normalfall Instandhaltungen durchführen, d. h. in einem Betrieb anwesend sind, stehen die gleichen Informationsquellen zur Verfügung wie bei der Überwachung der Umgebung der Techniker. Dem entsprechend greift eine Wartungsfirma auf ca. 7 kB Nutzdaten zu, d. h. 4 MB für alle Firmen.

Datenaustausch

Die Volatilität von Informationen wirkt sich auch auf das Zugriffsverhalten auf diese Informationen aus. Während statische Informationen oft nur einmalig oder selten zugegriffen werden müssen, erfordern dynamische Informationen häufige Aktualisierungen. Dieser „proportionale“ Zusammenhang wird in zwei Fällen außer Kraft gesetzt: wenn die Änderungsrate so hoch ist, dass entweder (a) die Anpassungen an veränderte Bedingungen durch zwischenzeitliche Änderungen invalidiert werden, oder (b) die Nutzung des angepassten Systems nicht lange genug möglich ist, d. h. neue Änderungen immer wieder Anpassungen verlangen. Während der erste Fall u. U. durch das adaptive System selbst beobachtet werden kann, ist im letzteren Fall nur Wissen über das Interaktionsverhalten und die Toleranz von menschlichen Nutzern hilfreich. Treten zu häufige Änderungen auf, so ist ggf. gar keine oder nur eine grob granulare Anpassung möglich. Die dafür notwendige Steuerung der Adaptivität und deren Überwachung sind jedoch nicht Gegenstand dieser Arbeit.

Grundsätzlich ist synchroner Informationszugriff unkritischer zu bewerten, als asynchroner. Letzterer kann aufgrund seines langfristigen, beobachtenden Charakters i. V. m. der Dynamik der Kontextinformationen zu einer größeren Menge von Informationsübertragungen führen. Dem entgegen ist im ersten Fall nur eine einzige Übertragung notwendig.

Tabelle 3.7.: Änderungsinduzierte, mittlere Nutzdatenraten und jährlich kumulierte Volumina der kommunizierten Informationen

Information	**Jahresvolumen V_a**	**Nutzdatenrate DR**
Technische Dokumentation	1 GB pro Anlage	250 Bit/s
Wartungsdokumentation	1,3 GB pro Anlage	330 Bit/s
Kundendaten	520 MB pro Kunde	130 Bit/s
Anlagenzustand	380 GB pro Anlage	100 kBit/s
Umgebung	370 MB pro Umgebung	100 Bit/s
Situation	370 MB pro Umgebung	100 Bit/s

Kombiniert man die Volatilität und das Volumen eines Informationstyps, so kann man das notwendige Übertragungsvolumen feststellen (Tab. 3.7), das entsteht, wenn ein Konsument eine Information über einen längeren Zeitraum beobachtet, und Änderungen zu einer vollständigen erneuten Übertragung der Information führen. Dieser Wert ist rein theoretischer Natur, jedoch wären praxisnähere, genauere Betrachtungen der Problematik noch wesentlich komplexer, ohne dass daraus allgemein gültige Aussagen abgeleitet werden könnten.

3.6. Anwendungsanforderungen

Das untersuchte Szenario hat die theoretisch begründeten Anforderungen /A 1/ bis /A 14/ bestätigt. Es wurden zusätzliche qualitative Anforderungen erkennbar, welche – neben /A 2/ und /A 4/ – weitere Teile von /A 0/ darstellen. So muss ein System für ubiquitäre Software die hohe Dynamik von Kontextinformationen und Kontextquellen behandeln, sowie eine Vielzahl von Kontextquellen verwalten muss (/A 15/ bis /A 17/; Abb. 3.2). Die Erfüllung dieser Anforderungen wird dadurch erschwert, dass die Ressourcen der mobilen Geräte stark eingeschränkt sind, und dem entsprechend geschont werden müssen /A 16/.

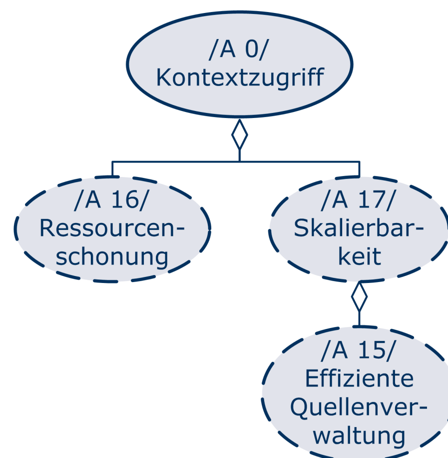

Abbildung 3.2.: Zusammenhänge zusätzlicher Anforderungen, welche durch die Charakteristika des Anwendungsszenarios bedingt sind

Die Analyse der Informationen, der Informationsquellen und der zugrunde liegenden Informationsknoten hat offen gelegt, dass eine große Heterogenität innerhalb der Informationen existiert. Einerseits gibt es großvolumige, vorwiegend statische Informationen, welche von wenigen – meist zentralen – Informationsquellen geliefert werden. Andererseits existiert eine Vielzahl weit verteilter Informationsquellen, die kompakte, jedoch z. T. sehr volatile Informationen liefern. Der Fokus soll auf letzterer Informationsklasse und der Verwaltung ihrer Quellen liegen. Weiterhin wurde erkennbar, dass eine starke Asymmetrie in Richtung der Quellen vorliegt, d. h. es gibt wesentlich mehr Informationsquellen als -senken (ca. 100:1). Somit sind kontextbezogene, adaptive Systeme grundsätzlich verschieden zu klassischen transaktionalen Systemen wie sie im Bereich der Datenbanken anzutreffen sind. Dort herrscht ein Übergewicht an Informationssenken und vorwiegend lesender Informationszugriff.

Aufgrund des Überangebots an Informationsquellen ist der Informationszugriff

der Senken äußerst selektiv. Das gilt sowohl für den Typ der Information, als auch für die räumliche und zeitliche Dimension. Genauso spezifisch wie die Nutzung der Kontextinformationen durch die Senken ist, ist auch deren Bereitstellung durch die Quellen. Eine Kontextquelle ist häufig sehr spezialisiert in der Art der Informationen die sie liefern kann, und sie liefert meist nur Informationen über einen eng umgrenzten zeitlichen und räumlichen Bereich. Bei Interesse einer Senke an einer konkreten Information ist folglich das Auffinden der Quellen, welche diese Information liefern können, sehr wichtig.

Bei der nachfolgenden Behandlung der Anforderungen soll der Fokus auf der Unterstützung des Zugriffs auf dynamische Informationen weit verteilter Kontextquellen liegen. Sie sind vor allem für die Unterstützung des Technikers, zu dessen Beobachtung und zur Überwachung seiner Umgebung und seiner betreuten Anlagen erforderlich. Die Mobilität des Technikers spielt dabei nur insofern eine Rolle, dass sie ggf. zu Verbindungsabbrüchen führen kann, und der Techniker an wechselnden Orten von wechselnden Geräten beobachtbar ist. Jedoch greift er bei seiner Arbeit häufig auf Informationen aus seiner direkten Umgebung zu, sodass die Auswirkungen der Mobilität hierbei begrenzt sind. Anders ist das bei der Beobachtung des Technikers durch seine Wartungsfirma.

Die identifizierten Anforderungen lassen sich unter den Schlagworten Effizienz und Skalierbarkeit zusammenfassen. Ausgehend von der Dynamik der Umgebungen und der Anzahl in ihnen enthaltener Geräte, werden folgende Thesen aufgestellt:

- Zur Verbesserung des Kontextzugriffs ist ein Paradigmenwechsel der Verwaltungsaktivitäten sinnvoll – weg von der Verwaltung von Kontextinformationen, hin zur Verwaltung ihrer Kontextquellen. Die Behandlung der dynamischen Verfügbarkeit von Kontextquellen durch transparente, lose Kopplung an die Kontextsenken kann die Robustheit der Kontextsenken gegenüber der schwankenden Verfügbarkeit von Kontextinformationen erhöhen.

- Die Mengen vorhandener Kontextquellen und vernetzter Geräte machen eine Abkehr vom C/S-Paradigma* erforderlich, um durch Skalierbarkeit die Funktionsfähigkeit global verteilter, ubiquitärer Software zu sichern.

*Client/Server (C/S)

4. Verwandte Arbeiten

Nicht erst seit Mark Weisers Vision des Ubiquitous Computing werden Middlewarearchitekturen entwickelt, welche den Betrieb kontextbezogener Software vereinfachen sollen. Ausgehend von unterschiedlichen Voraussetzungen wurden bei dieser Entwicklung verschiedene Strategien eingesetzt [HR06] – entweder Top-Down oder Bottom-Up. Der erste Ansatz nimmt die Anforderungen und Eigenschaften der Software zum Ausgangspunkt, um diese anschließend in einer gegebenen Umgebung zu realisieren. Das schließt ggf. auch die Ausstattung der Umgebung mit den notwendigen Kontextquellen (z. B. Sensoren) ein. Im Gegensatz dazu geht der zweite Ansatz von den technischen Gegebenheiten der Umgebung aus, und erschließt sich sukzessive eine größere Vielfalt realisierbarer Anwendungen.

Zur Unterstützung ubiquitärer Software ist der Top-Down-Ansatz weit verbreitet. Er ermöglicht die Erweiterung klassischer Anwendungen um den Aspekt der kontextbezogenen Adaptivität und die Entwicklung neuer Anwendungsklassen. Weil kontextbezogene, adaptive Software in einem großen Ausschnitt des täglichen Lebens eingesetzt werden soll, trifft sie auf heterogene Umgebungen. Eine der wesentlichen Aufgaben einer ubiquitären Middleware für derartige Anwendungen ist folglich die transparente Überwindung der Heterogenität der Umgebungen.

Während beim Top-Down-Ansatz oft einzelne, kostengünstige Sensoren in die Umgebung eingebracht werden, um diese zu beobachten und somit die Adaptivität von Software zu ermöglichen, existieren im industriellen Umfeld durchaus bereits Umgebungen, welche mit einer Vielzahl von Sensoren ausgestattet sind. Sie werden zur

Überwachung und Regelung technischer Prozesse eingesetzt, und befinden sich meist nicht von einander isoliert in der Umgebung, sondern bilden Netzwerke gleichartiger Sensoren, welche gemeinsam Überwachungsaufgaben erfüllen. Der Betrieb eines solchen Sensornetzwerkes ist sehr komplex. Dazu gehört die Koordination der Aktivität der Sensorknoten, Informationsaustausch und -verdichtung, sowie die Berücksichtigung der beschränkten Energieressourcen. Aufgrund ihres Einsatzes zur Steuerung und Regelung zeitkritischer Prozesse bestehen hohe qualitative Anforderungen an den Betrieb der Sensornetzwerke – z. B. Informationszugriff in Echtzeit. Dementsprechend sind Sensornetzwerke häufig stark an den jeweiligen Einsatzzweck angepasst und optimiert. Das führt dazu, dass Middleware für Sensornetzwerke primär deren Komplexität verbergen soll, und deshalb häufig nach dem Bottom-Up-Ansatz entwickelt wird.

Im Rahmen dieser Arbeit soll statt dessen die Unterstützung kontextbezogener, adaptiver Softwaresysteme für Ubiquitous Computing durch geeignete Middleware im Vordergrund stehen. Anstatt Sensornetzwerke gezielt zu betrachten, sollen Möglichkeiten geschaffen werden, diese – wie alle anderen Formen von Informationsquellen – unter dem Dach einer Middleware zu integrieren, und gemeinsam für kontextbezogene, adaptive Software nutzbar zu machen. Die heterogenen Landschaften von Geräten und Informationsquellen, welche im Ubiquitous Computing anzutreffen sind, gestatten keine oder nur ungenaue qualitative Aussagen über den Informationszugriff hinsichtlich Auslieferungssemantik, Zeitschranken (Echtzeitfähigkeit), etc. Wegen der daraus resultierenden Schwankungen von Verfügbarkeit und Qualität der Kontextinformationen ist ihre Verwendung für den Betrieb qualitativ anspruchsvoller Dienste problematisch.

4.1. Ubiquitous Computing Middleware und Frameworks

Im folgenden Abschnitt sollen vorhandene Mechanismen zur Systemunterstützung von ubiquitären Anwendungen vorgestellt werden. Dabei handelt es sich in erster Linie um Middleware, d. h. eine Infrastruktur, welche den Anwendungen Dienste zur Verfügung stellt und konkrete Aspekte des zugrunde liegenden Systems vor ihnen verbirgt. Ihren Ursprung hat die Systemunterstützung jedoch in so genannten Frameworks. Sie führen Abstraktionen ein und gestatten deren Nutzung durch Bereitstellung von Funktions- oder Klassenbibliotheken. Der Grad der Unterstützung ist bei einem Framework jedoch tendenziell niedriger als bei einer Middleware, weil die Entwickler des adaptiven Systems zwar vordefinierte Bausteine nutzen können, aber deren Kombination und Koordination selbst beeinflussen müssen.

Ein bekannter Vertreter der Frameworks ist das *Context Toolkit* [SDA99, Dey01]. Es führt den zentralen Begriff des *Context Widget* ein, welches eine Informationsquelle und deren Informationsgewinnung kapselt. Ein Context Widget besitzt eine

wohl-definierte Schnittstelle und repräsentiert die erfassten Informationen in einem einheitlichen Format. Die meisten der nachfolgend vorgestellten Vertreter der Middleware verfügen über eine Integrationsschicht, deren Komponenten mit Context Widgets vergleichbar sind.

4.1.1. Semantic Space und SOCAM

Semantic Space [WDC+04] und *SOCAM** [GPZ04a, GPZ04b, GPZ05] sind Systeme, welche in einer physischen Umgebung einen *semantischen Raum* etablieren. Dort können kontextbezogene Anwendungen auf Kontextinformationen zugreifen und diese austauschen. Die Namensgebung *Semantic Space* verweist auf die Erweiterung der zugrunde liegenden Idee des *Smart Space* um Mechanismen des *Semantic Web*. Ein *Smart Space* ist eine Umgebung die mit elektronischen Geräten ausgestattet ist, und welche in ihr befindliche Menschen auf intelligente Art und Weise unterstützt. Diese Unterstützung soll weitgehend automatisch erfolgen, d. h. ohne signifikante Mitwirkung der Nutzer zu erfordern, welche sie bei ihren Tätigkeiten behindern bzw. ablenken würde. Aufgrund ihrer engen inhaltlichen Verwandtschaft und ihres Ursprungs – beide entstammen der Abteilung „Context-Aware Systems“ des Institute for Infocomm Research, Singapur (I2R) – werden Semantic Space und SOCAM gemeinsam betrachtet.

Überblick

Die Arbeiten setzen drei Schwerpunkte:

1. *Explizite Repräsentation* der Kontextinformationen in einem einheitlichen Format und mit klar definierter Semantik. Dies ermöglicht Anwendungen die Nutzung der Informationen, welche von den zugrunde liegenden heterogenen Quellen ursprünglich in einer Vielzahl von Formaten bereitgestellt werden.

2. Durch gezielte *Abfrage von Kontext* sollen Anwendungen in die Lage versetzt werden, den für sie notwendigen Ausschnitt aus der Beschreibung ihrer Umgebung zu erhalten. Für diese Eingrenzung sind ausdrucksstarke Anfragen erforderlich.

3. Mittels *Kontextinferenz* können abstrakte, höherwertige Kontextinformationen† aus den erfassten Kontextinformationen gewonnen werden, welche eine bessere Beschreibung der Situation von Nutzern und Systemen erlauben. Dies führt zur Entlastung kontextbezogener Anwendungen von der Aufgabe der Kontextanalyse, ohne jedoch die Situationsabhängigkeit ihres Verhaltens zu beschränken.

*Service-Oriented Context-Aware Middleware (SOCAM)
†High-Level Context Information

Kontextontologien Für die Umsetzung von Punkt 1 wurde auf Basis der Beschreibungssprachen RDF [KC04] und Web Ontology Language (OWL) [PSHH04] eine Möglichkeit zur Kontextmodellierung entwickelt. Im Semantic Web wird OWL zur Definition von Ontologien eingesetzt die eine semantische Beschreibung von Internetseiten erlauben. Dementsprechend definiert SOCAM Kontextontologien, welche die ausgetauschten Kontextinformationen repräsentieren. Um eine umfassende Anwendbarkeit der Ontologien auf verschiedenste Gebiete und ihre Kontextquellen zu erlauben, wird eine Hierarchie von Ontologien vorgeschlagen. Die Basis dieser Hierarchie ist die so genannte Upper Level Context Ontology (ULCO) – eine abstrakte High-Level Ontologie. Sie definiert grundlegende Konzepte, welche in den meisten Anwendungsgebieten anzutreffen sind. Neben der ULCO können beliebig viele domänenspezifische Ontologien definiert werden, welche die ULCO gemeinsam nutzen und sie um zusätzliche Konzepte erweitern.

Infrastruktur Um kontextbezogenen Anwendungen die Ausdrucksstärke der Kontextontologien zur Verfügung zu stellen (Punkt 2 und 3), ist ein geeignetes Inferenzsystem (logikbasierter Reasoner) notwendig, welcher eine Wissensbasis (engl. knowledge base) aufbaut und verwaltet. Derartige Komponenten sind auch Teile von SOCAM (Abb. 4.1 auf der nächsten Seite) – hier *Context Reasoner* und *Context Knowledge Base* genannt – und bilden den Kern des *Context Interpreter*. Kontextinformationen fragt er von den *Context Providern* ab, oder abonniert sie, um daraus abstrakte Kontextinformationen zu inferieren. *Context Provider* kapseln die heterogenen Kontextquellen und stellen deren Informationen als OWL-Repräsentation bereit. *Context-aware Applications* sind die Klienten des Systems und greifen direkt über die Context Provider oder indirekt über den Context Interpreter auf Kontextinformationen zu. Anwendungen können anwendungsspezifische Regeln definieren, welche der Context Interpreter evaluiert und anhand der Ergebnisse die Anwendungen benachrichtigt. Die dabei inferierten Kontextinformationen werden jedoch, anders als die Inferenzergebnisse des Context Reasoners, nicht in der Context Knowledge Base abgelegt. Die *Context Database* wird zur Speicherung von Ontologien und historischen Kontextinformationen eingesetzt. Für jede domänenspezifische Ontologie verwaltet das System eine eigene Context Database. Der *Service Locating Service (SLS)* dient als Verzeichnisdienst des SOCAM-Systems, und ermöglicht die Kooperation aller Komponenten, sowie die Suche von Anwendungen nach Diensten. Zu ihnen zählen in erster Linie die Context Provider und der Context Interpreter, welcher ebenfalls als ein generischer, abstrakter Context Provider betrachtet werden kann.

Die Architektur von Semantic Space ähnelt stark derjenigen von SOCAM, ist jedoch in einzelnen Punkten nicht so klar strukturiert*.

*Wegen fehlender Hintergrundinformationen wird davon ausgegangen, dass es sich bei Semantic

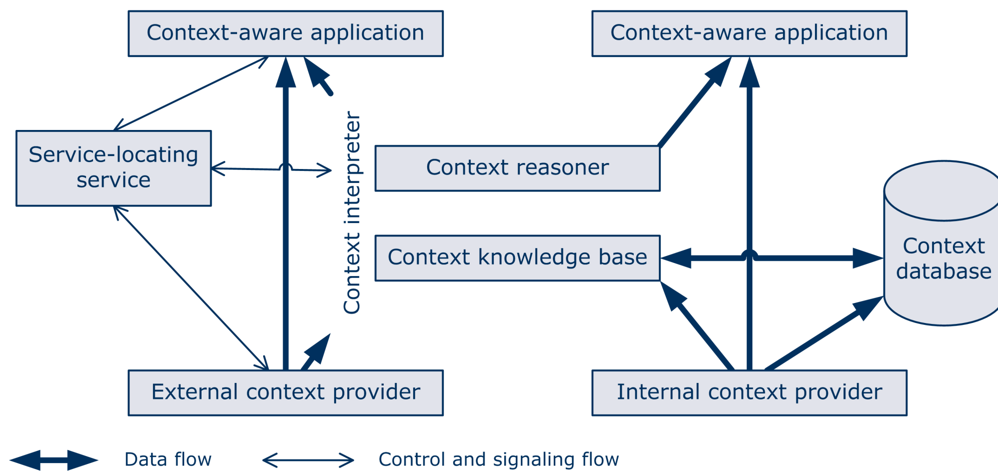

Abbildung 4.1.: Überblick der Architektur von SOCAM (nach [GPZ04b])

Verzeichnisdienst Dem SLS [GPY05] kommt eine bedeutende Rolle zu, weil er den Komponenten von SOCAM und den Anwendungen die Suche von Diensten gestattet. So kann u. a. der Context Interpreter feststellen welche Context Provider existieren, um sie entsprechend einbinden zu können.

Beim SLS handelt es sich um einen verteilten Dienst. Jedes Gerät das die SOCAM-Middleware betreibt, verfügt auch über eine Instanz des SLS. Es gibt zwei Ausführungen des SLS – Client und Server. SLS-Clients melden automatisch ihre Existenz, um von einem SLS-Server gefunden zu werden, und anschließend ihre Dienste registrieren zu können bzw. die Dienste der übrigen Geräte zu erfragen. Zur Verbesserung der Skalierbarkeit des SLS wird bei Bedarf eine Hierarchie von Servern aufgebaut. Dies erfolgt dergestalt, dass ein SLS-Server im Fall der Überlastung – z. B. durch Beitritt eines weiteren SLS-Clients – einen der SLS-Clients dazu veranlasst, ebenfalls als SLS-Server zu arbeiten. Dieser neue Server befindet sich auf der gleichen Ebene des Baumes, wie der Überlastete. Um als Server arbeiten zu können, lädt der Client zuvor die notwendigen Zusatzfunktionen aus dem Netzwerk. Neben dem neuen, gleichrangigen Server bestimmt der überlastete Server ggf. einen übergeordneten Server, welcher in der nächsthöheren Ebene der Hierarchie für den überlasteten und den neuen Server zuständig ist. Anfragen von Clients werden in der Hierarchie propagiert, dass alle Dienste, die einer Anfrage entsprechen, gefunden werden. Hierzu speichert jeder Server die Dienste seiner untergeordneten Knoten. Um den dafür erforderlichen Speicheraufwand in den höheren Ebenen der Hierarchie zu beschränken, werden die Dienstbeschreibungen verdichtet. Die Autoren von

Space um einen Vorgänger von SOCAM handelt.

SOCAM nennen dieses Verfahren *Service Aggregation*. Es basiert auf einem hierarchischen Klassifikationschema, welches zur Typisierung der Dienste eingesetzt wird. Ist z. B. auf einem Server ein Dienst vom Typ „mServiceAll.mInformation.weather“ registriert, so existiert auf seinem übergeordneten Server ein Eintrag mit dem Typ „mServiceAll.mInformation“ der u. a. auf den untergeordneten Server verweist.

Weil SOCAM unterschiedliche Diensttechnologien unterstützt, muss der SLS auch verschiedene Formen der Dienstbeschreibung verarbeiten können. Dienste können entweder mittels Name-Wert-Paaren oder semantisch* beschrieben werden. Sogar die Beschreibung anhand eines Java-Klassennamens ist möglich. Um diese drei Beschreibungsformen zu interpretieren, besitzt jeder Server drei *Matching Engines*. Sucht ein Client über den SLS nach einem Dienst, so muss er sich zuerst für eine der Beschreibungsformen entscheiden, und außerdem angeben, ob die Beschreibung in eine der anderen Formen konvertiert werden darf. Letzteres ist nur dann sinnvoll, wenn der Client auch die entsprechende technologische Unterstützung für den Dienstzugriff bietet.

Einschätzung

Die Entwicklung der ULCO als Basisontologie für die Beschreibung von Kontextinformationen stellt ein interessantes Konzept dar. Ob sie jedoch hinreichend allgemein gültig ist, und einen entsprechenden Nutzwert birgt, kann erst durch die Feststellung ihrer Akzeptanz in der Forschergemeinde und der Industrie eingeschätzt werden. Weil aber im Bereich des Pervasive und Ubiquitous Computing derzeit noch keine akademische Lösung eine dominierende Stellung besitzt, ist eine Entscheidung zu diesem Zeitpunkt nicht möglich.

Die direkte Kopplung zwischen kontextbezogenen Anwendungen und den Context Providern birgt erhöhten Aufwand bei der Realisierung der Anwendungen. So sind diese selbst sowohl für das Binden passender Context Provider, als auch für die Kommunikation mit diesen zuständig. Auftretende Verbindungsprobleme müssen explizit durch die Anwendungen behandelt werden. Darüber hinaus müssen die Anwendungen die technologische Heterogenität der Provider überwinden indem sie entweder sich auf die Nutzung einer Technologie beschränken, oder alle Technologien parallel unterstützen. Somit besteht die Wahl zwischen eingeschränkter Informations- bzw. Quellenvielfalt und erhöhter Anwendungskomplexität. Diese Probleme können nur dann umgangen werden, wenn die Anwendungen ausschließlich über den Context Interpreter auf Kontextinformationen zugreifen, weil dieser lediglich eine Form des Zugriffs bietet. In diesem Fall stünden den Anwendungen jedoch nur abstrakte Kontextinformationen für ihre Adaptivität zur Verfügung.

*durch Konstrukte der Sprache DARPA Agent Markup Language – Ontology Inference Layer (DAML-OIL) [CvHH+01, Hor02]

Aufgrund seiner Alleinstellung ist der Context Interpreter die zentrale Schwachstelle des Systems. Er beschränkt dessen Skalierbarkeit und Verfügbarkeit. Weiterhin ist die Verarbeitung volatiler Daten mittels logikbasiertem Reasoning als problematisch einzuschätzen [Eic07], weil dieses zeitaufwendig ist, und aktuelle Reasoner inkrementelles Reasoning nur bedingt unterstützen. Bei der Änderung von Informationen in der Context Knowledge Base ist eine erneute Inferenz über dem vollständigen Datenbestand der Context Knowledge Base durchzuführen. Nicht zuletzt fragt der Context Interpreter pauschal die Kontextinformationen aller Context Provider ab. Somit erscheint SOCAM für den Einsatz in größeren Umgebungen mit einer Vielzahl von Geräten und Kontextquellen weniger geeignet.

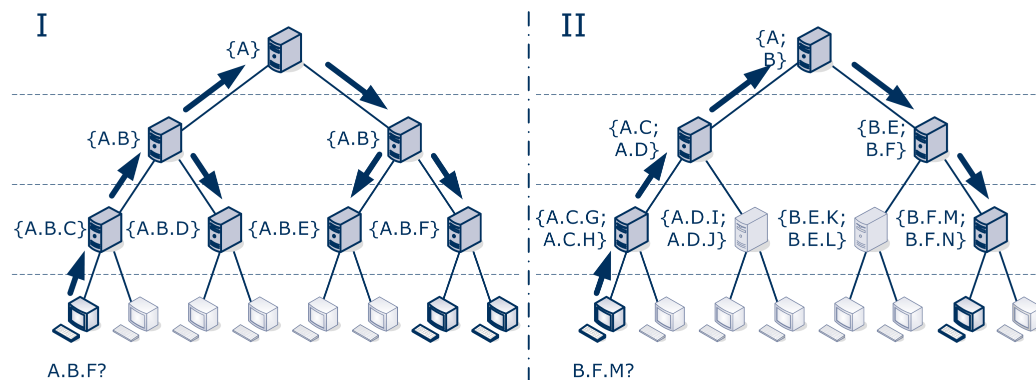

Abbildung 4.2.: Auswirkung homogener und heterogener Dienstlandschaften eines Semantic Space auf die Anfrageweiterleitung zwischen SLS Servern (angelehnt an [GPY05])

Die Praktikabilität der Service Aggregation ist nur dann gegeben, wenn die unterschiedlichen Dienstklassen des Semantic Space nicht homogen in diesem verteilt sind, oder wenn zur Anfragebearbeitung neben der Dienstklasse weitere Kriterien herangezogen werden (z. B. Ort des Dienstes). Zur Veranschaulichung sind in Abbildung 4.2 Beispiele für das Vorkommen von Dienstklassen in einem Semantic Space dargestellt. Auf beiden Seiten ist eine Hierarchie von Rechnern abgebildet, deren drei obere Schichten durch SLS Server gebildet werden. Die unterste Schicht bilden SLS Clients. Links neben jedem Server ist die Menge der Dienstklassen vermerkt, welche seine untergeordneten Knoten – Clients oder Server – bei ihm registriert haben. Aufgrund der Service Aggregation nimmt der Detaillierungsgrad dieser Registrierungsinformationen in Richtung der oberen Hierarchieebenen ab. Im Semantic Space von Fall I (Abb. 4.2, links) existiert eine Landschaft sehr ähnlicher Dienste – alle gehören der Klasse `A.B` an. Sucht ein Client nach einem Dienst der Klasse `A.B.F` so müssen die zugehörigen Anfragen zwischen den Servern über große Teile

der Hierarchie verteilt werden, bevor sie Server erreichen, welche die Anfrage beantworten oder verwerfen können. Der Semantic Space von Fall II (Abb. 4.2, rechts) beherbergt eine große Vielfalt von Diensten, welche nicht ungleichmäßig über die Knoten verteilt sind. Ist ein Client an einem Dienst der Klasse `B.F.M` interessiert, so ist während der Baumsuche eine frühzeitige Entscheidung über die Weiterleitung der Anfrage möglich. Die höhere Selektivität der Anfrage ist – unbeeinflussbar für den Client – nicht in ihrer Ausdrucksstärke begründet, sondern in der Heterogenität der Dienstlandschaft. SOCAM bevorzugt somit heterogene Dienstlandschaften, sowie SLS Clients, welche ausdrucksstarke Anfragen auf Basis von DAML-OIL* formulieren können.

4.1.2. Context Broker Architecture

Die Context Broker Architecture (CoBrA) [CFJ03a, CFJ03b, CFJ03c, CFJ03d, CFJ04a, CFJ04b] ist ebenfalls für den Einsatz in Smart Spaces entwickelt. Jedoch wird hier der Aspekt der ontologiebasierten Kontextrepräsentation stärker betont.

Überblick

CoBrA ist eine Middleware, welche auf der Technologie mobiler Agenten basiert. Diese Agenten sind Softwarekomponenten, die zwischen den Geräten migrieren können. Damit diese Migration überhaupt möglich ist, muss jedes Gerät, welches Agenten betreibt, über eine spezielle Ausführungsumgebung verfügen. Im Rahmen von CoBrA wird für diesen Zweck das Java Agent Development Framework (JADE) [BCPR03] eingesetzt. Die meisten Agenten werden auf den Geräten des Smart Space ausgeführt und gestatten den Zugriff auf die Kontextinformationen der dort vorhandenen Kontextquellen. Die wichtigste Aufgabe kommt bei CoBrA dem so genannten *Context Broker* zu – einem speziellen Agenten – der auf einem leistungsfähigen Rechner ausgeführt wird und die Rolle eines Servers übernimmt. Kontextbezogene Anwendungen greifen über den Context Broker auf Kontextinformationen zu.

Context Broker Der Context Broker hat mehrere Aufgaben:

1. Etablierung eines zentralen, einheitlichen Modells aller Kontextinformationen des Smart Space
2. Abfrage der Kontextinformation von Quellen, welche für leistungsschwache Geräte unerreichbar sind

*Die Nutzung von DAML-OIL zur Abfrage von Informationen wird in den Quellen erwähnt, korrespondiert jedoch nicht mit der Definition von OWL-basierten Ontologien. Letztere legen viel mehr die Nutzung der Resource Description Query Language (RDQL) nahe.

3. Inferenz von abstrakten Kontextinformationen die nicht von Geräten oder Sensoren abgefragt werden können

4. Erkennung und Beseitigung von Inkonsistenzen der Kontextinformationen aus den verschiedenen Kontextquellen des Smart Space

5. Schutz der personenbezogenen Kontextinformationen der Nutzer

Die erste Aufgabe erfüllt der Context Broker durch seine zentrale Rolle und die Architektur des Systems – Kontextinformationen sind nur von ihm erhältlich. Für die Datenhaltung verfügt der Broker über eine *Context Knowledge Base* in der alle Kontextinformationen persistent gespeichert werden. Das *Context Acquisition Module* erfüllt die zweite Aufgabe indem es eine Middleware zur Erfassung der Kontextinformationen zur Verfügung stellt. Teil dieser Middleware ist auch eine Bibliothek, welche die Implementierung entsprechender Erfassungskomponenten der Agenten gestattet. Die erfassten Kontextinformationen werden in der Knowledge Base abgelegt. Mit diesen Voraussetzungen kann die *Context Reasoning Engine* eingesetzt werden, um die dritte und vierte Aufgabe zu behandeln. Sie setzt das JENA Semantic Web Framework* ein und arbeitet auf dem Datenbestand der Knowledge Base. Zur Erfüllung der fünften Aufgabe existiert das *Module for Privacy Protection*. Es besteht aus einer Menge von Regeln, welche den Zugriff auf personenbezogene Kontextinformationen (z. B. den Aufenthaltsort eines Nutzers) steuern. Diese Regelmenge wird ebenfalls von der Reasoning Engine ausgewertet. Die Menge vordefinierter Regeln kann durch die Nutzer um eigene Regeln erweitert werden.

Verteilung Um die Verfügbarkeit des Brokerdienstes zu verbessern, soll der gleichzeitige Einsatz mehrerer Broker in einem so genannten *Persistent Broker Team* möglich sein [KCL00]. Die Mitglieder eines solchen Teams sind dafür zuständig, dass jederzeit eine Mindestanzahl von Brokern die Anfragen von kontextbezogenen Anwendungen beantwortet. Sinkt die Anzahl unter eine definierte Schwelle, so soll das Team neue Mitglieder bestimmen. Durch das Teamkonzept soll ebenfalls eine Föderation möglich sein, welche die Kopplung mehrerer Broker erlaubt. Auf diese Weise sollen auch räumlich größere Spaces realisierbar sein – z. B. der Campus einer Universität. Die Broker müssen regelmäßig kommunizieren, um ihre Context Knowledge Bases zu synchronisieren.

COBRA-ONT Für Modellierung von Kontextinformationen existiert in CoBrA die Ontologie COBRA-ONT. Sie gestattet die Repräsentation wesentlicher Aspekte der Umgebung (Zeit und Ort), des Systems (Agenten – inkl. ihrer Orte und Aktivitäten)

*Erste Versionen von CoBrA besaßen einen speziell entwickelten OWL-Reasoner der in der Sprache Flora-2 [YKZ03] implementiert war.

und der Zugriffsregeln. Im Gegensatz zu SOCAM entwirft CoBrA keine explizite Hierarchie von Ontologien.

Einschätzung

Der Einsatz von Ontologien ist generell als vorteilhaft zu betrachten, weil er eine generische Beschreibung von Kontextinformationen erlaubt. Sofern die entwickelten Ontologien jedoch nicht hinreichend akzeptiert sind – analog zu SOCAM – stellen sie lediglich proprietäre Modelle dar, welche nicht in der Lage sind, die Interoperabilität des Systems mit vorhandener Infrastruktur zu fördern.

Die zentralisierte Architektur impliziert, dass der Context Broker die Schwachstelle des Systems darstellt. Dementsprechend sind seine Verfügbarkeit und Skalierbarkeit als kritisch einzuschätzen. Den Autoren ist der erstere Aspekt bewusst. Leider ist der Vorschlag des Persistent Broker Team auch nur zur Behandlung dieses Punktes geeignet. Das Problem der Skalierbarkeit würde durch diesen Ansatz weiter verstärkt, weil die regelmäßige Synchronisation der Knowledge Bases aller Broker zur zusätzlichen Belastung der Kommunikations- und Verarbeitungsressourcen führt.

4.1.3. Gaia

Das Projekt Gaia [RC00, CHRC01, RHR+01, RHC+02] beschäftigt sich mit der Entwicklung einer Middleware für Ubiquitous Computing. Gaia führt weitere Abstraktionen im Konzept des Smart Space ein, um ein komponentenbasiertes, verteiltes Metabetriebssystem zu realisieren.

Überblick

Active Space Ein *Active Space* [RC00] bildet eine klar umgrenzte, physische Umgebung – den *Physical Space* – mit all ihren Geräten, Gegenständen und Akteuren – kurz: Entitäten – auf ein Modell und eine Software ab. Er bietet ein generisches Modell dieser Umgebung über ein generisches Application Programming Interface (API) an. Somit wird ein Active Space in Gaia als eine spezielle Form eines klassischen IT-Systems verstanden. Die *Hardware* des Active Space sind die abstrakten Repräsentationen der Ressourcen der physischen Umgebung – Prozessoren, Speicher (transient und persistent) und Peripheriegeräte. Dem *Betriebssystem* kommt die Aufgabe der Vereinigung der physischen Umgebung mit dem Active Space zu. Dabei muss das Betriebssystem eine generische Infrastruktur sein, welche die Ressourcen der Umgebung verwaltet – ebenso wie dies der Scheduler und die Speicherverwaltung eines klassischen Betriebssystems vornehmen. Elektronische Geräte und Anwendungen können entweder stellvertretend für ihre menschlichen Nutzer oder autonom auf die Ressourcen des Active Space zugreifen. Weil sie sich

dabei selbst in der physischen Umgebung des Active Space befinden und selbst Ressourcen und Informationen zur Verfügung stellen können, sind sie sowohl als Nutzer als auch als Betreiber des Active Space zu betrachten.

Architektur – GaiaOS Gegenstand der Arbeiten des Gaia-Projektes ist ein Betriebssystem für die Etablierung eines Active Space – GaiaOS. Es wird auf den jeweils lokal existierenden Betriebssystemen ausgeführt und nutzt die dort vorhandene komponentenbasierte, verteilte Middleware (z. B. die Common Object Request Broker Architecture (CORBA)). Dementsprechend besteht eine der grundlegenden Aufgaben von GaiaOS in der Kapselung der heterogenen Middleware zur Schaffung einer einheitlichen Umgebung für verteilte Komponenten. Diese Aufgabe übernimmt der *Unified Object Bus (UOB)* (Abb. 4.3 auf der nächsten Seite). Er nutzt die Kommunikationsmechanismen einer oder mehrerer Middlewareplattformen – hier als *Object Request Brokers* bezeichnet – und bindet sie mittels spezifischer *Integrators* ein. Der *Component Container* ist die Ausführungsumgebung der Gaia-Komponenten (oder *Unified Components*), welche unter Zugriff auf die Basisfunktionalität des *Component Management Core* realisiert werden können.

Mit diesen Voraussetzungen definiert GaiaOS sowohl Basis- als auch optionale Dienste. Sie bilden den *Gaia Kernel* bzw. die *Gaia Services*. Im *Space Repository* sind sämtliche Entitäten des Active Space registriert. Es gestattet die Suche von Entitäten anhand von Typinformationen und anderen Kriterien mittels einer speziellen Anfragesprache. Der *Presence Service* wiederum sorgt für die Überwachung der Komponenten, welche die registrierten Entitäten repräsentieren. Zum Austausch der dafür notwendigen Heartbeat-Nachrichten wird der *Event Manager Service* verwendet. Er ermöglicht die lose Kopplung der Komponenten des Active Space über Nachrichtenkanäle, die definiert und abonniert werden können. Zusätzlich zu dieser anonymen, generischen Form der Kommunikation ist auch noch die Kommunikation innerhalb einer Gruppe bereits bekannter Komponenten möglich.

Das *Gaia Application Model* stellt durch den *Application Management Service* Funktionen zur Verfügung, welche das Registrieren, Suchen, Aktualisieren und De-Registrieren von Anwendungen, sowie das Austauschen von Ereignissen erlauben. Teil des Application Model ist MPACC* – eine erweiterte Version des MVC†-Entwurfsmusters, welches die adaptive Präsentation von Inhalten auf heterogenen Ausgabegeräten ermöglichen soll. Das *Active Space Execution Environment* repräsentiert schließlich die Komponenten der Anwendungen des Active Space, die über dessen Schnittstellen homogen auf den Active Space zugreifen.

*Model-Presentation-Adapter-Controller-Coordinator (MPACC)
†Model-View-Controller (MVC)

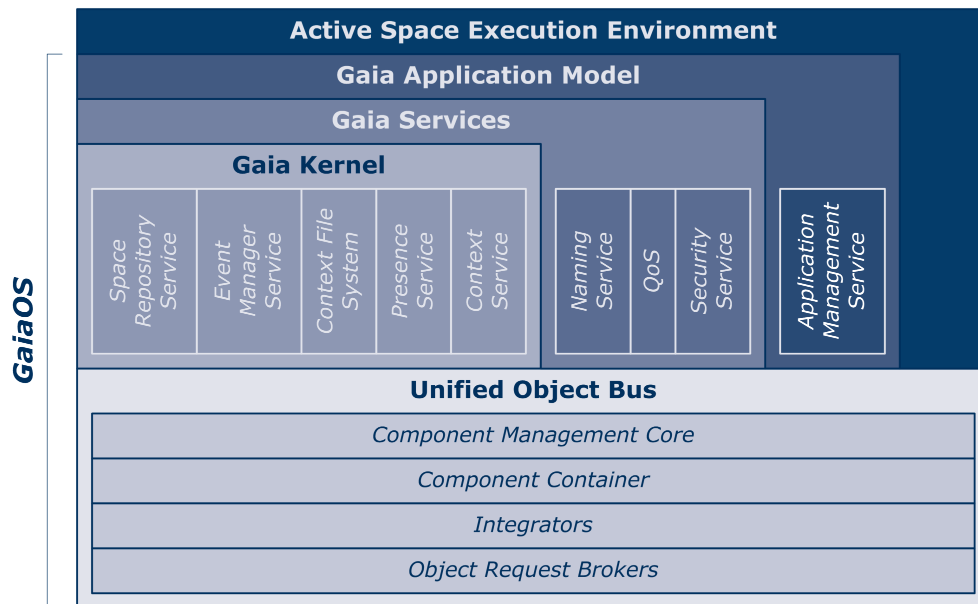

Abbildung 4.3.: Architektur von GaiaOS (angelehnt an [RC00, CHRC01, RHR+01, RHC+02])

Context Service Aufgrund der zentralen Bedeutung von Kontextinformationen beim Betrieb eines Active Space ist ein Context Service Teil des Gaia Kernel. Kontextquellen werden von ihm in so genannten Context Providern gekapselt – analog den *Generators* bzw. *Context Widgets* des *Context Toolkit* [SDA99, Dey01]. Ihre Kontextinformationen können abgefragt oder abonniert werden. Der eigentliche Austausch findet über den Event Manager Service statt. Die verfügbaren Kontextinformationen werden u. a. vom *Context File System* genutzt. Dabei handelt es sich um ein kontextbezogenes Dateisystem, welches einerseits den Anwendungen jederzeit ortstransparenten Zugriff auf ihre Dateien ermöglicht, und andererseits das Format dieser Dateien bei Bedarf an die Anforderungen der Anwendung adaptiert. Neben dem Context Service kann auch das *Space Repository* Umgebungsinformationen des Active Space liefern. Die dort registrierten virtuellen Entitäten erlauben den Rückschluss auf die repräsentierten physischen Entitäten.

Erweiterungen Mit *Mobile Gaia* [CAMCM05] existiert eine Variante von GaiaOS zur Realisierung so genannter *Personal Active Spaces*. Ein solcher Space besteht aus einem Cluster der persönlichen, mobilen Geräte des Nutzers, welche durch ein Wireless Personal Area Network (WPAN) verbunden sind. Ein Cluster verfügt immer über ein Gerät, welches den Cluster kontrolliert – den *Coordinator*. Die übrigen

Geräte werden als *Clients* bezeichnet. Jedes Gerät im Cluster bzw. Space betreibt einen minimalen Gaia Kernel. Dieser unterscheidet sich vom klassischen Gaia Kernel durch drei zusätzliche Dienste. Analog zur den Geräten des Clusters kann jede Dienstinstanz auf einem Gerät entweder als *Coordinator* oder als *Client* betrieben werden – unabhängig von der Rolle des Gerätes. Der *Discovery and Cluster Management Service* findet benachbarte Geräte, verhandelt ihren Beitritt zum Cluster und steuert diesen. Das *Service Deployment Framework* enthält die Dienste des Clusters und entscheidet auf jedem Gerät über die Rolle der lokalen Dienstinstanzen. Der *Location Service* aggregiert mittels probabilistischer Methoden die Positionsinformationen der Kontextquellen des Clusters zu einer einzigen Position des Clusters – und somit zu der seines menschlichen Nutzers.

Neben der Unterstützung persönlicher Active Spaces existiert auch eine Arbeit zur weiteren Generalisierung der Kommunikationsinfrastruktur [CAMCM04]. Sie skizziert ein *Service Interaction Layer* das oberhalb des UOB die Kopplung der auf dem UOB basierenden Dienste erbringt. Für diese Funktionalität bietet die Schicht die Kommunikationsparadigmen *Distributed Querying*, *Distributed Events* und *Direct Access* an. Sie basieren auf den Kommunikationsprinzipien des bereits vorgestellten Event Manager Service – themenbasiertes Publish/Subscribe und Punkt-zu-Punkt-Kommunikation.

Für die Kopplung mehrerer Spaces existiert ein Ansatz namens *Super Space*. Dabei werden benachbarte oder sich überlappende Active Spaces entweder hierarchisch oder gleichberechtigt, d. h. mittels P2P*-Kommunikation, angeordnet. So soll eine Abbildung der organisatorischen oder physischen Strukturen auf die virtuellen Strukturen des Super Space möglich sein.

Einschätzung

GaiaOS skizziert eine ambitionierte Vision – die homogene Kopplung sämtlicher elektronischen Geräte in der Umgebung menschlicher Nutzer durch ein komponentenbasiertes, verteiltes Metabetriebssystem bzw. eine Middleware. Jedoch bleiben die Arbeiten den Nachweis der Praktikabilität des Systems schuldig. So wird die Integrationsfähigkeit des UOB dadurch belegt, dass Instanzen der CORBA-Implementierungen *dynamicTAO* und *LegORB* erfolgreich miteinander gekoppelt werden. Aufgrund der Plattformunabhängigkeit von CORBA sollte dies ohnehin möglich sein, und genügt nicht als Existenzberechtigung für eine weitere Middleware wie GaiaOS.

Unabhängig von der Integration existierender Middleware gelingt es den Autoren nicht, die zugrunde liegenden Mechanismen und Algorithmen mit einem Detailgrad zu beschreiben, welcher ihre theoretische und praktische Einschätzung gestattet. In

*Peer-to-Peer (P2P)

diesem Rahmen ist es ebenfalls als problematisch zu betrachten, dass in mehreren Veröffentlichungen unterschiedliche Mengen von Diensten des Gaia Kernel und der Gaia Services mit abweichender Funktionalität präsentiert werden. So entsteht kein homogenes Bild des Systems.

Der für diese Arbeit interessante Context Service wird nur in einem kleinen Abschnitt einer einzigen Veröffentlichung erwähnt. Die aufgeführten Eigenschaften – synchroner und asynchroner Kontextzugriff, Kapselung von heterogenen Kontextquellen und logikbasierte Anfrageschnittstelle – ähneln denen konkurrierender Systeme. Die für den Kontextzugriff und -austausch notwendige Kommunikation erfolgt über die Paradigmen des Event Manager Service. Für asynchrone Kommunikation setzt dieser themenbasierte Nachrichtenkanäle, d. h. Subject-based Publish/Subscribe, ein. Aufgrund der geringen Flexibilität dieses Ansatzes können kontextbezogene Anwendungen nur grob-granular Kontextinformationen abonnieren. Dies wird den spezifischen Anforderungen der einzelnen Anwendung nicht gerecht und führt zum unnötigen Austausch ungenutzter Kontextinformationen. Wegen fehlender Informationen über die Realisierung dieser Kommunikationsform kann deren Skalierbarkeit nicht abschließend beurteilt werden.

Grundsätzlich ist die isolierte Betrachtung einzelner Bereiche der physischen Umgebung problematisch. Diese Sichtweise wird in [RHC+02] erkennbar:

> „A physical space ... is a geographic region with limited and well-defined physical boundaries containing physical objects, heterogeneous networked devices, and users performing a range of activities."

Aufgrund der klaren Grenzen eines Physical Space ist es wahrscheinlich, dass ein mobiler, menschlicher Nutzer diesen und seinen zugehörigen Active Space verlässt, und in einen anderen Active Space eintritt. Folglich sind Lösungen für diesen Übergang und zur Unterstützung von Mobilität erforderlich. [CAMCM05] entwickelt persönliche Active Spaces, welche sich mit ihren Nutzern bewegen können. Wie dabei die Ausdehnung der Active Spaces kontrolliert wird, ist jedoch unklar. Weiterhin ist zu erwarten, dass ein Nutzer nicht nur eine einzelne Anwendung nutzt. Diese Anwendungen existieren in unterschiedlichen Active Spaces, welche keinerlei physische Kohärenz haben müssen. In [AMCRC04] wird aufgezeigt, dass es mehrere Active Spaces geben kann, und dass diese sowohl in Hierarchien, als auch gleichberechtigt nebeneinander existieren können. Wie die konkrete Koordination zwischen den Active Spaces erfolgt, bleibt aber unerkennbar.

4.1.4. Hydrogen

Das *Hydrogen*-System [HSP+03] ist explizit für den Einsatz auf mobilen und integrierten Geräten vorgesehen.

Überblick

Um die Heterogenität der Kontextquellen vor den Anwendungen zu verbergen, besitzt Hydrogen eine dreischichtige Architektur. In der untersten Schicht – der *Adaptor Layer* – kapseln Adapter die vorhandenen Kontextquellen des mobilen Gerätes. Sie ermitteln Low-Level* Kontextinformationen durch regelmäßiges Polling. Während dieser Gewinnung werden aus den Kontextinformationen ggf. High-Level† Kontextinformationen gewonnen, und den Low-Level Kontextinformationen hinzugefügt. Die zentrale Schicht heißt *Management Layer* und sorgt für die Kopplung der darüber bzw. darunter befindlichen Schichten. In ihr befindet sich der *Context Server*, welcher die Kontextinformationen der Umgebung des mobilen Gerätes speichert. Er ist ebenfalls für den Informationsaustausch mit anderen Dienstinstanzen auf weiteren mobilen Geräten zuständig. Für die oberste Schicht – die *Application Layer* – bietet er den Kontextzugriff, sowohl in synchroner als auch in asynchroner Form an.

Kontextinformationen werden als Objekte der Programmiersprache Java repräsentiert und gespeichert. Hydrogen verfügt über eine Menge vordefinierter Klassen zur Repräsentation spezifischer Kontextinformationen mit definierter Semantik. Die Erweiterung des Typsystems um anwendungs- oder domänenspezifische Aspekte ist durch Deklaration neuer Klassen möglich, welche Unterklassen der bereits existierenden Klassen sein müssen. Der Context Server verwaltet mehrere Mengen von Kontextinformationen. Während der *Local Context* die Kontextinformationen der Kontextquellen des lokalen Gerätes enthält, beherbergen ein oder mehrere *Remote Contexts* jeweils die Kontextinformationen eines entfernten Gerätes. Der Informationsaustausch zwischen den Schichten der lokalen Architektur und den verteilten Instanzen der Context Server erfolgt über XML-codierte Nachrichten. Um Anwendungen von der Realisierung dieser nachrichtenbasierten Kommunikation zu befreien, verfügt die Application Layer über eine spezielle Klasse, welche analog zu den unterschiedlichen Nachrichten entsprechende Methoden für den Kontextzugriff anbietet.

Einschätzung

Die vorgeschlagene Architektur gestattet ein dynamisches Binden von Kontextquellen, welches transparent vor den Kontextsenken durch den Context Server realisiert werden kann. Dabei ist sogar transparent, ob die gelieferten Kontextinformationen vom lokalen oder einem entfernten Gerät stammen. Die Relationen zwischen Low-Level und High-Level Kontextinformationen sind in den jeweiligen Adaptern verborgen. Diese können weiterhin transparent diverse Inferenzmechanismen einsetzen.

*Bei Hydrogen als *physical context* bezeichnet.

†. . . oder *logical context*

4.1.5. JCAF

Ziel des Java Context-Awareness Framework (JCAF) [Bar05] ist die Bereitstellung einer leichtgewichtigen, ausdrucksstarken Menge von Schnittstellen, welche die einfache Entwicklung kontextbezogener Anwendungen erlaubt. Darüber hinaus bietet JCAF Middlewarefunktionen zur Kontexterfassung, -verwaltung und -verteilung. Instanzen der Middleware werden auf den Geräten der Umgebung installiert. Die Kopplung dieser Instanzen erfolgt in einer hierarchischen oder P2P-Struktur – je nach Einsatzszenario – sodass ein verteilter Kontextdienst entsteht.

Überblick

Die Architektur von JCAF besteht aus drei Schichten: *Context Client Layer*, *Context Service Layer* und *Context Monitor and Actuator Layer*. Die Context Client Layer beherbergt die kontextbezogenen Anwendungen, welche auf ein oder mehrere Context Service Instanzen zugreifen, um Kontextinformationen zu erhalten. JCAF unterstützt sowohl synchronen als auch asynchronen Kontextzugriff. Bei letzterem kann ein Client entweder eine einzige konkrete Kontextinformation (hier Entität genannt) oder alle Entitäten eines konkreten Typs beobachten. Clients können sogar die Topologie der Vernetzung der Context Service Instanzen steuern.

In der Context Service Layer befindet sich eine Context Service Instanz pro Gerät. Sie gestattet den Clients den Kontextzugriff über ein API und führt während dessen eine Zugriffskontrolle durch. Mittels einer Access Control List (ACL) wird nicht-authentisierten oder nicht-autorisierten Clients der Zugriff auf personenbezogene Informationen verwehrt. Die Hauptaufgaben des Context Service bestehen in der Verwaltung – einerseits der Entitäten in einem *Entity Container* und andererseits der so genannten *Transformer* in einem *Transformer Repository*. Bei letzteren handelt es sich um eine vordefinierte Menge von Komponenten, welche Entitäten in andere Entitäten überführen können. Welche Form der Transformation dabei zum Einsatz kommt, und ob diese durch Entwickler auch um Mechanismen zur Inferenz erweitert werden können, ist nicht näher spezifiziert.

In der untersten Schicht befinden sich Komponenten der Typen *Context Monitor* und *Context Actuator*. Erstere werden irreführend auch als Clients bezeichnet, was eine Assoziation mit der schon beschriebenen Context Client Layer nahe legt. Jedoch ist ein Context Monitor dafür zuständig, auf Sensoren zuzugreifen, und deren Informationen als Entitäten im Entity Container zu speichern, d. h. die Kontextbereitstellung vorzunehmen. Abhängig von der zugrunde liegenden Kontextquelle kann er die Informationen entweder synchron, d. h. per Polling, oder asynchron, d. h. per Benachrichtigung liefern. Das Pendant zum Context Monitor bildet der Context Actuator der den Kontext manipulieren kann indem er Veränderungen in der physischen Umgebung auslöst. Weil ein Actuator eine solche Veränderung erst

in Abhängigkeit von Kontextinformationen auslöst, kann er auch als spezielle Form des Clients angesehen werden. Er registriert sich beim Context Service für die durch ihn zu manipulierenden Entitäten, um bei deren Änderung benachrichtigt zu werden. Durch diese direkte Kopplung kann er auch zur Synchronisation der Kontextinformationen zwischen zwei Context Service Instanzen eingesetzt werden.

JCAF bietet Schnittstellen und Klassen für die Implementierung der Komponenten der oberen und unteren Schicht der Architektur – Clients, Monitore und Aktuatoren – zur Entwicklung kontextbezogener Anwendungen. Zusätzlich kann über Schnittstellen umfassend Einfluss auf die Betriebsweise der Komponenten der mittleren Schicht – den Context Service und die Transformer – genommen werden. Für die Repräsentation der Kontextinformationen stehen Basisklassen und -schnittstellen zur Verfügung, welche durch anwendungsspezifische Klassen zu erweitern und zu implementieren sind.

Einschätzung

JCAF trägt sowohl Eigenschaften eines Frameworks als auch einer Middleware. So stellen Verwaltung der Komponenten des Frameworks und der Kontextinformationen durchaus Funktionen einer Middleware dar, welche die zugrunde liegende Heterogenität oder technische Details kapseln. Jedoch müssen die Entwickler kontextbezogener Anwendungen über das API signifikanten Einfluss auf JCAF nehmen, um dessen Betrieb zu ermöglichen bzw. ihn an die Gegebenheiten des Einsatzszenarios anzupassen. Dazu gehört die Steuerung der Kommunikationstopologie zwischen den Context Service Instanzen, der Einsatz von Aktuatoren zur verteilten Synchronisation von Kontextinformationen, etc. Weil diese Aufgaben jedoch Aspekte betreffen, die orthogonal zur Anwendungsentwicklung sind, trägt JCAF den deutlichen Charakter eines Frameworks zur Entwicklung von Kontextdiensten und kontextbezogenen Anwendungen.

4.1.6. Nexus

Die Entwickler der Nexus-Plattform [HKL+99] gehen davon aus, dass eine Vielzahl der Informationen des menschlichen Alltags und der menschlichen Umgebung ortsbezogen ist, oder geographisch angeordnet ist. Dementsprechend soll sie die Entwicklung und den Betrieb ortsbezogener Software ermöglichen.

Überblick

Augmented Area Model Ähnlich wie die Systeme SOCAM, CoBrA und Gaia erzeugt Nexus ein Modell seiner Umgebung. Dabei wird ein mehr oder weniger klar umgrenzter räumlicher Bereich (z. B. Raum, Etage, Gebäude) als so genannte *Aug-*

mented Area[*] modelliert. Für jedes anwendungsrelevante *physische Objekt*, d. h. jede Entität, wird eine Datenstruktur im so genannten *Augmented Area Model* erzeugt. Neben einer Vielzahl möglicher Attribute enthält sie die räumliche Position des repräsentierten Objektes. Zusätzlich zu den physischen Objekten kann ein Augmented Area Model *virtuelle Objekte* enthalten. Sie stellen Platzhalter bzw. Schnittstellen dar und ermöglichen die Einbindung externer Dienste und Systeme, welche nicht Teil der Nexus-Plattform sind.

Mehrere Augmented Areas können zu einer *Augmented World* integriert werden. Aufgrund der einheitlichen Modellierung können einerseits Beziehungen zwischen Augmented Areas (z. B. Überdeckung, Entfernung) ermittelt, und anderseits deren Eigenschaften (z. B. Größe) mit einander verglichen werden. Jede Augmented Area kann über die einheitlichen APIs der Nexus-Plattform durch Anwendungen genutzt werden. Aufgrund der Integration in die Augmented World können Anwendungen über die APIs bedarfs- und ereignisabhängig auf wechselnde Augmented Areas zugreifen.

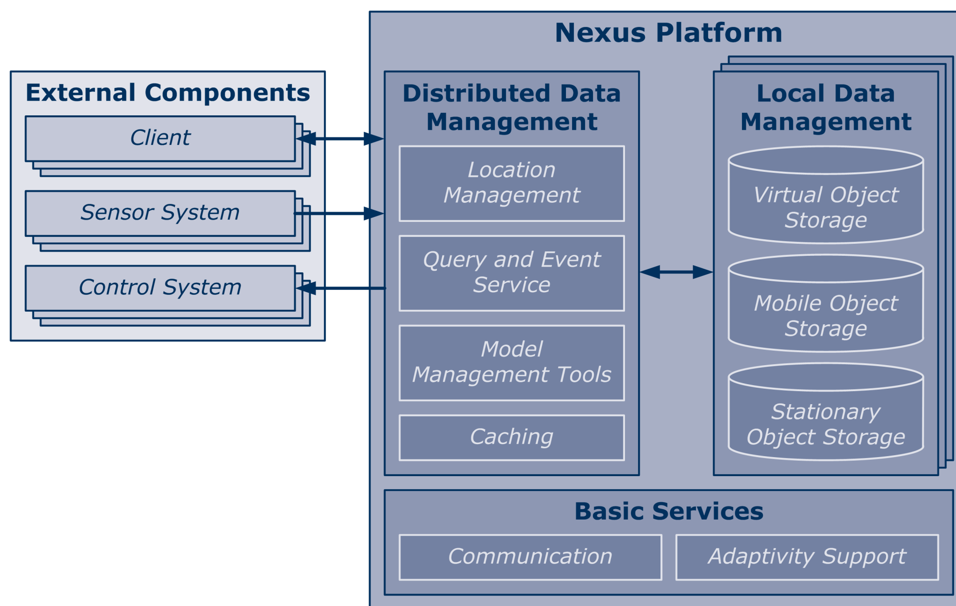

Abbildung 4.4.: Architektur der Nexus-Plattform (angelehnt an [HKL+99])

Architektur Der Aufbau der Nexus-Plattform besteht im wesentlichen aus drei Blöcken (Abb. 4.4). Sie verwalten das Augmented Area Model und bieten *externen*

[*] augmented (dt. erweitert, angereichert)

Komponenten die erforderlichen Dienste zum Zugriff darauf an.

Zu den externen Komponenten zählen neben *Clients* – Anwendungen welche auf das Augmented Area Model und die Funktionalität der Nexus-Plattform zugreifen – auch *Sensor* und *Control Systems*. Während erstere die Informationen zur Repräsentation der Augmented Area liefern, können letztere an diese Repräsentation gebunden werden, sodass Veränderungen in der Umgebung wiederum zur Steuerung von Vorgängen in Derselben führen können. Alle externen Komponenten werden über einheitliche APIs in die Nexus-Plattform integriert.

Die *Basic Services* bieten Grundfunktionen für den Betrieb der Plattform. Dazu gehört die Kopplung der beteiligten Knoten der Plattform einer Augmented Area mittels entfernter *Kommunikation*. Ein wesentliches Problem stellt dabei die Integration der heterogenen Menge von Kommunikationstechnologien und -strukturen dar. *Adaptivity Support* verwaltet die verfügbaren Ressourcen und ermöglicht die Benachrichtigung von Anwendungen, sodass sich diese an veränderte Gegebenheiten anpassen können.

Jede Augmented Area betreibt eine eigene Instanz des *Local Data Management*, welches zur Speicherung und Verwaltung des Augmented Area Model notwendig ist. Abhängig von ihrem Änderungsverhalten werden die verschiedenen Objekte – stationäre, virtuelle und mobile – auf unterschiedlichen Knoten der Nexus-Plattform gespeichert.

Die Komponenten des *Distributed Data Management* erbringen die verschiedenen Funktionen der Nexus-Plattform und integrieren die einzelnen Local Data Management Instanzen der Augmented Areas zu einer gemeinsamen Augmented World. Sie ermöglichen die Betrachtung und Nutzung der Augmented World als ein homogenes Konstrukt. Der Dienst des *Location Management* bildet die Objekte und Bereiche der Augmented Area auf diejenigen Knoten der Nexus-Plattform ab, welche die zugehörigen Informationen bzw. Teile des Augmented Area Models speichern. Er wird deshalb vom *Query and Event Service* genutzt, um die Aufrufe der Clients an die zugehörigen Knoten weiter zu leiten, sodass der Zugriff auf die Daten des Augmented Area Models oder die Funktionalität der Plattform gewährleistet ist. Für letzteres ist die Manipulation der zugehörigen virtuellen Objekte erforderlich, welche durch die *Model Management Tools* bereitgestellt wird. Für die Realisierung der eingeschränkten abgekoppelten Operation von Nexus-Knoten werden Funktionen für das *Caching* und Prefetching von Modelldaten vorgeschlagen.

Einschätzung

Nexus fokussiert sehr stark auf die Probleme der Modellierung der Augmented Areas und Augmented World. Dazu gehört die Integration verschiedenster Formen der Ortsmodellierung. Aktuellere Arbeiten [BN04] folgen – wie SOCAM und CoBrA – dem Trend zur Verwendung von Ontologien und untersuchen deren Einsatz für

die Ortsmodellierung und das Augmented Area bzw. World Model. In wieweit die Unterstützung von ortsunabhängigen Anwendungen durch Nexus möglich ist, kann nicht beurteilt werden.

4.2. Vergleich

Die vorgestellten Arbeiten (Kap. 4.1.1 bis 4.1.6) sind verschiedenen Ursprungs und verfolgen unterschiedliche Ziele. Als Basis des anschließenden Vergleichs werden die Anforderungen /A 1/ bis /A 17/ herangezogen. Die Anforderungen aus der theoretischen Betrachtung (Kap. 2) zielen auf die Unterstützung der Entwicklung und der Ausführung kontextbezogener Software ab. Das beinhaltet einerseits die einheitliche und umfassende *Modellierung* und Repräsentation von Kontextinformationen – ggf. inklusive zusätzlicher Merkmale (z. B. Qualität, Ort, Zeit). Andererseits aber auch die Entwicklung von Kontextquellen und kontextbezogener Software basierend auf entsprechenden *Komponentenmodellen* und *Frameworks*. Nicht zuletzt bestehen Forderungen nach einer *Middleware* mit umfangreicher Laufzeitunterstützung zur Behandlung der Dynamik von Geräten und Informationsquellen. Als Ergebnis der Analyse des Anwendungsszenarios (Kap. 3) muss das Gesamtsystem aus Middleware und Anwendungskomponenten, sowie Kontextquellen, qualitativen Anforderungen hinsichtlich *Skalierbarkeit* und Leichtgewichtigkeit gerecht werden.

4.2.1. Zusammenfassung

Die Gegenüberstellung (Tab. 4.1 auf den Seiten 82–83) zeigt die Verschiedenheit der untersuchten Arbeiten. Sie reichen von leichtgewichtigen Ansätzen für mobile Geräte (Hydrogen) bis hin zu umfangreichen verteilten Betriebssystemen (Gaia).

Die Unterstützung zur Modellierung von Kontextinformationen reicht von einfachen Java-Klassen bis hin zu Ontologien – z. B. bei SOCAM und CoBrA. Durch ihre ausdrucksstarke Semantik gestatten Letztere nicht nur die detaillierte Beschreibung der Eigenschaften und Zusammenhänge von Kontextinformationen, sondern auch die Validierung der Kontextmodelle, sowie logikbasierte Inferenz. Problematisch ist diesbezüglich einerseits die ressourcenintensive Verarbeitung und andererseits die Allgemeingültigkeit der Ontologien zu bewerten. Weil sie für die vorgestellten Systeme und deren konkrete Einsatzfälle entwickelt wurden, sind sie potentiell domänen- oder anwendungsspezifisch. Dadurch wird die Integration von Kontextquellen und -senken erschwert, welchen abweichende Domänenmodelle zugrunde liegen. Vor dem Hintergrund der heterogenen Definition des Begriffes Kontext ist selbst der Versuch der Definition einer abstrakten Super-Ontologie eine große Herausforderung.

Für die Entwicklung kontextbezogener Software stehen entweder einheitliche APIs oder Anfragesprachen zur Verfügung. Erstere unterstützen in den meisten Fällen sowohl asynchronen als auch synchronen Kontextzugriff, d. h. Push und Pull. Jedoch

ist ihre Ausdrucksstärke begrenzt, sodass der selektive Zugriff auf große Mengen von Kontextinformationen nicht möglich ist. Anfragesprachen hingegen sind durch ihre Ausdrucksstärke zum gezielten Zugriff auf Kontextinformationen besser geeignet. Es werden sowohl systemspezifische als auch generische Anfragesprachen wie RDQL in Kombination mit Ontologien eingesetzt. Die Einbindung von Kontextquellen und anderen Ressourcen wird vorwiegend durch Spezifikation von Frameworks unterstützt, die Adapterklassen mit einheitlichen Schnittstellen bieten. JCAF verfolgt das Konzept des Frameworks soweit, dass die Entwickler sogar Einfluss auf grundlegende Funktionen wie Kopplung und verteilte Datensynchronisation nehmen müssen. Diese Aspekte sind jedoch orthogonal zu den Aspekten der Anwendung und deshalb für die Auslagerung in die Middleware prädestiniert.

Während Middleware für ontologiebasierte Systeme primär die Verarbeitung der Ontologien (inkl. Inferenz) zum Gegenstand hat, besteht ihre Hauptaufgabe bei den übrigen Systemen in der transparenten Kopplung der Systeminstanzen der beteiligten Geräte. Weiterhin übernimmt sie die Kopplung der Kontextquellen und -senken auf lokaler Ebene, sowie die Verwaltung und Synchronisation der verteilten Kontextinformationen. Weil in Nexus der Position zentrale Bedeutung zukommt, sorgt die Middleware dort für die Lokalisation von Geräten und anderen Objekten. Der Aspekt der entfernten Kommunikation zur Kopplung von Geräten erfährt keine detaillierte Betrachtung. Es wird sowohl das C/S- als auch das P2P-Paradigma eingesetzt. Meist ohne auf damit verbundene Aspekte der Skalierbarkeit einzugehen.

Im Fall von SOCAM erlaubt die Beschreibung des Systems eine Beurteilung der Eigenschaften der Kommunikationsarchitektur, welche nur wenig skalierbar ist. Bei vielen Systemen werden Ereignisdienste nach dem Publish/Subscribe-Prinzip eingesetzt, um die Verwaltung der Kontextquellen oder sogar den Austausch der Kontextinformationen zu realisieren. Jedoch bieten die verwendeten themenbasierten Nachrichtenkanäle nur grob granulare Strukturierungsmöglichkeiten und stellen somit einen hohen Kommunikationsaufwand für das System dar. Aufgrund der Strukturierung nach Themen kann die detaillierte Filterung der Informationen nur lokal auf jedem Gerät vorgenommen werden. Bei vielen Systemen existieren aufgrund der Fokussierung auf andere Aspekte oft nur unzureichende Informationen über die Kommunikation zwischen einzelnen Geräten. Die Problematik des erhöhten Ressourcenbedarfs zur Verarbeitung von Ontologien ist bereits bekannt. Sie wird z. T. dadurch verursacht, dass viele Inferenzsysteme nicht in der Lage sind, mit volatilen Informationen umzugehen [Eic07]. In Folge dessen führt die Änderung einer Information zur Entfernung der veralteten Information aus der Wissensbasis der Reasoner, dem Einfügen neuer Information und einer erneuten Überprüfung der Wissensbasis. Weil dieser Vorgang mehrere Sekunden dauern kann, ist er nicht für Informationen geeignet, welche sich innerhalb von Sekundenbruchteilen ändern.

Tabelle 4.1.: Gegenüberstellung verwandter Arbeiten

System / Ansatz	**Modellierung**	**Komponenten / Framework**	**Middleware**	**Skalierbarkeit**
Semantic Space & SOCAM	definiert Hierarchie von Kontextontologien; Akzeptanz und Allgemeingültigkeit der ULCO unklar; RDQL für Anfragen	Kapselung von Kontextquellen durch einheitliche Context Provider	Context Interpreter macht Reasoning als Dienst der Middleware; SLS realisieren Verzeichnisdienst	Effektivität und Skalierbarkeit der Service Aggregation problematisch; ressourcenintensives Reasoning; Context Interpreter ist Engstelle
CoBrA	Definition einer Kontextontologie; RDQL für Anfragen	JADE-Framework für mobile Agenten zum Zugriff auf Informationsquellen	Kontextaggregation und -inferenz, sowie Konsistenz und Datenschutz durch Context Broker	P2P-ähnliches Broker Team; hoher Aufwand zur Synchronisation der Knowledge Bases
Gaia	*MPACC*-Entwurfsmuster für adaptive Inhaltspräsentation; spezielle Anfragesprache für Objektsuche	einheitliche Objekte für jegliche Ressourcen; Context Provider kapseln Kontextquellen; uniformes API für Objektzugriff	verteiltes Betriebssystem mit Objektverzeichnis und ereignisbasierter Kommunikation; Integration vorhandener Middleware	erhöhter Kommunikationsaufwand durch grob granulare Ereigniskanäle

Fortsetzung auf der nächsten Seite

Gegenüberstellung verwandter Arbeiten – Fortsetzung

System / Ansatz	**Modellierung**	**Komponenten / Framework**	**Middleware**	**Skalierbarkeit / Effizienz**
Nexus	Ortsmodellierung für Augmented Area bzw. World Model	APIs für Zugriff auf Augmented Area Model, sowie Integration von Sensoren und Aktoren	entfernte Kommunikation; Ressourcenüberwachung und Benachrichtigung; Integration der Augmented Areas; lokale und verteilte Datenverwaltung; Push-/Pull-Zugriff; Positionsverwaltung	(keine Einschätzung möglich)
Hydrogen	Java-Klassen zur Kontextrepräsentation	Quellen durch Adapter gekapselt (ggf. inkl. Inferenz)	verteilter Kontextdienst für ortstransparentes, dynamisches Binden von Kontextquellen; Push-/Pull-Zugriff; Kopplung per P2P	zeitintensives Polling aller Kontextquellen
JCAF	Java-Klassen zur Kontextrepräsentation	Klassen für Monitore und Aktuatoren; Kommunikationssteuerung durch Clients; explizite Kontextsynchronisation durch Aktuatoren	Push-/Pull-Zugriff per ACLs kontrolliert; Transformatoren für Inferenz; dynamische Einbindung der Monitore und Aktuatoren	(keine Einschätzung möglich)

4.2.2. Abgrenzung

Im Rahmen dieser Arbeit soll der Schwerpunkt der Betrachtungen auf der effizienten und skalierbaren Verwaltung dynamisch verfügbarer, heterogener Kontextquellen liegen, weil dieser Aspekt bisher unzureichend behandelt ist. Er soll die Hauptfunktion einer Middleware darstellen, welche den Betrieb kontextbezogener Software ermöglicht. Des Weiteren sind Möglichkeiten zur Entwicklungsunterstützung in Form von Frameworks zu untersuchen, um die Einbindung der heterogenen Kontextquellen weiter zu vereinfachen. Die Entwicklung ausdrucksstarker Modelle bzw. Metamodelle zur Repräsentation von Kontextinformationen soll hier nur insoweit betrachtet werden, wie dies für die Diskussion der zuvor genannten Punkte notwendig ist. Die Komplexität der Verarbeitung derartiger Modelle und die Problematik ihrer Akzeptanz durch die Entwickler wurden bereits erwähnt. Wegen der Heterogenität von Ableitungsschemata bzw. Inferenzmechanismen kann die Unterstützung der Middleware ohnehin nicht auf logische Inferenz beschränkt bleiben. Andere Formen der Ableitung sollten demzufolge gleichberechtigt als Quellen abstrakter Kontextinformationen durch eine Middleware integriert werden können. Nicht zuletzt eröffnen die Methoden des Model-Driven Software Development (MDSD) die Möglichkeit, bei der Entwicklung von Kontextquellen und -senken ausdrucksstarke Sprachen zur Repräsentation und Verarbeitung von Kontextinformationen einzusetzen, und diese durch Transformationen auf unterschiedliche Ausführungsumgebungen (inkl. ihrer Middleware) abzubilden, welche ggf. weniger leistungsfähig sind.

4.3. Aktuelle Entwicklungen

Während der Entstehung dieser Arbeit fanden Entwicklungen im Bereich ubiquitärer Middleware statt, welche keinen Eingang in die vorliegende Arbeit gefunden haben, aber nicht unberücksichtigt bleiben können. Deshalb werden im Folgenden weitere Ansätze vorgestellt, um neue relevante Lösungen zu identifizieren.

4.3.1. Meteor

Im Gegensatz zu den bereits vorgestellten Ansätzen verzichtet das Projekt *Meteor* [JSP06, JQSP07] auf eine Betrachtung der Kapselung von Informationsquellen. Es enthält aber interessante Konzepte zur dezentralen Aggregation von Informationen in Netzwerken, deren Anwendung auf die Verwaltung von Kontextquellen viel versprechend ist, und die in Kapitel 7 erneut aufgegriffen wird.

Überblick

Meteor basiert auf der Idee der Distributed Hash Table (DHT) bei welcher alle Knoten eines P2P-Netzwerkes einen strukturierten, verteilten Speicher bilden*. Weil jeder Peer des Netzwerkes Informationen sowohl in diesem Speicher ablegen, als auch daraus auslesen kann, eignet sich eine DHT auch zum Informationsaustausch. Meteor macht sich diese Möglichkeit zunutze, und erweitert die Funktionalität durch Überlagerung der DHT mit weiteren Protokollebenen.

Die Basis des Protokollstapels von Meteor bildet die DHT. Sie etabliert eine logische Struktur auf dem Netzwerk aller Peers, welche es ermöglicht, eine Nachricht in durchschnittlich $log\,n$ Sprüngen† zu einem von n Peers zu vermitteln. Aufgrund dieser geringen Komplexität eignet sich der Einsatz einer DHT für Netzwerke mit vielen Peers, und stellt somit einen wichtigen Ansatz zur Lösung von Anforderung /A 17/ dar.

Um Daten in einer DHT abzulegen, muss ihnen ein Schlüssel aus der logischen Struktur der DHT zugeordnet werden. Während die klassische Vergabe der Schlüssel durch eine uniforme Hash-Funktion erfolgt, setzt Meteor auf der zweiten Protokollebene – über der DHT – eine so genannte *Hilbert Space Filling Curve (SFC)* ein. Eine solche Funktion hat die Fähigkeit, d-dimensionale Schlüsselräume zu einem 1-dimensionalen Schlüsselraum zu linearisieren. Diese Transformation ist notwendig, weil die gespeicherten Daten in Meteor nicht nur anhand eines einzigen Schlüssels, sondern anhand einer ganzen Menge von bis zu d Schlüsselmerkmalen identifiziert werden. Diese umfangreiche Beschreibung gestattet eine ausdrucksstarke Beschreibung der gespeicherten Daten und – in Kombination mit der SFC und der DHT – eine hohe Effizienz.

Bereits mit diesen Mitteln ist der gezielte Zugriff auf einzelne Daten möglich indem die Merkmale der gesuchten Daten als Schlüssel beim Zugriff spezifiziert werden. Können oder sollen die Schlüsselmerkmale jedoch nicht auf exakte Werte eingeschränkt werden, d. h. müssen Wertebereiche, Teilmerkmale oder Platzhalter eingesetzt werden, so ist eine weitere Funktion von Meteor notwendig. Die Abbildung der Merkmalsmenge auf eine Menge von Bereichen der SFC – anstatt auf einen einzigen Punkt auf der SFC. Diese Abbildung wird genutzt, um einen Multicastbaum in Form eines Präfixbaumes aufzubauen. Eine solche Kommunikationstopologie ermöglicht die effiziente Verteilung der Zugriffsnachrichten an die beteiligten Peers, welche den Bereichen der SFC entsprechen. Der Präfixbaum wird sogar in umgekehrter Richtung zur Aggregation der Daten der einzelnen Peers der adressierten Bereiche eingesetzt.

Basierend auf den Ebenen der DHT und der SFC kann die oberste Ebene die eigentliche Anwendungslogik zur Datenaggregation realisieren. In Meteor stellt je-

*Details zur Funktionsweise einer DHT sind in Kap. 5.6.4 beschrieben.

†Ein Sprung (engl. hop) ist die Weiterleitung einer Nachricht von einem Peer zu einem Anderen.

der Peer einen so genannten *Associative Rendezvous (AR)* dar. Dabei handelt es sich um einen Punkt an dem die gespeicherten Daten mit den Zugriffsanfragen zusammen treffen und anhand assoziativer Vorgänge miteinander verglichen werden. Sowohl Daten als auch Anfragen werden mittels Nachrichten zwischen den ARs ausgetauscht. Der wichtigste Teil der Nachricht ist das Profil, welches die Schlüsselmerkmale enthält. Darüber hinaus ist in der Nachricht kodiert, ob es sich um eine Anfrage- oder eine Datennachricht handelt, und welche Reaktion der Sender vom empfangenden AR erwartet. Ein AR kennt vier Reaktionen auf eine Nachricht – *store*, *retrieve*, *notify* und *delete*. Sie erlauben das Speichern, Abfragen und Löschen von Daten und Anfragen, sowie die Benachrichtigung infolge der Speicherung von Daten. Mit diesen Dienstprimitiven kann sowohl synchroner als auch asynchroner Zugriff auf die gespeicherten Daten vorgenommen werden. Sogar die proaktive Speicherung von Anfragen ist möglich, welche erst im Moment der Speicherung der gesuchten Daten bedient werden.

Einschätzung

Die vorgestellte Architektur bietet einerseits eine hohe Skalierbarkeit, und anderseits die generische Beschreibung von auszutauschenden Daten. Aufgrund der starken Asymmetrie zwischen Kontextquellen und Kontextsenken, sowie der potentiell großen Dynamik von Kontextinformationen sind jedoch trotzdem Skalierbarkeitsprobleme zu erwarten, wenn Meteor zum Austausch von Kontextinformationen eingesetzt wird.

Hinsichtlich der Schlüsselmerkmale muss festgehalten werden, dass Meteor streng hierarchische Typsysteme voraussetzt. Weil dies jedoch bei der Modellierung von Kontextinformationen mit Ontologien – oder dem Metamodell dieser Arbeit – nicht notwendigerweise sicher gestellt ist, sind mögliche Probleme und Lösungen zu untersuchen.

Durch die Verwendung der SFC anstatt einer uniformen Hash-Funktion [KLL+97] wird die Lastverteilung zwischen den ARs beeinträchtigt. Dadurch kann es zur Konzentration des Datenaustauschs auf eine Untermenge der ARs kommen. Darüber hinaus stellt der Betrieb der DHT und des restlichen Systems eine signifikante Last für mobile und integrierte Geräte dar, welche diese Geräte für ihre Anwender unbenutzbar machen können. Aufgrund der inhärent engen Kopplung der Knoten einer DHT verursachen häufige Beitritte und Austritte bzw. Ausfälle von mobilen Geräten einen hohen Kommunikationsaufwand, um die Konsistenz innerhalb der DHT wieder herzustellen. Mobile Geräte sind für den Betrieb einer DHT folglich weniger gut geeignet.

Abschließend sei bemerkt, dass die Implementierung von Meteor auf dem P2P-Framework JXTA* basiert und frei verfügbar ist [Mat08]. Jedoch gelang dem Au-

*Juxtapose (JXTA) [TAA+03]

tor keine erfolgreiche praktische Untersuchung des Systems, weil der veröffentlichte Quelltext signifikante Lücken aufweisst. Laut vorliegender Quellen wurde das System sowohl experimentell in einer lokalen Testumgebung mit 64 Rechnern und in der PlanetLab-Umgebung [Pla08] auf derzeit ca. 1.000 Rechnern weltweit getestet, als auch simulativ hinsichtlich seiner Skalierbarkeit untersucht.

4.3.2. Weitere Arbeiten

Die folgenden Arbeiten sollen nur kurz vorgestellt werden, weil sie im Vergleich zu Meteor wesentlich einfacher strukturiert sind, jedoch z. T. interessante Eigenschaften aufweisen oder bekannte Vertreter aktueller Ubiquitous Computing Middleware sind.

Contory

Aufgabe der Middleware des Systems *Contory* [RdF06] ist die Kontextbereitstellung für Mobiltelefone. Contory vereint unterschiedliche Ansätze der Kontextbereitstellung – von direkter Integration der Quellen in die Anwendung bis hin zu bedarfsabhängigem Austausch zwischen mehreren Geräten.

Neu ist bei Contory – wie auch in der vorliegenden Arbeit – die Verbesserung der Kontextquellenverwaltung und die Behandlung ihrer dynamischen Verfügbarkeit. Dies dient primär der Aufrechterhaltung der Verfügbarkeit von Kontextinformationen.

DOLCLAN

Im Gegensatz zu Contory und der vorliegenden Arbeit beschäftigt sich das System *DOLCLAN* [BM07] nicht mit der Verwaltung von Kontextquellen, sondern dem verteilten Zugriff auf Informationen. Dieser Zugriff wird durch so genannte Peer-to-Peer Distributed Shared Objects (PDSO) realisiert – Objekte welche mittels Kopien auf die Knoten des Systems verteilt werden. Um PDSO zu nutzen, ist die Einflussnahme des Entwicklers notwendig, welcher u. a. die Synchronisation der Objektkopien vornehmen muss. Ähnlich wie JCAF [Bar05] bietet DOLCLAN primär Unterstützung in Form eines Frameworks. Die Entwickler kontextbezogener Systeme müssen jedoch bei der Nutzung des Frameworks Einfluss auf Funktionen nehmen, welche nichts mit der zu entwickelnden Anwendung, sondern dem Betrieb des Frameworks bzw. der Middleware zu tun haben. Ein solches Vorgehen ist nur in überschaubaren Szenarien praktikabel. Ebenso ist der Einsatz der Objektkopien der PDSO, sowie deren ständige Synchronisation aufgrund zu erwartender Skalierbarkeitsprobleme nur in Anwendungen mit wenigen Objekten und kleinen Netzwerken ratsam.

COSMOS

Ein weiterer Vertreter der Frameworks ist *Context Entities Composition and Sharing (COSMOS)* [CRS07]. Obwohl es eine dreischichtige Architektur entwirft, welche die Kontexterfassung, Kontextverarbeitung und Kontextnutzung/Adaption separiert, sind die Grundbausteine von COSMOS – so genannte *Context Nodes* – auf allen drei Ebenen einsetzbar. Neu ist jedoch die Entkopplung dieser Context Nodes durch nachrichtenbasierte Kommunikation deren Realisierung auf das Framework *Dynamic Reflective Asynchronous Middleware (DREAM)* [LQS05] zurückgreift. COSMOS gestattet damit die Erfüllung der Anforderungen /A 8/ und /A 14/. Jeder Context Node kapselt die von ihm genutzten Informationsquellen, d. h. ggf. auch andere Context Nodes deren Informationen er für Inferenz benutzt. Die so entstehende Hierarchie von Context Nodes muss aufgrund des Frameworkcharakters von der Anwendung bzw. ihren Entwicklern zusammengestellt und verwaltet werden. Es gelten also auch bei COSMOS die Einschränkungen der übrigen Frameworks hinsichtlich Unterstützung beim Betrieb der kontextbezogenen Anwendungen. Weil die Leistungsfähigkeit von DREAM nur eingeschränkt untersucht wurde, ist eine Einschätzung der Skalierbarkeit der Kommunikation nicht möglich.

5. Verwaltung heterogener Kontextquellen

Bevor in diesem Kapitel die wesentlichen Konzepte zur Verwaltung von heterogenen Kontextquellen entwickelt werden, sollen – nach einem einleitenden Überblick – grundlegende Voraussetzungen für deren Verständnis geschaffen werden. Es wird einerseits das Konzept der Context Provider vorgestellt, welches die Integration der Kontextquellen in das Gesamtsystem ermöglicht, und andererseits das Metamodell des Context Service das u. a. der Beschreibung der Provider dient.

5.1. Überblick

Im Anwendungsszenario wurde bereits ersichtlich, dass Kontextquellen auf jeglichen Geräten anzutreffen sind. Dabei entstehen Konstellationen (Abb. 5.1 auf der nächsten Seite) in denen auf einem Gerät (A) Kontextquellen und Kontextsenken existieren, oder ein Gerät (B) ausschließlich über Kontextquellen verfügt. Es kann sogar vorkommen, dass ein Gerät (C) lediglich Kontextsenken betreibt. Aufgrund der Heterogenität von Kontextsenken und Kontextquellen (Anf. /A 6/) ist es angebracht, diese mittels einheitlicher Schnittstellen zu koppeln, um ihre Austauschbarkeit zu erhöhen. Durch diese Schnittstellen kommt es zu einer Kapselung der spezifischen Eigenschaften der Kontextquellen und Kontextsenken. Die so entstandenen Kom-

ponenten werden nachfolgend als *Context Provider*[*] – sie kapseln Kontextquellen – und als *Context Consumer* – sie kapseln Kontextsenken – bezeichnet. Zur Kopplung der Provider und Consumer wird weiterhin eine Middleware etabliert, welche zusätzliche Funktionen übernimmt – der *Context Service*.

Weil die einheitliche Kapselung von Kontextquellen und der Betrieb des Context Service an die Unterstützung bestimmter Technologien, sowie an ein Minimum von Ressourcen gebunden ist, ist es durchaus wahrscheinlich, dass ein Gerät (D) über Kontextquellen verfügt, aber weder die zugehörigen Provider noch den Context Service betreiben kann. Auf diese Kontextquellen müssen die zugehörigen Provider (hier: Provider S) mittels entfernter Kommunikation zugreifen. Dies erschließt einerseits die Welt drahtlos (z. B. Bluetooth, Radio Frequency Identification (RFID)) angebundener Peripherie – vorwiegend Sensoren (z. B. bei Provider Q) – und andererseits die Quellen aller erreichbaren Rechner, d. h. Web Services, Datenbanken, etc.

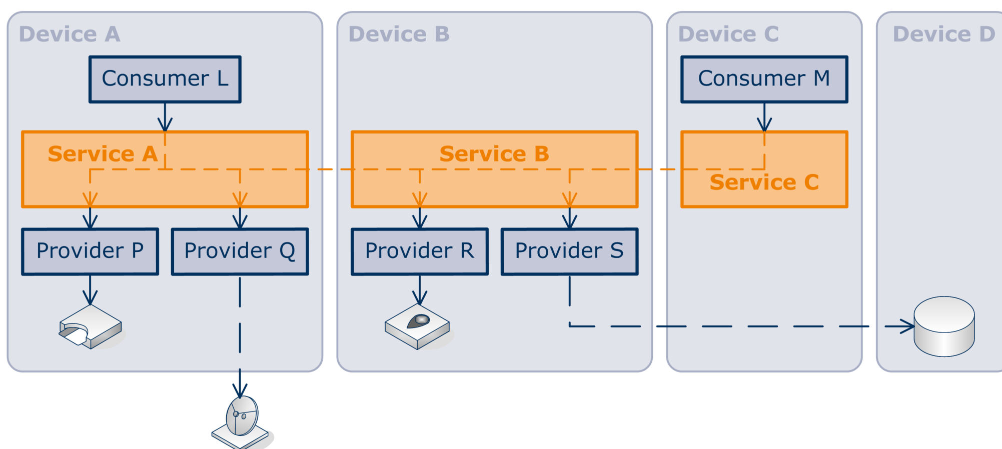

Abbildung 5.1.: Kopplung von Consumern und Providern über den verteilten Context Service, sowie Kapselung und Anbindung von Kontextquellen durch Provider

Damit der Zugriff der Kontextsenken nicht auf diejenigen Kontextquellen ihres lokalen Gerätes beschränkt bleibt, sollen die Instanzen des Context Service miteinander gekoppelt werden, um so die Menge verfügbarer Kontextquellen zu erhöhen. Eine wesentliche Funktion des Context Service liegt somit in der Kopplung der Consumer und Provider. Greift ein Consumer auf Kontextinformationen zu, so soll es

[*] Aufgrund vollständiger Englischsprachiger Benennung und Dokumentation bei der Implementierung, werden im Folgenden englische statt deutscher Begriffe verwendet – z.B. *Context Service*, *Provider* und *Consumer* anstatt von *Kontextdienst*, *-anbieter* und *-konsument*. Sofern die Bezeichnung eindeutig ist, wird auf den Vorsatz *Context* verzichtet.

für ihn unerheblich sein, ob sich die Kontextquellen, welche die benötigten Kontextinformationen liefern, auf dem lokalen Gerät des Consumers oder einem beliebigen entfernten Gerät befinden. Um den Kontextbedarf der Consumer zu befriedigen, nutzt der Kontextdienst jederzeit die größtmögliche Menge geeigneter Kontextquellen. Fällt eine Kontextquelle aus, so wird – sofern verfügbar – automatisch eine alternative Kontextquelle genutzt. Wird eine neue Kontextquelle verfügbar, so wird sie ebenfalls genutzt, sofern sie komplementäre Informationen zur Verfügung stellt. Andernfalls stellt sie eine Alternative dar, und bleibt (vorerst) ungenutzt, erhöht jedoch die Verfügbarkeit der Kontextinformationen. Diese Strategie verbessert die Robustheit des Systems gegenüber der schwankenden Verfügbarkeit einzelner Kontextquellen. Zusätzlich ermöglichen die Kapselung der Kontextquellen und der Zugriff auf sie über den Context Service die parallele Bedienung mehrerer Consumer, selbst wenn die eigentliche Kontextquelle nur exklusiv (für einen einzigen Provider) nutzbar ist.

Weil jederzeit eine Situation entstehen kann, in welcher der Kontextbedarf eines Consumers durch keine der Kontextquellen der lokalen und aller erreichbaren Kontextdienstinstanzen bedient werden kann, ist die schwankende Verfügbarkeit von Kontextquellen nicht vollständig zu verbergen. Sie kann auch durch lokale Ereignisse ausgelöst werden (z. B. Energiesparmaßnahmen). Für den Kontextbedarf der Consumer ist es unerheblich, ob die Bedienung aufgrund fehlender lokaler oder entfernter Kontextquellen nicht erfolgen kann. Man könnte aufgrund dieser Maskierung des Verteilungsaspektes durch die dynamische Quellenverfügbarkeit sogar von Verteilungstransparenz sprechen. Jedoch wird nachfolgend gezeigt, dass das System nicht vermag, die signifikant höheren Latenzen bei der Kommunikation mit den entfernten Providern vollständig zu verbergen.

5.2. Context Provider

Neben dem Context Service besitzen die Context Provider zentrale Bedeutung für die Realisierung ubiquitärer Systeme. Ihre wichtigste Aufgabe besteht in der Abbildung des generischen Informationszugriffs, welcher über die einheitliche Schnittstelle eines jeden Providers bereitgestellt werden soll, auf einen quellenspezifischen Informationszugriff. Details zur Form der Schnittstelle und ihrer Implementierung sind im Rahmen der Realisierung und Evaluierung in Kapitel 6.3 beschrieben. Ein Framework zur Reduzierung des Implementierungsaufwandes wird im Verlauf dieses Kapitels vorgestellt (Kap. 5.2.1). Im Zuge der Abbildung müssen die von der Kontextquelle gelieferten Kontextinformationen auf Basis eines einheitlichen Metamodells repräsentiert (Kap. 5.3) werden.

Der Context Service verwaltet alle vorhandenen Provider und stellt ihre Kontextinformationen den Consumern zur Verfügung. Um deren Informationsbedarf bedienen

zu können, muss der Context Service feststellen können, welche Informationen ein Provider liefern kann. Diese Metainformationen zur Selbstbeschreibung stellt jeder Provider über seine Schnittstelle zur Verfügung (siehe Kap. 5.4.1).

Jedoch müssen die Provider im Rahmen der Verwaltung in zwei Gruppen eingeteilt werden. Die Exemplare der ersten Gruppe – die *Low-Level Provider* – kapseln Informationsquellen, welche vorwiegend gemessenen Kontext bereitstellen. Ihre wichtigste Eigenschaft besteht darin, dass diese Provider sofort einsatzbereit sind, sofern die von ihnen gekapselten Quellen vorhanden sind. Weil die Kapselung auch die Suche der jeweiligen Quelle einschließt, wird die Abhängigkeit des Providers von seiner Quelle durch den Provider vor dem Kontextdienst verborgen. Aus Sicht des Kontextdienstes besitzen Low-Level Provider keine Abhängigkeiten, welche über deren Verwendbarkeit entscheiden und demzufolge zu berücksichtigen wären.

Dem gegenüber kapseln die Vertreter der zweiten Gruppe – die *High-Level Provider* – Informationsquellen, welche ihre Kontextinformationen aus anderen Kontextinformationen gewinnen. Bei der Informationsgewinnung kann einerseits eine Ableitung auf Basis eines Inferenzschemas (z. B. probabilistisch) vorgenommen werden, oder andererseits eine Aggregation mehrerer Informationen durchgeführt werden. Deshalb sind High-Level Provider erst dann einsetzbar, wenn die von ihnen benötigten Informationen durch andere Provider bereitgestellt werden. Die Verwaltung beider Providerklassen wird in den Kapiteln 5.4 bis 5.6 geschildert.

Aufgrund der dynamischen Verfügbarkeit der Quellen ist beim Kontextzugriff eine entsprechende Behandlung erforderlich, um den Kontextzugriff von Consumern trotz schwankender Verfügbarkeit aufrecht zu erhalten. Dies ist einerseits ein nichtfunktionaler Aspekt und andererseits eine Aufgabe, welche unabhängig vom Typ der Quellen ist. Folglich ist sie prädestiniert für die Auslagerung in den Context Service (Abb. 5.1 auf Seite 90), sodass eine einheitliche und effiziente Lösung möglich ist. Eine Aufgabe des Context Service besteht folglich in der Bedienung der Anfragen von Consumern durch Selektion geeigneter Provider und entsprechende Weiterleitung, sowie Übertragung der Antworten – des Kontextes. Die Providerselektion erfolgt erst bei Bedarf, um der schwankenden Verfügbarkeit von Providern Rechnung zu tragen.

Mittels entfernter Kommunikation werden mehrere Instanzen des Context Service, welche auf unterschiedlichen Geräten ausgeführt werden, zu einem verteilten Dienst integriert. Diese Kopplung wird vollständig durch den Context Service realisiert. Die Provider werden transparent eingebunden, sodass (a) ein Provider nicht feststellen kann, ob er von einer entfernten Instanz genutzt wird und (b) ein Consumer nicht feststellen kann woher die Kontextinformationen stammen. Jede Instanz des Context Service verwaltet ihre lokal vorhandenen Context Provider – unabhängig davon, wo sich die von ihnen gekapselten Kontextquellen befinden. Entfernte Provider, d. h. Provider die von Context Service Instanzen entfernter Geräte verwaltet werden, können transparent genutzt werden, wenn zur Befriedigung des Kontextbedarfs lokaler

Consumer keine oder zu wenige passende, lokale Provider zur Verfügung stehen.

5.2.1. Provider Framework

Zur Kapselung einer Kontextquelle durch einen Provider gehört, neben der Umwandlung der Informationen von ihrer quellenspezifischen Repräsentation in eine generische Repräsentation, vor allem die Realisierung des eigentlichen Infomrationszugriffes. Um sowohl synchrone Anfragen als auch asynchrone Abonnements von Consumern bedienen zu können, soll jeder Provider beide Formen der Zugriffssemantik unterstützen (Anf. /A 12/ und /A 13/). Bei näherer Betrachtung der Informationsquellen hinsichtlich ihrer nativ unterstützten Zugriffssemantik zeigt sich, dass nur die wenigsten Quellen beide Formen bieten. Weil die jeweils fehlende Funktionalität nachgebildet werden muss, ist eine nähere Betrachtung der Quellen sinnvoll, um zu erwartende technische Herausforderungen und den entstehenden Aufwand zu untersuchen.

Erweitertes Quellenmodell

Bisher wurden lediglich zwei Typen von Kontextquellen unterschieden, solche die nieder- und andere die höherwertige Kontextinformationen liefern (Kap. 2.3.1). Diesem Unterschied im Quellenmodell wurde durch die Entwicklung von Low- bzw. High-Level Context Providern Rechnung getragen. Die angesprochene Erweiterung des Quellenmodells führt einen weiteren Aspekt ein – die native Zugriffssemantik der Quelle.

Vorwiegend bei niederwertigen Kontextquellen sind folgende Quellenklassen anzutreffen:

- Die *Pull Source* liefert ihre Daten nur auf Anfrage.
- Die *Periodic Push Source* liefert ihre Daten selbstständig und in regelmäßigen Abständen, unabhängig davon, ob sie sich geändert haben.
- Die *Change Push Source* liefert ihre Daten selbstständig, aber nur dann, wenn sie sich geändert haben.

Die Existenz dieser drei Klassen würde sich ohne eine einheitliche Schnittstelle für Low-Level Provider aufgrund der Informationsabhängigkeit auch auf die High-Level Provider übertragen, sodass dort ebenfalls entsprechende Klassen zu finden wären.

Zusätzlich führt die Art der Ableitungsschemata auch bei den High-Level Providern zu unterschiedlichen nativ unterstützten Zugriffsformen. Handelt es sich um ein aufwändiges, zeitintensives Ableitungsschema – z. B. logikbasiertes Reasoning – so ist eine Durchführung der Ableitung nur in größeren Zeitabständen möglich. Aber

dieser Aufwand lohnt sich auch nur dann, wenn sich die zugrunde liegenden Kontextinformationen geändert haben. Folglich kann ein logikbasierter Reasoner eine Change Push Source darstellen. Dem gegenüber sind einfachere Ableitungschemata, wie simple mathematische Konvertierungen, zeitnah sowohl im Moment der Anfrage, als auch bei jeder änderungsbedingten Benachrichtigung einsetzbar. Sie stellen eine Kombination aus Pull Source und Change Push Source dar.

Zugriffsemulation

Kapselt ein Provider eine Kontextquelle die zu einer der beschriebenen Klassen gehört, so muss seine Implementierung die fehlende Zugriffssemantik emulieren. Diese Nachbildung ist unabhängig vom Zugriff auf die Quellen und darüber hinaus unabhängig von der Umwandlung der Quelleninformationen in eine domänenspezifische, metamodellkonforme Repräsentation. Deshalb werden die Aufgaben der *Zugriffsemulation* und der *Informationsextration* separaten Bestandteilen der Providerimplementierung zugewiesen. Letztere übernimmt ein *Adapter*. Ein quellenneutraler Emulator bildet dagegen die fehlende Zugriffssemantik nach – siehe Abbildungen 5.2 bis 5.4 auf den Seiten 95–96.

Die nachfolgenden Abschnitte beschreiben die verschiedenen vorgefertigten Bausteine, welche zur Zugriffsemulation bei der Implementierung von Providern eingesetzt werden können. Durch diese Unterstützung gelingt es, den Implementierungsaufwand auf ein Mindestmaß zu reduzieren. Lediglich die quellenspezifische Informationsextraktion im Adapter muss „manuell", d. h. durch einen Entwickler, implementiert werden.

Push-Emulation Wie ihr Name schon impliziert, unterstützt eine Pull Source inhärent den Pull-Zugriff. Zur Bedienung synchroner Anfragen muss der kapselnde Provider die Informationen lediglich von der Quelle über ihre Schnittstelle abfragen ($R1$ und $R2$, Abb. 5.2 auf der nächsten Seite), sie in eine metamodellkonforme Repräsentation umwandeln ($R3$) und anschließend an den Context Service ausliefern (R4). Werden die Kontextinformationen jedoch abonniert, so muss ein separater Prozess – der *Pull Worker* – gestartet werden ($S1$), welcher regelmäßig die Kontextinformationen abfragt (S_i und S_{i+1}) und konvertiert (S_{i+2} und S_{i+3}). Anschließend übergibt er die Informationen an einen *Comparator* (S_{i+4}), welcher deren letzte Werte aus einem *Cache* abfragt (S_{i+5} und S_{i+6}). Haben sich die Kontextinformationen geändert (S_{i+7}), wird sowohl der Context Service benachrichtigt, als auch der Cache mit den neuen Werten aktualisiert. Zur Beendigung des Abonnements muss lediglich der Pull Worker beendet werden.

Pull-Emulation Pull-Zugriff ist zu emulieren, wenn eine Change Push Source gekapselt wird. Weil diese Quelle bereits asynchron arbeitet, ist bei asynchronen Abon-

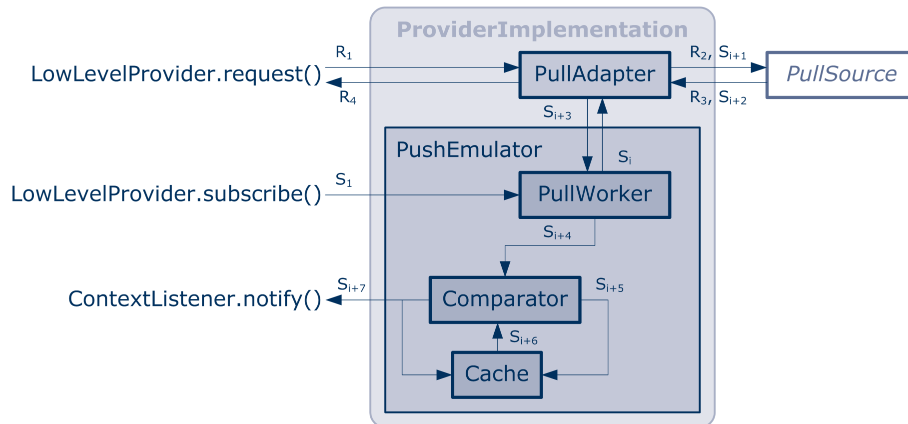

Abbildung 5.2.: Emulation des Push-Zugriffs bei Kapselung einer Pull-basierten Kontextquelle

nements die Konvertierung der Informationen erforderlich. Zusätzlich müssen die Benachrichtigungen auf die Zeiten beschränkt werden, während der ein Abonnement besteht. Diesem Zweck dient der so genannte *Push Filter*. Bei der tatsächlichen Durchführung eines Abonnements ($S1$) wird er derart konfiguriert, dass alle nachfolgenden Änderungsmeldungen (S_i, Abb. 5.3) der Quellen konvertiert (S_{i+1}) und ausgeliefert werden (S_{i+2}). Bei der Beendigung des Abonnements wird im Push Filter die Weiterleitung der Benachrichtigungen deaktiviert. Für die Emulation des Pull-Zugriffs müssen die Kontextinformationen der letzten Benachrichtigungen in einem *Cache* zwischengespeichert werden (S_{i+1}). Nimmt der Context Service einen synchronen Zugriff vor ($R1$), so werden die aktuellsten Kontextinformationen aus dem Cache an ihn ausgeliefert ($R2$).

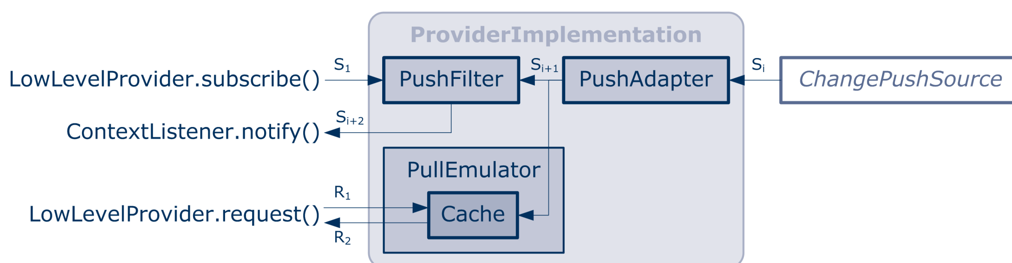

Abbildung 5.3.: Emulation des Pull-Zugriffs bei Kapselung einer Push-basierten Kontextquelle, welche nur bei Änderungen benachrichtigt

Kombinierte Pull-/Push-Emulation Für eine Periodic Push Source sind beide Formen der Emulation erforderlich (Abb. 5.4), weil sie (1) keinen synchronen Zugriff unterstützt und (2) Benachrichtigungen unabhängig davon vornimmt, ob sich die beobachteten Werte geändert haben. Wie bei der reinen Pull-Emulation wird jeder Pull-Zugriff ($R1$) aus dem Cache bedient ($R2$). Für die Behandlung von Abonnements ($S1$) ist analog zur Push-Emulation die Kombination eines *Comparators* und eines Caches erforderlich. Jedoch ist aufgrund des Push-Charakters der Quelle kein abfragender Prozess nötig. Die regelmäßigen Benachrichtigungen (S_i) werden konvertiert (S_{i+1}) und anschließend mit zwischengespeicherten Informationen (S_{i+1}, S_{i+3}) verglichen. Bei Feststellung einer Änderung wird diese an den Push Filter weitergeleitet (S_{i+4}), welcher sie bei einem bestehenden Abonnement ausliefert (S_{i+5}). Die Einrichtung eines Abonnements ($S1$) erfolgt analog zur Pull-Emulation.

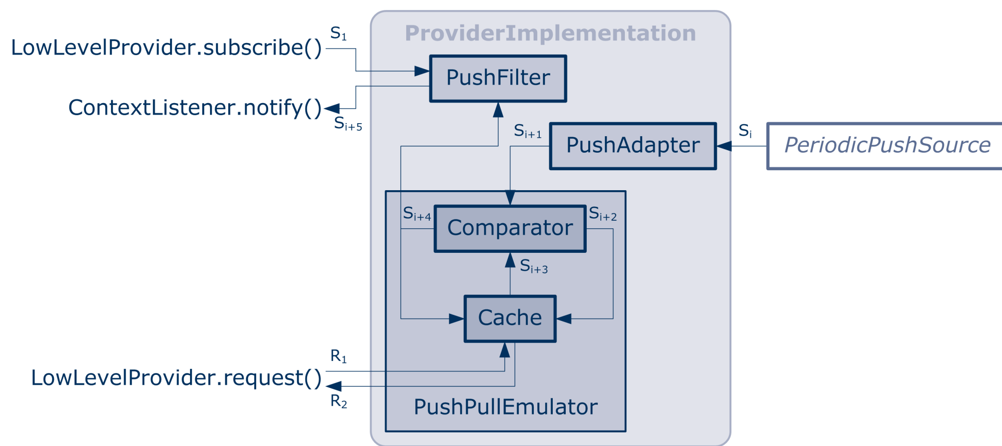

Abbildung 5.4.: Emulation des Pull- und Push-Zugriffs bei Kapselung einer Push-basierten Kontextquelle die regelmäßig benachrichtigt

Anmerkungen

Zur Einsparung von Systemressourcen wäre es wünschenswert, bei der Emulation des Pull-Zugriffs die Konvertierung ebenfalls über den Push Filter zu deaktivieren, sofern kein Abonnement besteht. Das ist jedoch nicht möglich, weil dann der Cache nicht mehr aktualisiert wird, und in Folge dessen bei einem synchronen Zugriff veraltete Informationen ausgeliefert würden.

Die Entscheidung zur Unterstützung des erweiterten Quellenmodells durch ein Framework gestattet die Reduzierung des Implementierungsaufwandes ohne gleichzeitige Einschränkung der Implementierung von Providern. Somit kann bei der Integration spezieller Quellen auf die Nutzung des Frameworks verzichtet oder aber

dessen Umfang erweitert werden. Die Unterstützung der verschiedenen Quellentypen durch die Middleware selbst würde dem gegenüber zu keiner weiteren Reduzierung des Implementierungsaufwandes führen, die Bandbreite unterstützter Quellentypen jedoch auf die bereits bekannten limitieren. Die Integration zukünftiger Quellentypen würde dadurch nennenswert erschwert.

5.3. Kontextmetamodell

Nicht nur die tatsächlich erfassten oder ausgetauschten Kontextinformationen, sondern auch die durch Provider lieferbaren Kontextinformationen, sowie die von Consumern benötigten Kontextinformationen müssen zur Laufzeit repräsentiert werden, sodass der Kontextdienst die Kopplung von Consumern und Providern vornehmen kann. Deshalb wird an dieser Stelle das zugrunde liegende Metamodell vorgestellt.

Weil man Kontext als implizites Wissen betrachten kann, bieten sich zu dessen Modellierung die Ansätze zur Wissensrepräsentation an. Dazu zählen neben Ontologien (siehe Kap. 4) auch das Topic-Map-Metamodell und das Entity-Relationship-Metamodell [Che76]. Die beiden letzten Ansätze wurden zur Entwicklung des Metamodells des Kontextdienstes herangezogen [SKSY06]. Topic Maps sind eng mit Ontologien verwandt und eignen sich zur Modellierung logischer Strukturen und Zusammenhänge. Im Gegensatz dazu werden Entity-Relationship-Modelle in vielen klassischen Softwareentwurfsprozessen zur Datenmodellierung eingesetzt. Das gewählte Metamodell ist somit für die Modellierung von Kontextinformationen und Anwendungsdaten geeignet und ebnet den Weg für eine einheitliche Behandlung beider Informationsklassen.

Für die Repräsentation von Kontextinformationen sind im Metamodell drei Typen so genannter *Kontextelemente* definiert (Abb. 5.5 auf der nächsten Seite). Ein *Entity* repräsentiert ein eigenständiges Objekt. Dabei kann es sich sowohl um ein physisches als auch ein logisches Objekt handeln. Jede Eigenschaft einer solchen Entität wird durch ein *Attribute* modelliert, welches eine einzelne Kontextinformation aufnimmt. Alternativ kann ein Attribut wiederum aus mehreren Attributen aufgebaut sein, welche jeweils die eigentlichen Informationen beherbergen. Der Tiefe dieser Schachtelung/Strukturierung sind keine technischen Grenzen auferlegt. Auf diese Weise lassen sich mit geschachtelten Attributen komplexe Datenstrukturen modellieren. Weil eine Entität selten isoliert existiert, kann durch Definition einer *Association* eine unidirektionale Relation zwischen zwei Entitäten hergestellt werden.

Jedes Kontextelement wird durch einen eindeutigen *Key* k identifiziert. Ein Schlüssel besteht aus drei Teilen (Gl. 5.1).

$$k = (m, t, i) \,|\, m \in \mathbb{M} = \{EN, AT, AS\} \,\wedge\, t \in \mathbb{T} \,\wedge\, i \in \mathbb{I} \tag{5.1}$$

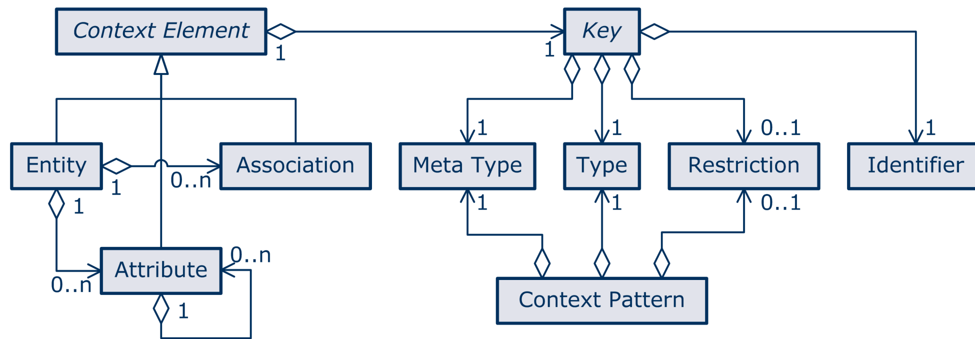

Abbildung 5.5.: Typen des Kontextmetamodells

Der *Meta Type* m des Kontextelements zeigt an, dass es sich um eine Entität (EN), ein Attribut (AT) oder eine Assoziation (AS) handelt. t ist der anwendungs- bzw. domänenspezifische *Type*. Und der *Identifier* i ist ein innerhalb von t eindeutiger Bezeichner. Während die Menge der Metatypen $\mathbb{M}$ aufgrund des Metamodells fest vorgegeben ist, können $\mathbb{T}$ und $\mathbb{I}$ frei durch die Entwickler gewählt werden. Durch Spezifikation eines $t \in \mathbb{T}$ kann eine Klasse von gleichartigen Kontextelementen definiert werden, welche dieselbe Semantik besitzen. Die Schlüssel der einzelnen Kontextelemente einer solchen Klasse unterscheiden sich ausschließlich in der Komponente i.

In Gleichung 5.2 sind Beispiele für solche Schlüssel dargestellt. k_m identifiziert eine Entität vom Typ „Person“ mit dem Bezeichner „7204938294“ – einen Nutzer. Das Attribut, welches den Vornamen dieses Nutzers enthält, könnte durch k_n identifiziert werden. k_o ist der Schlüssel des Attributes „Strasse“, das Teil des strukturierten Attributes „Anschrift“ eines Nutzers ist. Um die Eindeutigkeit von t und i innerhalb des Attribut-Metatyps zu gewährleisten, kann eine simple Vorschrift für ihre Definition angewendet werden. Sie erzeugt t bzw. i des Attributschlüssels durch Konkatenation von t bzw. i des übergeordneten Kontextelements mit einem frei gewählten Attributnamen. Die einzelnen Fragmente werden durch ein Separatorzeichen getrennt. Durch diese hierarchische Definition können Attributnamen bedeutungsgleich in mehreren Entitätstypen wiederverwendet werden. Der Schlüssel k_p identifiziert eine Assoziation des Typs „kennt“.

$$
\begin{aligned}
&k_m = (EN, Person, 7204938294)\\
&k_n = (AT, Person.Vorname, 7204938294.Vorname)\\
&k_o = (AT, Person.Anschrift.Strasse, 7204938294.Anschrift.Strasse)\\
&k_p = (AS, kennt, 48jalf24j0f934j0f3)
\end{aligned}
\tag{5.2}
$$

Die Definition der domänenspezifischen Typen von Entitäten, Attributen und Assoziationen muss im Rahmen der Softwareentwicklung von Providern und Consumern

bei der Kontextmodellierung vorgenommen werden. Nur so kann gewährleistet werden, dass die von Providern gelieferten Kontextinformationen die von Consumern erwartete Semantik besitzen. Um den technischen Aufwand und den Ressourcenbedarf des Context Service zu beschränken, führt dieser keine Validierung der gelieferten oder angefragten Kontextinformationen durch. Durch diese generische Form der Beschreibung und Verwaltung können jederzeit neue Provider entwickelt und auf Geräten installiert werden, ohne dass der Context Service umfangreich konfiguriert werden muss. Das gestattet ebenfalls den parallelen Betrieb von Providern und Consumern unterschiedlicher Anwendungsdomänen auf ein und derselben Context Service Instanz.

Um Mengen der Kontextinformationen zu beschreiben, wie sie von Consumern benötigt werden, sind die zuvor beschriebenen Schlüssel nur bedingt einsetzbar. Sie beschreiben ein einziges Kontextelement. Diese Eindeutigkeit könnte vermieden werden, wenn statt des Tripels (m, t, i) nur das Tupel (m, t) verwendet werden würde. Jedoch wäre diese Lösung unflexibel, weil ein solches Tupel alle Kontextelemente einer konkreten Klasse* beschreibt. Einerseits würde das dazu führen, dass die Consumer von Kontextelementklassen mit vielen Instanzen dazu gezwungen sind, die gelieferten Informationen selbst einer weiteren Analyse und Filterung zu unterziehen. Und andererseits könnte es dazu führen, dass Provider, welche unterschiedliche Instanzen derselben Kontextelementklasse bereitstellen, als fälschlicherweise gleichwertig betrachtet werden.

Zur Einschränkung der Menge der gesuchten Instanzen einer Klasse wird das Tupel (m, t) um die Komponente r erweitert. Sie enthält einen optionalen, eingrenzenden Ausdruck – die Restriktion – (z. B. „id > 10 & id < 25"). Das entstandene Tripel (Gl. 5.3) wird *Context Pattern cp* genannt. Wie solche Context Pattern zur Verwaltung der Context Provider eingesetzt werden, wird in den Kapiteln 5.4 bis 5.6 beschrieben. In diesem Zusammenhang wird auch der Inhalt von r und dessen Behandlung durch den Kontextdienst vorgestellt.

$$cp = (m, t, r) \,|\, m \in \mathbb{M} \wedge t \in \mathbb{T} \wedge r \in \mathbb{R} \tag{5.3}$$

5.4. Lokale Providerverwaltung

Es wurde schon darauf hingewiesen, dass ein nennenswerter Teil der dynamischen Verfügbarkeit von Context Providern bzw. ihren Quellen durch gerätelokale Vorgänge (siehe Kap. 2.4.1) verursacht ist. Deshalb muss die Menge der auf einem Gerät installierten Context Provider jederzeit überwacht werden (Kap. 6.1), sodass ein neu gestarteter Provider automatisch in die Verwaltung einbezogen werden kann

*Als *Klasse* wird nachfolgend die Kombination aus Meta Type und Type verwendet.

(Kap. 5.4.4). Dies gilt analog für die Entfernung des Providers aus der Verwaltung infolge seiner Beendigung (Kap. 5.4.5).

Weil High-Level Provider nur dann Kontext liefern können, wenn sie in der Lage sind, den von ihnen benötigten Kontext zu konsumieren, sind sie von Low-Level Providern oder weiteren High-Level Providern abhängig. Dabei handelt es sich um eine lose Kopplung, weil für die Bedienung des Informationsbedarfs keine konkrete Instanz eines Providers, sondern lediglich eine konkrete Information erforderlich ist. Um die Verwendbarkeit eines High-Level Providers feststellen zu können, muss demzufolge ermittelt werden, ob die erforderlichen Kontextinformationen verfügbar sind, d. h. durch andere Provider geliefert werden können. Dies kann jedoch nur geschehen, wenn eine geeignete Beschreibung des Providers existiert.

5.4.1. Providerbeschreibung

Die beiden Facetten eines High-Level Providers – Bereitstellung und Konsum von Kontext – können durch Context Patterns beschrieben werden. Dazu wird je eine Menge so genannter *Provided* und *Consumed Context Patterns* definiert, welche mit den Symbolen $\mathbb{P}$ und $\mathbb{C}$ dargestellt werden. Nachfolgend werden $\mathbb{C}_i$ und $\mathbb{P}_i$ genutzt, um die Mengen eines einzelnen Providers PRO_i zu referenzieren. C und P hingegen enthalten die entsprechenden Patterns aller n installierten Provider (Gl. 5.4).

$$\begin{aligned} \mathbb{P} &= \bigcup_{i=1}^{n} \mathbb{P}_i \\ \mathbb{C} &= \bigcup_{i=1}^{n} \mathbb{C}_i \end{aligned} \tag{5.4}$$

Nachfolgend sind die Mengen $\mathbb{P}_i$ und ggf. auch $\mathbb{C}_i$ von drei Providern dargestellt. Dabei liefern die zwei Low-Level Provider PRO_1 und PRO_2 die Attribute $A.w$ und $A.x$, d. h. zwei Kontextvariablen.

$$\begin{aligned} \mathbb{P}_1 &= \{(AT, Device.display, \emptyset)\} \\ \mathbb{P}_2 &= \{(AT, Device.audio, \emptyset)\} \end{aligned}$$

Der High-Level Provider PRO_3 aggregiert lediglich die von PRO_1 und PRO_2 gelieferten Attribute zu einem Entitätstyp A.

$$\begin{aligned} \mathbb{P}_3 &= \{(EN, Device, \emptyset)\} \\ \mathbb{C}_3 &= \{(AT, Device.audio, \emptyset), (AT, Device.display, \emptyset)\} \end{aligned}$$

Ein High-Level Provider PRO_i kann folglich durch Tupel charakterisiert werden (Gl. 5.5). Die Verwendung dieser Form für Low-Level Provider mit $\mathbb{C}_i = \emptyset$ ermöglicht später die einheitliche Betrachtung beider Providertypen.

$$PRO_i = (\mathbb{P}_i, \mathbb{C}_i) | \mathbb{P}_i \subseteq \mathbb{CP} \wedge \mathbb{C}_i \subseteq \mathbb{CP} \quad (5.5)$$

Die semantische Abhängigkeit der beiden Mengen kann durch die Funktion *dep* (Gl. 5.6) beschrieben werden, welche eine strikte Ordnung auf der Menge aller Patterns $\mathbb{CP}$ darstellt. Die Basis des somit entstehenden Abhängigkeitsgraphen von Patterns wird immer durch die Provided Patterns von Low-Level Providern gebildet. Sind diese nicht bzw. nur teilweise verfügbar, so ist auch nur ein entsprechender Teil des Graphen verfügbar. Gleiches gilt für die Provided Patterns von High-Level Providern, welche Teil der Consumed Patterns weiterer High-Level Provider sind. Die Feststellung der Verfügbarkeit ist Gegenstand der Abschnitte 5.4.4 bis 5.4.5.

$$dep : \mathbb{P}_i \mapsto \mathbb{C}_i \quad (5.6)$$

5.4.2. Filterung von Providerbeschreibungen

In Kapitel 5.3 werden so genannte Context Pattern vorgestellt, die eine flexible Beschreibung von Kontextinformationen gestatten. Das diese Context Pattern auch zur Beschreibung von Providern und Consumern eingesetzt werden können, wurde im Verlauf dieses Kapitels bereits verdeutlicht. Bisher wurde jedoch nicht auf den Inhalt der Komponente r des Context Pattern Tripels eingegangen.

Es gibt eine Vielzahl möglicher Merkmale zur näheren Beschreibung von Kontextinformationen, welche darüber hinaus in unterschiedlichen Formen spezifiziert werden können. So kann beispielsweise der Ursprungsort einer Kontextinformation durch eine physische Position in Form von GPS-Koordinaten, oder mittels der logischen Bezeichnung eines Raumes in einem Gebäude angegeben werden. Jedes Merkmal und jede Beschreibungsform hat in bestimmten Szenarien oder Anwendungsdomänen ihre Berechtigung. Deshalb ist es unpraktikabel von Entwicklern die Unterstützung einer festen Menge von Merkmalen in einer vorgeschriebenen Form zu fordern mit welchen sie die Provider und Consumer beschreiben müssen. Statt dessen ist eine Lösung notwendig, welche mehrere domänenspezifische Beschreibungen erlaubt, um die gleichzeitige Nutzung von Providern und Consumern unterschiedlicher Anwendungsdomänen mit demselben Context Service zu ermöglichen.

Restriktionen

Um die Repräsentation der verschiedenen Merkmale einer Kontextklasse zu ermöglichen, wird r dahingehend erweitert, dass es eine unbegrenzt große Menge von getypten Merkmalen $\mathbb{F}$ enthalten kann (Abb. 5.6 auf der nächsten Seite). Jedes dieser Merkmale $F \in \mathbb{F}$ ist ein Tupel und besteht aus einem Typ FT (`FeatureType`) und einem Ausdruck FX (`FeatureExpression`). Der Semantik des Typs ist domänenspezifisch – ebenso wie das Format des Ausdruckes – und muss folglich von

den Entwicklern während der Entwicklung von Providern und Consumern definiert werden.

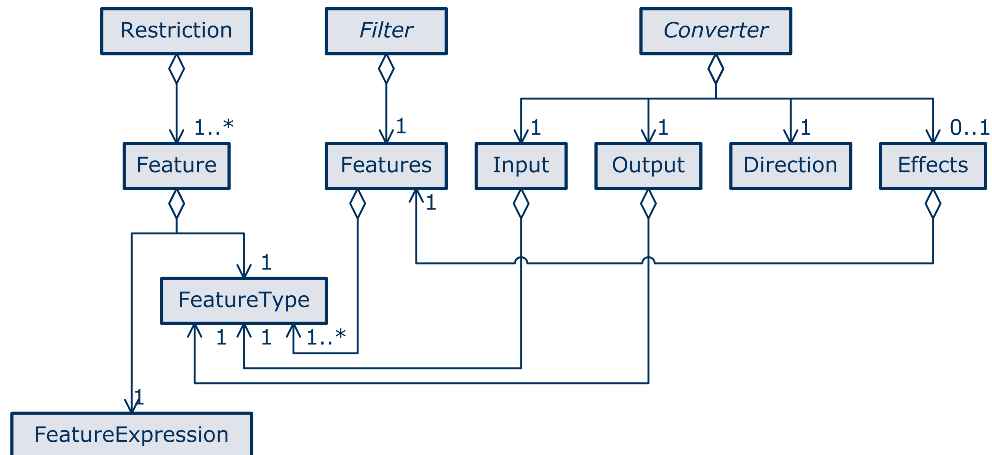

Abbildung 5.6.: Erweiterung der Restriktion eines Context Pattern um getypte Merkmale

Sollen die von einem Consumer benötigten Kontextinformationen anhand eines Context Pattern spezifiziert werden, so muss die Implementierung des Consumers dafür sorgen, dass die Restriktion des Pattern die Beschreibungen der eingrenzenden Merkmale enthält. Analog des Kontextbedarfs eines Consumers ist die Beschreibung des Kontextangebots eines Providers möglich (Kap. 5.4.1). Darüber hinaus ist mit Hilfe der Merkmale die Beschreibung jeder einzelnen Kontextinformation möglich, weil die Eigenschaften der Kontextinformationen eines einzelnen Providers stark schwanken können. Mögliche Ursachen dafür sind ein inhomogenes Verhalten des Providers bei der Kontextbereitstellung oder der Betrieb des Providers in unterschiedlichen Modi mit abweichenden qualitativen Eigenschaften. Somit stellt die Beschreibung des Providers nur einen groben Rahmen dar. Während für die Beschreibung der Provider und Consumer Context Pattern zum Einsatz kommen, ist für eine einzelne Kontextinformation bzw. ein Kontextelement dessen Schlüssel k um die Komponente r zu einem Quadrupel zu erweitern.

Filter

Ohne die technische Unterstützung der zuvor skizzierten Merkmale zur Laufzeit ist deren Spezifikation wenig sinnvoll. Andernfalls würden die zusätzlichen Merkmale zusammen mit ihren Kontextinformationen von einem Provider erzeugt und durch den Context Service lediglich an einen Consumer weitergeleitet. Dieser hätte dann

die Aufgabe, die Merkmale geeignet auszuwerten, und so die Kontextinformationen selbst zu filtern. Wenn Consumer ihren Kontextbedarf anhand von Merkmalen eingrenzen, dann ist bei der Selektion geeigneter Provider ebenfalls eine technische Unterstützung notwendig, weil sonst kein Vergleich der Merkmale der Context Pattern von Providern und Consumern möglich ist. Beide Parteien sind bei der Selektion passiv, weil sie allein durch den Context Service durchgeführt wird.

Damit die Interpretation domänenspezifisch definierter Merkmale weitgehend vom Context Service isoliert erfolgen kann, bietet dieser die Unterstützung so genannter *Filter* an. Dabei handelt es sich um Komponenten die in der Lage sind, ein oder mehrere domänenspezifische Merkmale bzw. Merkmalstypen $\mathbb{FT}$ zu interpretieren. Um einen Filter zur Analyse von Merkmalen nutzen zu können, müssen die Merkmale getypt sein, und der Filter muss spezifizieren, welche Merkmalstypen er unterstützt. Letzteres erfolgt durch Angabe einer Liste von Merkmalstypen.

Bei der Providerselektion werden Filter wie folgt verwendet:

1. Bestimmung der Merkmalstypen $\mathbb{FT}_C$ anhand derer der Consumer die Kontextinformationen eingrenzen möchte.
2. Feststellung für welche dieser Merkmalstypen Filter registriert sind $\mathbb{FT}_F$.
3. Analyse jedes Providers der die gesuchte Kontextelementklasse liefern kann:
   a) Isolation des Provided Pattern das die gesuchte Klasse beschreibt.
   b) Abgleich der Merkmalstypen des Pattern mit $\mathbb{FT}_F$.
   c) Jeder Filter wird mit den zu vergleichenden Context Pattern des Consumers und des Providers aufgerufen, und signalisiert, ob die Merkmalsausdrücke übereinstimmen*. Signalisiert ein Filter keine Übereinstimmung, wird die Filterung für den aktuellen Provider abgebrochen, der Provider als ungeeignet gewertet und die Filterung mit dem nächsten Provider fortgesetzt.

Alle Provider deren Context Pattern zu Übereinstimmungen mit dem Pattern des Consumers führen, werden zur die Bedienung der Anfrage gebunden. Abbildung 5.7 auf Seite 105 zeigt ein Beispiel der Providerselektion das über die Verwendung von Filtern hinausgeht. Die dort zusätzlich eingesetzten Konverter werden direkt im Anschluss an die Filter behandelt.

Aus Gründen der Effizienz und Eindeutigkeit setzt dieser Ansatz voraus, dass es für jeden Merkmalstyp nur einen einzigen Filter gibt. Mehrere Filter, welche u. U. widersprüchliche Filterergebnisse für denselben Merkmalstyp liefern sind nicht akzeptabel. Weiterhin wird in Kauf genommen, dass trotz Filterung unerwünschte Informationen an Consumer ausgeliefert werden. Das kann vorkommen, wenn:

*Je nach Semantik des Merkmalstyps kann dabei ein Wert-, Muster-, Mengenvergleich, etc. durch den Filter angewendet werden.

- Das Context Pattern des Consumers keinerlei Merkmalstypen spezifiziert oder solche, die nicht von Filtern unterstützt werden.
- Das Context Pattern des Providers keinerlei Merkmalstypen spezifiziert.

Die Alternative der Unterdrückung derartig unzureichend beschriebener Kontextinformationen ist im Sinne der Verfügbarkeit von Kontextinformationen nicht zu favorisieren.

Konverter

Weil es möglich ist, dass zwei Merkmalstypen semantisch identisch, jedoch syntaktisch unterschiedlich sind, gestattet der Context Service die Verwendung so genannter *Konverter*. Sie können Ausdrücke eines Merkmalstyps in Ausdrücke eines anderen umwandeln, sodass anschließend eine Filterung möglich ist.

Konverter sind ebenfalls selbstbeschreibende Komponenten. Im Gegensatz zu Filtern ist ihre Beschreibung jedoch umfangreicher. Sie besteht aus einem Eingabe- und einem Ausgabemerkmalstyp (`Input`, `Output`). Weiterhin definiert sie die Konvertierungsrichtung (`Direction` – uni- oder bidirektional), da eine Konvertierung nicht zwingend eindeutig und umkehrbar sein muss. Weil von der Konvertierung eines Merkmalstyps auch andere Merkmalstypen betroffen sein können, ist für diese eine optionalen Liste der betroffenen Merkmalstypen `Effects` vorgesehen.

Providerselektion

Abbildung 5.7 auf der nächsten Seite zeigt den Datenfluss bei der Providerselektion. Der `TemperatureConsumer` benötigt Temperaturinformationen aus seiner Umgebung deren Ursprung er anhand einer Bereichsangabe eingrenzt. Das zugehörige Merkmal seines Context Pattern hat den Typ `position.gps` und repräsentiert die GPS-Koordinaten des geforderten Bereiches. Der `ContextService` ermittelt anhand von `MetaType` und `Type` prinzipiell geeignete Provider – `TemperatureProvider A` und `B`.

Bevor die Filterung der Providerbeschreibungen statt finden kann, ermittelt der `ContextService` die eingrenzenden Merkmalstypen des Pattern des Consumers $\mathbb{FT}_C$ und vergleicht sie mit den unterstützten Merkmalstypen der registrierten Filter $\mathbb{FT}_F$. Die resultierende Menge von Merkmalstypen – hier `position.gps` – vergleicht er mit den Pattern eines jeden Providers. Für `TemperatureProvider A` ist dieser Vergleich erfolgreich. Die Merkmalstypen von `TemperatureProvider B` hingegen werden von keinem Filter unterstützt. Anschließend wird nach Übereinstimmungen zwischen den `Output`-Merkmalstypen der registrierten Konverter und den Merkmalstypen der Filter gesucht. Mit den `Input`-Merkmalstypen $\mathbb{FT}_{FC}$ der übereinstimmenden Konverter, wird `TemperatureProvider B` erneut analysiert und als geeignet eingestuft.

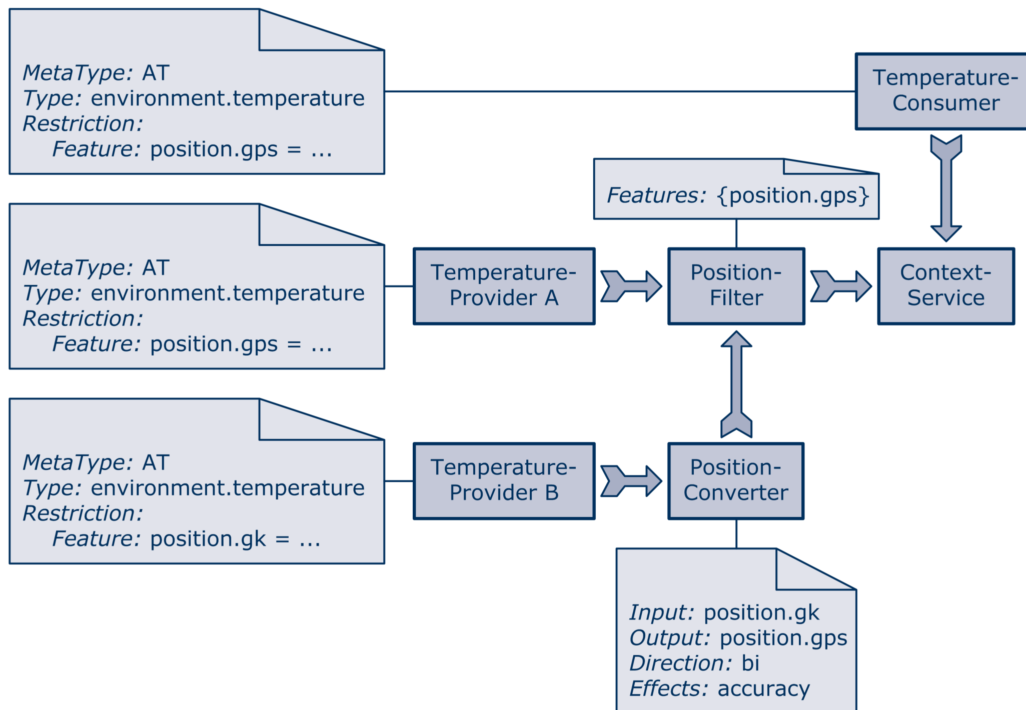

Abbildung 5.7.: Datenfluss bei der Selektion zweier Provider unter Verwendung eines Filters und eines Konverters

5.4.3. Lebenszyklus eines Providers

Neben der Erfüllbarkeit der informationsspezifischen Abhängigkeiten eines Providers gibt es weitere Aspekte, welche sich in seinem Lebenszyklus niederschlagen. Dazu gehört einerseits die dynamische Installation und De-Installation des Providers, welche dazu führt, dass er gestartet bzw. gestoppt wird. Andererseits werden Provider aus Gründen der Ressourceneinsparung nur dann aktiviert, wenn sie benötigt werden. In den übrigen Zeiten bleiben sie de-aktiviert. Die daraus resultierenden Zustände sind in Abbildung 5.8 auf der nächsten Seite dargestellt.

Ist ein Provider *gestoppt*, so ist er für das System nicht auffindbar (vgl. nachfolgend Provider *gestartet*)*. Dieser Zustand gilt initial für jeden Provider, und kann jederzeit aus jedem anderen Zustand durch den *Stopp* des Providers erreicht werden. Ein solcher *Stopp* muss durch die Analyse der potentiellen Auswirkungen auf die übrigen Provider behandelt werden (siehe Kap. 5.4.5). Im Gegensatz dazu, ist der

*Die Erkennung von Start und Stopp von Providern wird im Rahmen der Realisierung (Kap. 6.1) beschrieben.

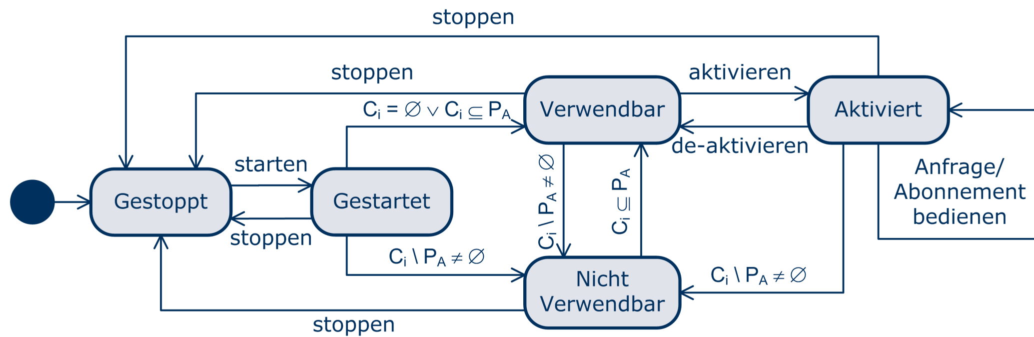

Abbildung 5.8.: Lebenszyklus eines Context Providers

Übergang in den einzigen Folgezustand nur durch den *Start* des Providers möglich. Wurde ein gestarteter Provider für den Context Service sichtbar, kann anschließend seine Verwendbarkeit festgestellt werden (siehe Kap. 5.4.4). Je nach Ergebnis der Analyse geht der Provider in den Zustand *verwendbar* oder *nicht-verwendbar* über.

Ein *verwendbarer* Provider könnte, aufgrund erfüllter Abhängigkeiten, theoretisch für die Bedienung von Anfragen und Abonnements genutzt werden. Aus Gründen der Ressourcenersparnis bleibt er jedoch solange deaktiviert bis er benötigt wird. Zustandsübergänge anderer verwendbarer Provider können sich ggf. derart auf den verwendbaren Provider auswirken, dass dessen Abhängigkeiten nicht länger erfüllt sind – er wird demzufolge *nicht-verwendbar*.

Nicht-verwendbare Provider verbleiben solange in diesem Zustand, wie sich eventuelle Zustandsänderungen anderer Provider nicht positiv auf die Erfüllbarkeit der Abhängigkeiten des Providers auswirken.

Um einen verwendbaren Provider tatsächlich zu nutzen, muss dieser mittels *Aktivierung* in den *aktivierten* Zustand versetzt werden. Nur in diesem ist der synchrone und asynchrone Zugriff auf die Kontextinformationen des Providers möglich. Ist der Zugriff nicht länger erforderlich, so kehrt der Provider nach seiner *De-Aktivierung* in den Zustand *verwendbar* zurück. Analog zu diesem Zustand können sich auch im *aktivierten* Zustand die Zustandsänderungen anderer Provider auf diesen Provider auswirken.

5.4.4. Provider gestartet

Weil die Charakteristik eines Providers während dessen Lebenszyklus konstant ist, kann eine proaktive* Analyse der Abhängigkeiten zwischen den Providern vorgenommen werden, um die Verfügbarkeit der Patterns zu ermitteln. Primäres Ziel ist die

*Um unnötige Latenz bei der Beantwortung von Anfragen zu vermeiden.

Unterteilung von $\mathbb{P}$ in die disjunkten Mengen der verfügbaren ($\mathbb{P}_A$) und nicht verfügbaren Provided Patterns ($\mathbb{P}_U$). So soll bei Anfragen effizient festgestellt werden, (a) ob diese überhaupt erfolgreich bedienbar sind, und (b) welche der gestarteten Provider für deren Bedienung geeignet sind. Darüber hinaus wird $\mathbb{C}$ ebenfalls unterteilt, weil die Verfügbarkeit der $\mathbb{C}_i$ eines Providers PRO_i Auswirkung auf die Verfügbarkeit seiner $\mathbb{P}_i$ hat. Für die Selektion geeigneter Provider zur Laufzeit und für die Ermittlung der Auswirkungen von Zustandsänderungen der Provider müssen die Patterns aller Mengen mit ihren liefernden bzw. konsumierenden Providern assoziiert sein. Dies wird durch die Datenstruktur der Patternmengen realisiert. Tabelle 5.1 fasst die Eigenschaften der verschiedenen Mengen zusammen.

Tabelle 5.1.: Verwaltete Mengen von Context Patterns

Menge	Inhalt	Verwendung
$\mathbb{P}_A$	Verfügbare Provided Patterns, mit ihren Assoziationen zu den jeweiligen verwendbaren Providern durch welche sie verfügbar gemacht werden.	Bei der Bedienung von Anfragen und Abonnements der Consumer zur Ermittlung, ob (a) die benötigten Kontextinformationen verfügbar sind, und (b) welche Provider diese liefern können.
$\mathbb{P}_U$	Nicht verfügbare Provided Patterns, sowie deren nicht verwendbare Provider.	Diese Menge existiert nur theoretisch, im praktischen Einsatz wird sie nicht benötigt
$\mathbb{C}_A$	Verfügbare Consumed Patterns und deren Provider, welche aufgrund der Verfügbarkeit ihrer $\mathbb{C}_i$ verwendbar sind.	Zum Vergleich mit $\mathbb{P}_A$ benutzt, um nach dem Stopp eines Providers abhängige Provider zu ermitteln, welche voraussichtlich nicht mehr verwendbar sind, und deshalb erneut untersucht werden müssen.
$\mathbb{C}_U$	Nicht verfügbare Consumed Patterns und ihre Provider, welche aufgrund der Nicht-Verfügbarkeit ihrer $\mathbb{C}_i$ nicht verwendbar sind.	Zusammen mit $\mathbb{P}_A$ benutzt, um nach einem Providerstart abhängige Provider zu ermitteln, welche potentiell verwendbar werden könnten, und deshalb untersucht werden müssen.

Es bleibt zu bemerken, dass $\mathbb{P}$ vollständig in die disjunkten Mengen $\mathbb{P}_A$ und $\mathbb{P}_U$ unterteilt wird. Gleiches gilt für $\mathbb{C}$ mit $\mathbb{C}_A$ und $\mathbb{C}_U$. Nachfolgend werden im Sinne der Vereinfachung zwei Mengen definiert: die verwendbaren und die nicht-verwendbaren Provider – $\mathbb{PRO}_A$ und $\mathbb{PRO}_U$ (siehe Gleichungen 5.7 und 5.8).

$$\mathbb{PRO}_A = \{PRO_i | \mathbb{C}_i = \emptyset \vee \mathbb{C}_i \subseteq \mathbb{P}_A\} \tag{5.7}$$

$$\begin{aligned} \mathbb{PRO}_U &= \mathbb{PRO}'_A \\ &= \mathbb{PRO} \setminus \mathbb{PRO}_A \\ &= \{PRO_i | \mathbb{C}_i \neq \emptyset \wedge \mathbb{C}_i \not\subseteq P_A\} \end{aligned} \tag{5.8}$$

Die Analyse eines Providers kann beginnen, sobald das System seinen Start festgestellt hat. Sie besteht aus zwei Phasen (siehe Abb. 5.9 auf Seite 110). In Phase I (1 - 6) wird der gestartete Provider PRO_i analysiert, während Phase II (7 & 8) eventuelle Auswirkungen des neuen Providers auf die bereits verwalteten analysiert:

1. Zuerst wird zwischen Low- und High-Level Providern unterschieden. Für erstere kann die aufwendige Überprüfung der Abhängigkeiten übersprungen werden – weiter mit 5.

2. Bei High-Level Providern wird ermittelt, ob die Consumed Patterns $\mathbb{C}_i$ durch die Provided Patterns existierender, verwendbarer Provider verfügbar sind. Ist das nicht der Fall, so geht es weiter mit 4.

3. Das Entstehen zirkulärer Abhängigkeiten innerhalb der Menge verwendbarer Provider $\mathbb{PRO}_A$ durch die Integration von PRO_i muss ausgeschlossen werden. Wäre das der Fall, könnte PRO_i nicht verwendbar gemacht werden, und wäre nachfolgend für eine erneute Analyse zu registrierten.

4. Ist ein Provider PRO_i aufgrund unerfüllter Consumed Patterns oder eines Abhängigkeitszyklus' nicht verwendbar, so muss er entsprechend markiert werden ($\mathbb{PRO}_U \cup PRO_i$ und $\mathbb{PRO}_A \setminus PRO_i$). Gleichzeitig wird eine Menge $\mathbb{C}_U$ geschaffen, welche Consumed Patterns enthält, deren Provider nicht verwendbar sind. Sie wird verwendet, um bei späteren Änderungen von $\mathbb{P}_A$ ggf. erneut zu untersuchende Provider bestimmen zu können. Deshalb werden ihr die $\mathbb{C}_i$ hinzugefügt, welche mit dem nicht verwendbaren Provider assoziiert sind.

5. Neue, verwendbare Provider werden markiert. Sie werden mit ihren zugehörigen Patterns assoziiert, welche Teil von $\mathbb{P}_A$ und ggf. auch von $\mathbb{C}_A$ werden. Für zuvor nicht verwendbare Provider (siehe 7 und 8) müssen zusätzlich die Assoziationen zwischen den nicht verfügbaren Consumed Patterns in $\mathbb{C}_U$ und ihnen gelöst werden.

6. Macht der neu verwendbare Provider Provided Patterns verfügbar, welche zuvor nicht verfügbar waren, so beginnt Phase II. Andernfalls endet die Analyse des Providers, da er keine Auswirkungen auf bereits vorhandene, nicht verwendbare Provider haben kann.

7. Alle Provider, deren Consumed Patterns nicht verfügbar waren, jedoch Teilmenge der durch den neuen Provider verfügbaren Provided Patterns sind, sind erneut zu überprüfen. Entsprechend wird die Menge $\mathbb{PRO}_{eff}$ der High-Level Provider bestimmt, auf welche sich die Verwendbarkeit des neuen Providers auswirken könnte.

8. Rekursiv wird diese Teilmenge in $\mathbb{C}_U$ registrierter Provider einer erneuten Verfügbarkeitsanalyse unterzogen. Die Rekursion endet automatisch, wenn keine nicht verwendbaren Provider gefunden werden, welche weitere Provided Patterns verfügbar machen könnten.

Zyklenvermeidung

Bei der Einfügung von $\mathbb{P}_i$ in $\mathbb{P}_A$ sind zyklische Abhängigkeiten zu verhindern, andernfalls könnten die resultierenden gegenseitigen Informationszugriffe der Provider den Context Service unbrauchbar machen bzw. stark beeinträchtigen. Zyklen könnten einerseits durch das Kontextmodell a priori ausgeschlossen, oder andererseits durch Analyse der $\mathbb{C}_i$ und $\mathbb{P}_i$ a posteriori erkannt werden. Die aktuelle Version des Context Services unterstützt allerdings keine Validierung anhand des Kontextmodells. Folglich ist zu prüfen, ob durch das Einfügen ein Zyklus entstehen würde, welcher dann ggf. vermieden bzw. beseitigt werden muss.

Zyklen – auch Deadlocks genannt – können mit Methoden der Graphentheorie erkannt werden. In diesem Fall wird ein einfacher Markierungsalgorithmus [Jia88] eingesetzt, welcher die Komplexität $\mathcal{O}(n^2)$ hat*. Für seine Anwendung benötigt man den Abhängigkeitsgraph der Provided und Consumed Patterns, welche die Knotenmenge des Graphen darstellen. Seine Kanten entstehen durch die „ist abhängig von"-Relation, welche Teil der Abhängigkeitsfunktion *dep* eines Providers ist.

Während der Verfügbarkeitsprüfung (Schritt 3) werden die Provided Patterns dem Graphen hinzugefügt. Anschließend wird die Zyklenerkennung durchgeführt. Ist diese positiv, d. h. wurde ein Zyklus gefunden, werden die vorläufig hinzugefügten Provided Patterns wieder entfernt. Bei einer negativen Prüfung verbleiben die Pattern dauerhaft im Graphen für die Prüfung weiterer, später installierter, Provider. In jedem Fall wird ein entsprechendes Ergebnis geliefert, um die Einfügung der Pattern in die Verwaltungsstrukturen zu steuern.

Alternativ zur hier eingesetzten Strategie der Zyklenvermeidung, d. h. Prüfung potentieller Auswirkungen i.V.m. eventueller Einfügung, ist auch eine Zyklenbeseitigung möglich. Dabei würde die Erkennung der Zyklen analog erfolgen. Jedoch bestünden bei der Wahl der zu entfernenden Patterns weitere Möglichkeiten zur Anpassung des Systems. So könnten Patterns für die Entfernung aus dem Zyklus selektiert werden, welche wenig/gar nicht benutzt werden oder deren Entfernung nur

* n ist die Anzahl der Knoten des Graphen.

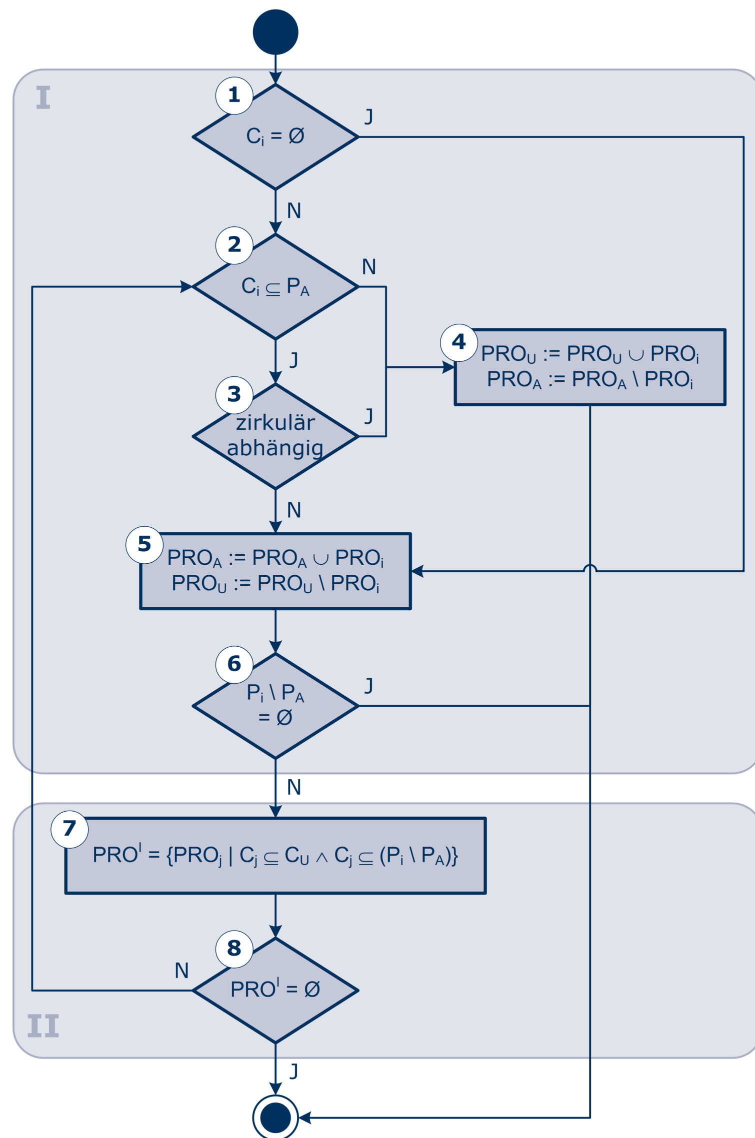

Abbildung 5.9.: Behandlung eines gestarteten Context Providers

ein Minimum des Graphen unverfügbar machen würde. Bei einem solchen Ansatz müsste eine Bewertungsfunktion definiert werden, um die Knoten des Graphen entsprechend zu gewichten. Aus Komplexitätsgründen wird dieser Ansatz jedoch nicht verfolgt.

Eine vollständige Alternative zur beschriebenen Zyklenerkennung bzw -vermeidung ist eine modellbasierte Lösung. Diese ist in grundsätzlich zwei Varianten möglich. Die erste Variante setzt voraus, dass zur Beschreibung von Providern ein zyklenfreies, domänenspezifisches Kontextmodell vorliegt. Bei der Analyse von Providern wäre durch den Context Service lediglich deren Beschreibung formal gegenüber dem Modell zu validieren. In einer zweiten Variante müsste ein ausdrucksstarkes Metamodell – z. B. OWL-basierte Ontologien – eingesetzt werden. Darauf basierende Providerbeschreibungen würden durch den Context Service interpretiert. Bei der Wahl einer Variante ist zu berücksichtigen, dass die Erste weniger Unterstützung durch den Context Service benötigt, und dementsprechend auch weniger Ressourcen der eingesetzten Geräte. Jedoch müssen die Entwickler selbst die Zyklenfreiheit des Modells sicher stellen – ggf. mit Unterstützung durch geeignete Entwicklungswerkzeuge. Die zweite Variante ist voraussichtlich wesentlich flexibler in der Beschreibung der Provider, jedoch setzt sie umfangreiche Interpretationssysteme (z. B. logikbasierte Reasoner) voraus, deren Einsatz einen hohen Ressourcenbedarf und große Herausforderungen bei der Integration mit sich bringt. Aus den soeben genannten Gründen wurde eine einfache und effiziente Lösung eingesetzt, welche zwar zwangsläufig auf einem Modell basiert, aber mögliche Fehler bei dessen Einsatz durch Verwendung der Zyklenerkennung vermeidet.

5.4.5. Provider gestoppt

Stellt das System fest, dass ein Provider gestoppt bzw. deinstalliert wurde, so ist dieser aus der Verwaltung zu entfernen. Provider werden normalerweise nur dann gestoppt*, wenn sie nicht mehr auf ihre Kontextquelle zugreifen können. Auf diese Weise werden Fehler bzw. unerwünschtes Verhalten aufgrund der Nutzung des Providers nach dem Verlust seiner Kontextquelle vermieden.

Die Behandlung des Stopps (Abb. 5.10 auf der nächsten Seite) besteht, wie auch die des Starts, aus zwei Phasen: I (1 - 4) für die Behandlung des Providers selbst, II (5 - 7) für die Behandlung eventueller Auswirkungen auf verwaltete Provider:

1. Zuerst wird festgestellt, ob die Provided Patterns des Providers vor seinem Stopp verfügbar waren. War dies der Fall, so sind mögliche Effekte festzustellen und ggf. zu behandeln – weiter mit 3.

*Aktuell ist die Deinstallation noch Aufgabe des Providers selbst. Später kann dies auch automatisch geschehen.

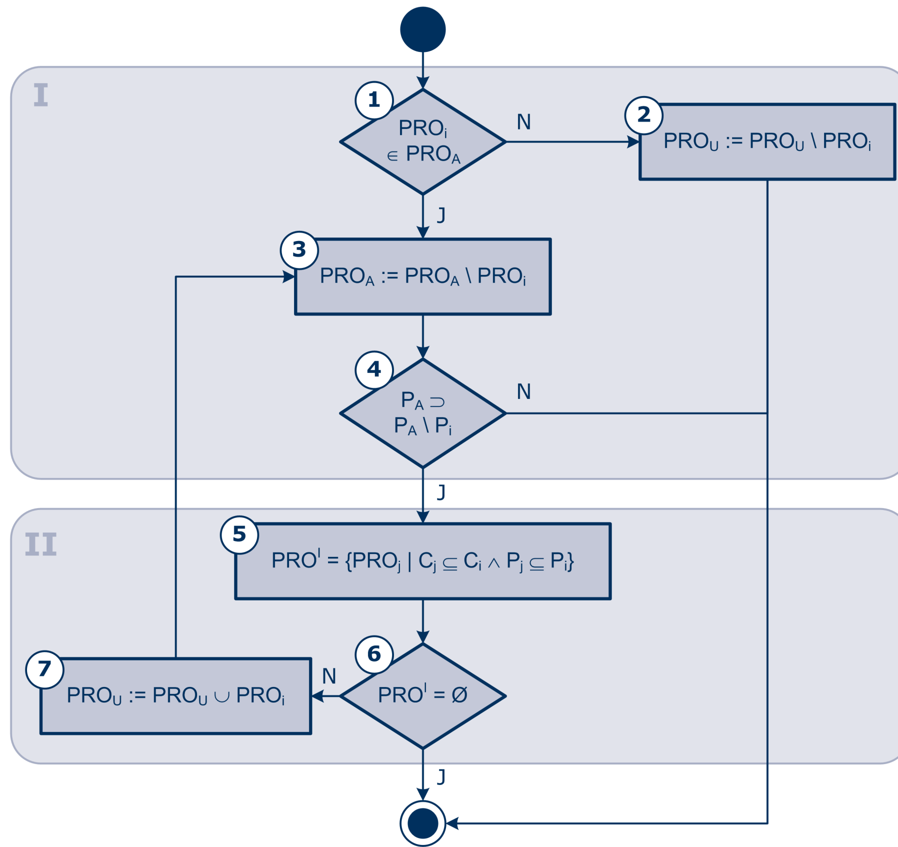

Abbildung 5.10.: Behandlung eines gestoppten Context Providers

2. Ein bisher nicht verwendbarer Provider muss lediglich aus der Verwaltung entfernt werden, d. h. $\mathbb{PRO}_U := \mathbb{PRO}_U \setminus PRO_i$. Dazu gehört auch, dass die Assoziationen seiner Patterns aus $\mathbb{P}_U$ und $\mathbb{C}_U$ zu ihm beseitigt werden. Führt das dazu, dass ein Pattern keine Assoziationen zu einem Provider mehr hat, so ist es zu entfernen. Weil keine Abhängigkeiten verwendbarer Provider von einem nicht verwendbaren Provider bestehen konnten, hat seine Entfernung keine weiteren Folgen – die Behandlung wird beendet.

3. Ein bisher verwendbarer Provider muss ebenfalls aus der Verwaltung entfernt werden, hier jedoch mit $\mathbb{PRO}_A := \mathbb{PRO}_A \setminus PRO_i$. Wie in 2. sind die Mengen $\mathbb{P}_A$ und $\mathbb{C}_A$ zu bereinigen.

4. Es wird festgestellt, ob die Bereinigung zur Reduzierung der verfügbaren Provided Patterns geführt hat – $\mathbb{P}_A \supset \mathbb{P}_A \setminus \mathbb{P}_i$. War dies nicht der Fall, d. h.

$\mathbb{P}_A = \mathbb{P}_A \setminus \mathbb{P}_i$, hat die Entfernung keine Auswirkung auf die verbleibenden verwendbaren Provider – die Behandlung wird beendet.

5. Die Menge der vom Stopp von PRO_i betroffenen Provider wird ermittelt.

6. Alle betroffenen Provider müssen – ähnlich wie ein gestoppter Provider – behandelt werden, denn durch ihre Abhängigkeit von seinen Provided Patterns werden sie nicht-verwendbar.

7. Weil die betroffenen Provider jedoch noch installiert und gestartet sind, werden sie nicht entfernt, sondern als nicht verwendbar registriert.

5.4.6. Bemerkungen

Im Sinne der Vollständigkeit sei bemerkt, dass selbst Low-Level Provider nicht unabhängig sind. Sind die von ihnen benötigten Kontextquellen nicht verfügbar, so können sie deren Kontextinformationen nicht bereitstellen. Jedoch ist die Abhängigkeit zwischen einer Ressource und einem Low-Level Provider wesentlich einfacher, als die zwischen mehreren High-Level Providern. Die Verwaltung der Provider wird durch die Berücksichtigung der Kontextquellen komplexer, weil bei der Behandlung des Starts eines Low-Level Providers die Verfügbarkeit seiner Kontextquelle(n) zu prüfen ist. Zusätzlich ist eine dauerhafte Überwachung benötigter Kontextquellen erforderlich. Bei der Schaffung einer entsprechenden Infrastruktur sind Komponenten notwendig, welche die quellenspezifische Suche und Überwachung vornehmen, und die auftretenden Ereignisse über einheitliche Schnittstellen an den Context Service bzw. die Provider melden. In wieweit sich der Implementierungsaufwand für diese Überwachungskomponenten von dem Implementierungsaufwand für einen Provider unterscheidet, ist äußerst fragwürdig, weil beide Komponenten quellenabhängig sind. Weiterhin mach eine generische Ressourcensuche und -überwachung nur dann Sinn, wenn die Zahl der Ressourcen- oder Quellentechnologien niedriger ist, als die der Provider. Andernfalls werden eine Vielzahl von Überwachungskomponenten betrieben, um eine kleine Menge von Low-Level Providern zu bedienen.

Abweichend von der bisherigen Darstellung von High-Level Providern ist es, je nach Implementierung des Providers, theoretisch vorstellbar, dass nicht jedes der Elemente aus $\mathbb{P}_i$ vollständig von allen Elementen in $\mathbb{C}_i$ abhängig ist. Es wären folglich Situationen möglich, in denen ein Provider *teilweise verwendbar* ist, d. h. ein dritter Zustand wäre notwendig. Dieser Zustand könnte durch Verwaltung virtueller Provider vermieden werden. Dabei würde ein realer High-Level Provider PRO_i – im Rahmen der Provideranalyse – in m virtuelle Provider PRO_{ij} mit $1 \leq j \leq m$ zerlegt, sodass deren $\mathbb{P}_{ij}$ disjunkt sind, und alle $P_{ijk} \in \mathbb{P}_{ij}$ vollständig von den $C_{ijl} \in \mathbb{C}_{ij}$ abhängig sind. Aufgrund der vollständigen Abhängigkeit wäre ein virtueller Provider

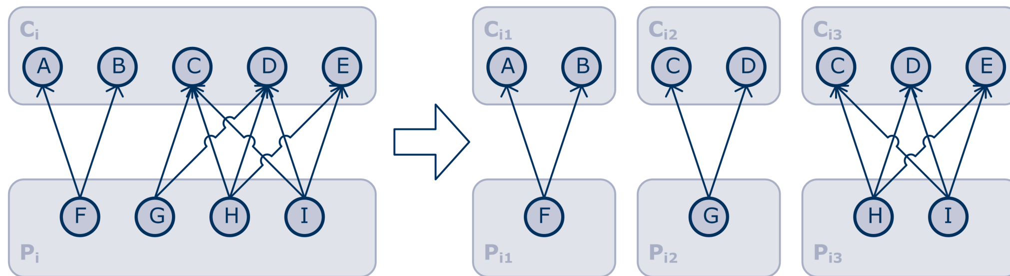

Abbildung 5.11.: Zerlegung eines realen High-Level Context Providers in drei virtuelle Provider

nur dann verwendbar, wenn alle C_{ijl} verfügbar sind. Dadurch ist die zuvor beschriebene Verwaltung wieder ohne Einschränkungen anwendbar. Abbildung 5.11 zeigt ein Beispiel für die Zerlegung eines realen Providers in drei Virtuelle. Sie verdeutlicht auch, dass die entstehenden $\mathbb{C}_{ij}$ unterschiedlich sind, jedoch nicht disjunkt sein müssen – z. B. $\mathbb{C}_{i2} \subset \mathbb{C}_{i3}$.

5.5. Zentrale Providerverwaltung

Mit den Mechanismen aus dem vorangegangenen Kapitel kann lediglich ein Context Service realisiert werden, welcher lokalen Context Consumern die Kontextinformationen der lokal verwendbaren Context Provider zur Verfügung stellt. Kontextinformationen anderer, per entfernter Kommunikation erreichbarer, Knoten sind nicht nutzbar. Um dies zu ändern, muss jeder Knoten in die Lage versetzt werden, auf die Gesamtheit der verwendbaren Provider aller erreichbaren Knoten zuzugreifen – eine verteilte Form der zuvor diskutierten lokalen Verwaltung ist erforderlich.

Vorerst, d. h. in diesem Kapitel wird die verteilte Verwaltung durch einen zentralisierten Ansatz erreicht. Folglich gibt es in der Menge der vernetzten Knoten eine speziellen Knoten – den *Server* – welcher eine besondere Rolle bei der Verwaltung der Provider einnimmt. Alle übrigen Knoten – die *Clients* – nutzen die Funktionalität des Servers. Diese besteht in der Übernahme der Verwaltung der Provider, welche zuvor lokal von jeder Instanz des Context Service vorgenommen wurde. Die Clients benachrichtigen den Server über gestartete und gestoppte Provider, während dieser die Provider analysiert, deren Verwendbarkeit ermittelt, und das Ergebnis, d. h. die Mengen verwendbarer und nicht-verwendbarer Provider $\mathbb{PRO}_A$ und $\mathbb{PRO}_U$ an die Clients verteilt. Der Server besitzt zu jeder Zeit das Wissen über die verwendbaren und nicht-verwendbaren Provider auf allen Clients – den *globalen Zustand*.

5.5.1. Erreichbarkeit

Koppelt man mehrere Instanzen des Context Service, sodass deren Consumer ortstransparent auf Kontextinformationen der Provider anderer Knoten zugreifen können, kommt zu den bisher betrachteten Ursachen der dynamischen Verfügbarkeit eine Weitere hinzu – das Auftreten von Kommunikationsstörungen. Solche Probleme können theoretisch in Netzen jeglicher Art auftreten, wenn diese überlastet sind, und die an der Übertragung beteiligten Knoten folglich Nachrichten verwerfen müssen. Häufiger sind Störungen jedoch in drahtlosen Netzen, aufgrund der Offenheit des Übertragungsmediums. Sie ermöglicht Wechselwirkungen des Netzes mit anderen Netzen derselben oder ähnlicher Technologien, sowie mit physikalischen Vorgängen in der Umgebung. Sind drahtlos kommunizierende Knoten darüber hinaus mobil*, so erhöht sich die Wahrscheinlichkeit von Störungen weiter. Infolge dessen ist die Überwachung der Erreichbarkeit der Knoten entfernter Context Provider in den betrachteten Szenarien unerlässlich.

Die Überwachung ist für beide Seiten erforderlich:

1. Jeder Client muss sicher stellen, dass jederzeit die Möglichkeit besteht, Nachrichten mit aktuellen Informationen über verwendbare entfernte Provider zu erhalten.

2. Der Server muss feststellen, ob die Clients die Möglichkeit haben, ihn zu benachrichtigen, um den globalen Zustand zu aktualisieren. Weiterhin wird somit die gegenseitige Erreichbarkeit der Clients ermittelt.

Um ihre Erreichbarkeit zu prüfen, schickt der Server in regelmäßigen Abständen t_{PING} Nachrichten (Kap. 5.5.4) an alle Clients. Erhält er innerhalb einer festgesetzten Zeitspanne t_{PONG} eine Antwort, so wird der Client als erreichbar betrachtet. Aufgrund intermittierender Störungen können Clients kurzzeitig unerreichbar sein. Deshalb wird ein Client erst dann als unerreichbar betrachtet, wenn eine festgesetzte Anzahl m_{PING} von aufeinander folgenden Nachrichten unbeantwortet bleibt.

Clients müssen keine aktive Erreichbarkeitsprüfung vornehmen, weil sie die Nachrichten des Servers als Bestätigung seiner Erreichbarkeit werten können. Passiv müssen sie jedoch den gleichen Algorithmus mit denselben Werten für t_{PING}, t_{PONG} und m_{PING} anwenden, um nach einer festgesetzten Menge ausbleibender Nachrichten den Server als unerreichbar betrachten zu können.

Um den Aufwand der Überwachung zu reduzieren, werden auch reguläre Nachrichten, welche zum Datenaustausch an den Server oder die Clients versendet werden,

*Hier ist vorwiegend Mobilität relativ zu den übrigen Kommunikationspartnern bzw. Knoten gemeint. Die Mobilität einer ganzen Gruppe gegenüber der Umgebung (z. B. Laptops von Passagieren an Bord eines Zuges), ohne Kommunikationsbeziehungen zu Knoten der Umgebung und ohne Mobilität innerhalb der Gruppe, ist nicht von Interesse.

vom jeweiligen Empfänger als Zeichen der Erreichbarkeit gewertet. Treffen solche Nachrichten während t_{PING} ein, so beginnt diese von vorn. Bei großer Häufigkeit von Zustandsänderungen lokaler Provider sind u. U. gar keine zusätzlichen Nachrichten zur Überwachung erforderlich, weil die Propagierung der Änderungen an den Server und die Verteilung ihrer Auswirkungen an alle übrigen Clients die Erreichbarkeit implizit.

Trotz der Überwachung müssen Kommunikationsstörungen beim gegenseitigen Zugriff der Clients auf Kontextinformationen behandelt werden. Ebenso wie der Server intermittierende Störungen bis zu einem bestimmten Maß toleriert, muss dies auch für die Kommunikation der Clients untereinander gelten. Andernfalls können derartige Inkonsistenzen zu zwei Formen des Fehlverhaltens führen:

- Der Server betrachtet Clients bereits als unerreichbar, obwohl diese selbst es noch gar nicht tun. Folglich werden unerreichbare Provider entsprechend an die Clients signalisiert und von ihnen nicht mehr genutzt, obwohl die Clients die Störungen tolerieren würden. Die Verfügbarkeit von Kontextinformationen würde demzufolge unnötig eingeschränkt.

- Der Server betrachtet Clients noch als erreichbar, obwohl sie es schon nicht mehr sind. Entsprechend müssen die Clients bei den erfolglosen Zugriffen unnötig viele Störungen behandeln. Es entsteht ein erhöhter Aufwand des Systems, welcher Betriebsmittel verbraucht, ohne zu einem Nutzen für Consumer oder Provider zu führen.

Obwohl der letzte Fall als weniger kritisch eingeschätzt wird, ist eine konsistente Konfiguration* der Überwachungsmechanismen von Clients und Server anzustreben.

5.5.2. Clients

Unabhängig von der Existenz einer verteilten Verwaltung ist auf jedem Gerät, welches Kontextquellen bereitstellt, eine Context Service Instanz für die Überwachung der zugehörigen lokalen Context Provider zuständig, weil diese nicht auf direktem Wege entfernt zugreifbar sind. Werden während der Überwachung Provider gestartet oder gestoppt, so werden diese Änderungen an den Server übertragen, sodass eine Aktualisierung des globalen Zustandes erfolgen kann.

Damit die Benachrichtigung des Servers erfolgen kann, muss der Client zuerst den Server finden – Zustand „Suche“ in Abbildung 5.12 auf der nächsten Seite. In dieser Arbeit wird der Server aufgrund seiner statischen IP†-Adresse „gefunden“, welche Teil der Konfigurationsdaten eines jeden Clients ist. Alternativ könnte ihm

*Das heißt: gleiche Parameter t_{PING}, t_{PONG} und m_{PING} für alle Clients und ihren Server.

†Internet Protocol (IP)

ein statischer Name zugeordnet werden, welcher durch einen Namensdienst auf eine gültige Adresse abgebildet wird – z. B. mittels DNS*.

Hat der Client den Server gefunden, meldet er sich bei diesem mit einer entsprechenden Nachricht an (Kap. 5.5.4). Jetzt ist er bereit, lokale Änderungen der Provider an diesen Server zur Aktualisierung des globalen Zustandes zu melden bzw. von ihm derartige Aktualisierungen zu empfangen. Nur in diesem Zustand „Aktualisierung (lokal)" ist der Client in der Lage auf den Server und somit auch die Provider anderer Clients zuzugreifen. Wird die Verbindung zum Server absichtlich oder unabsichtlich unterbrochen, kehrt der Client wieder zur „Suche" zurück, und kann ausschließlich lokale Provider nutzen.

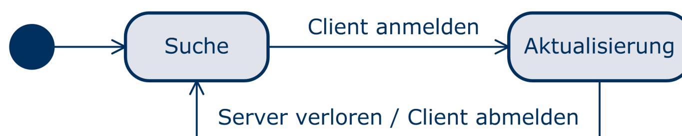

Abbildung 5.12.: Zustandsdiagramm eines Context Service Clients

5.5.3. Server

Zusätzlich zu der von jedem Client vorgenommenen Überwachung der lokalen Provider, führt der Server die Verwaltung der gestarteten Provider aller ihm bekannten Knoten durch. Es ist die Aufgabe der Clients, den Server zu suchen und sich selbständig bei ihm anzumelden. Deshalb befindet sich der Server nach seinem Start sofort im Zustand „Betrieb" (Abb. 5.13) und wartet auf Nachrichten der Clients.

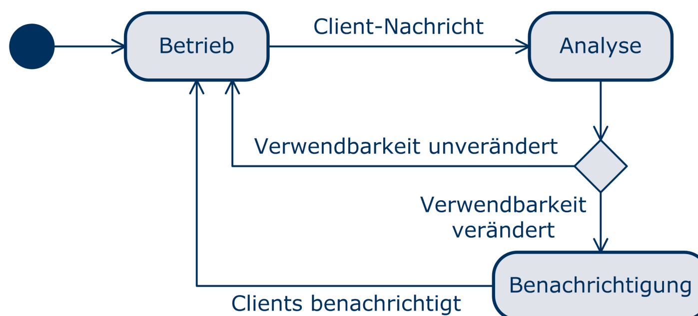

Abbildung 5.13.: Zustandsdiagramm des Context Service Servers

Erhält er eine Nachricht von einem Client, so wird diese analysiert. Wird als Ergebnis der „Analyse"-Phase eine Veränderung der Menge verwendbarer Provider

*Domain Name System (DNS)

festgestellt, so werden die betroffenen Clients benachrichtigt, d. h. der globale Zustand wird aktualisiert und verteilt.

5.5.4. Nachrichten

Die Nachrichten, welche zwischen Clients und Server ausgetauscht werden, können in drei Klassen unterteilt werden: *Erreichbarkeits-*, *Provider-* und *Zugriffsnachrichten*. Ihre einzelnen Vertreter sind in den Tabellen 5.2 bis 5.4 auf den Seiten 120–122 aufgeführt, und werden nachfolgend näher beschrieben. Die Spalte „Inhalt“ gibt an, welche Daten für die „Funktion“ der Nachricht wesentlich sind.

Grundsätzlich enthält jede Nachricht mindestens folgende Daten:

- *SendID* – Die eindeutige Adresse des sendenden Knotens – z. B. eine IP-Adresse.
- *RecvID* – Die eindeutige Adresse des Knotens, für den die Nachricht bestimmt ist.
- *SeqNo* – Eine fortlaufende Nummer, welche zur Ordnung gesendeter Nachrichten verwendet wird, und demzufolge Sender-lokal eindeutig sein muss.

Die ohnehin in jeder Nachricht enthaltenen Informationen SendID und RecvID werden werden in den Tabellen nur dann dargestellt, wenn sie für die Verarbeitung der Nachricht erforderlich sind – sonst werden sie ignoriert. Bis auf die Erreichbarkeitsnachrichten enthalten alle übrigen Nachrichten Nutzdaten, welche in Extensible Markup Language (XML) kodiert übertragen werden. Die Datei mit der XML Schema Definition (XSD) der XML-kodierten Nachrichteninhalte ist in Anhang A.1 dargestellt.

Erreichbarkeitsnachrichten (Tab. 5.2 auf Seite 120) dienen der Überwachung der Knoten und somit auch der Verwaltung des Netzwerkes der Instanzen des Context Service. Clients müssen sich, durch Senden einer SIGNON-Nachricht, aktiv für die Überwachung registrieren. Ist vorhersehbar, dass ein Client zukünftig nicht mehr erreichbar sein wird – z. B. weil der zugehörige Knoten abgeschaltet wird – so kann er sich durch eine SIGNOFF-Nachricht *aktiv* abmelden. Erreichbarkeitsnachrichten werden nur zwischen Client und Server ausgetauscht.

Während der Überwachung versendet der Server im Intervall von t_{PING} PING-Nachrichten zur Prüfung der Erreichbarkeit der Clients (Kap. 5.5.1). Diese müssen innerhalb des Intervalls t_{PONG} mit PONG-Nachrichten – unter Wiederverwendung der SeqNo der PING-Nachricht – antworten. Bleibt diese Antwort mehrfach (m_{PING}) aus, so wird der Client nach $t_{SIGNOFF} = m_{PING} \times t_{PING}$ *passiv*, d. h. durch den Server, abgemeldet. Aufgrund der Abmeldung – egal ob passiv oder aktiv

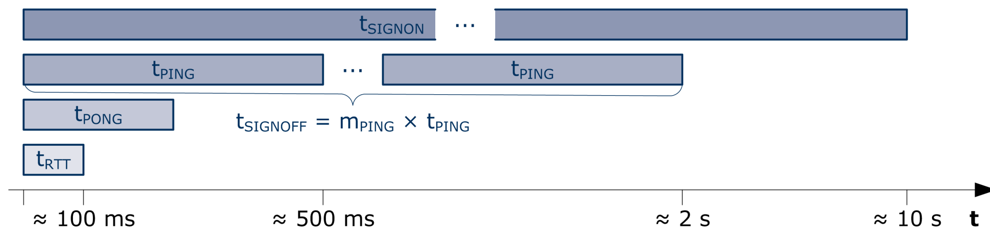

Abbildung 5.14.: Relationen zwischen Nachrichtenumlaufzeit t_{RTT}, Erreichbarkeitsintervall t_{PING}, Antwortintervall t_{PONG} und Antwortschwelle m_{PING}

– erhält ein Client keinerlei Nachrichten des Servers mehr. Um die eigene passive Abmeldung erkennen zu können, muss jeder Client das Eintreffen von PING-Nachrichten überwachen, und dabei dieselben Werte für t_{PING}, t_{PONG} und m_{PING} anwenden wie der Server. Der Wert von t_{PONG} sollte über der zu erwartenden Nachrichtenumlaufzeit* liegen, genauso wie die Zeit zwischen den Erreichbarkeitstests signifikant über t_{PONG} liegen sollte (Gl. 5.9 und Abb. 5.14).

$$t_{PING} \gg t_{PONG} > 2 \times t_{RTT} \tag{5.9}$$

Ist ein Client passiv abgemeldet (oder noch gar nicht angemeldet gewesen), so sendet er regelmäßig nach Ablauf des Anmeldeintervalls $t_{SIGNON} \gg t_{SIGNOFF}$ eine SIGNON-Nachricht, bis die Anmeldung erfolgreich war.

Providernachrichten (Tab. 5.3 auf Seite 121) werden zur globalen Verwaltung der Provider ausgetauscht. Wie Erreichbarkeitsnachrichten werden sie nur zwischen Client und Server eingesetzt. Nach dem Start oder Stopp von Providern überträgt der verwaltende Client deren Beschreibungen an den Server. Dieser integriert die Informationen in den globalen Zustand, ermittelt die Auswirkungen und verteilt die Änderungen (neu verwendbare bzw. nicht mehr verwendbare Provider) mit dem gleichen Nachrichtentyp an alle Clients.

Um den Zustand von Clients zu synchronisieren, welche sich neu oder nach einem Verbindungsabbruch erneut am Server angemeldet haben, erhält jeder dieser Clients eine Providernachricht mit den verwendbaren Providern aller übrigen Clients. Diese erste UPDPRO-Nachricht wird als Antwort auf die SIGNON-Nachricht verschickt.

Jede UPDPRO-Nachricht enthält zwei Listen unterschiedlicher Tupel. Die einen Tupel bestehen aus der Adresse des betreffenden Clients und den Beschreibungen seiner verwendbaren Provider – die anderen Tupel enthalten ebenfalls die Clientadresse,

*Round Trip Time (RTT)

Tabelle 5.2.: Erreichbarkeitsnachrichten zur Überwachung der Clients und des Servers, sowie zur Verwaltung des Netzwerkes

Typ	Funktion	Inhalt	Behandlung
SIGNON	Registrierung eines Clients beim Server	SendID	Der Client (SendID) wird fortan überwacht. Änderungen der global verwendbaren und nicht verwendbaren Provider werden an ihn weitergeleitet. Beantwortung mit einer ersten Providernachricht.
SIGNOFF	De-Registrierung eines Clients vom Server	SendID	Die Überwachung des Clients wird beendet. Seine bisher verwendbaren Provider werden aus der Verwaltung entfernt und abhängige Clients ggf. per UPDPRO-Nachrichten informiert.
PING	Überprüfung der Erreichbarkeit eines Clients durch den Server	SendID, SeqNo	Implizite Wertung seitens des Clients als Erreichbarkeit des Servers und Beantwortung mit PONG-Nachricht unter Wiederverwendung der SeqNo.
PONG	Bestätigung der Erreichbarkeit durch den Client	SendId, SeqNo	Wertung seitens des Servers als Erreichbarkeit des Clients. Ausbleibende PONG-Nachrichten werden als Unerreichbarkeit gewertet.

aber nur die Bezeichner der nicht-verwendbarer Provider. Folglich enthalten Providernachrichten, welche von einem Client versendet werden, pro Liste jeweils nur ein Tupel, weil jeder Client immer nur den Start oder Stopp seiner eigenen Provider signalisieren kann. Dem gegenüber können vom Server gesendete Nachrichten mehrere Tupel mit den Beschreibungen bzw. Bezeichnern der Provider mehrerer Clients enthalten.

Zugriffsnachrichten (Tab. 5.4 auf Seite 122) dienen, wie der Name impliziert, dem Zugriff auf die Kontextinformationen eines entfernten Providers. Im Gegensatz zu den zuvor genannten Nachrichten, werden sie nur zwischen Clients ausgetauscht. Besitzt der Server eigene lokale Provider oder Consumer, so kann er in der Rolle eines Clients ebenfalls Zugriffsnachrichten austauschen.

Tabelle 5.3.: Providernachrichten zur Verteilung der Beschreibungen verwendbarer und nicht-verwendbarer Provider

Typ	Funktion	Inhalt	Behandlung
UPDPRO	Mitteilung (a) gestarteter und gestoppter durch einen Client oder (b) verwendbarer und nicht-verwendbarer Provider durch den Server	{(HostId, {Provider})}, {(HostId, {ProviderId})}	(a) seitens des Servers: De-/Registrierung der Provider und Analyse der Auswirkungen, sowie Verteilung der Änderungen (b) seitens der Clients: De-/Registrierung der Provider

Synchroner Zugriff wird durch Senden einer REQ-Nachricht angefordert. Die angefragten Kontextinformationen werden in einer RESP-Nachricht an den anfragenden Client geschickt, wobei die Sequenznummer der REQ-Nachricht wiederverwendet wird. Asynchroner Zugriff wird durch eine SUB-Nachricht eingeleitet, und mit einer TICKET-Nachricht bestätigt. Bei Kontextänderungen, welche dem Pattern einer SUB-Nachricht entsprechen, wird der registrierte Client mit einer NOTIFY-Nachricht, unter Wiederverwendung des Inhalts der zugehörigen SUB-Nachricht, benachrichtigt. Durch eine UNSUB-Nachricht mit dem Ticket kann die Benachrichtigung bei Kontextänderung aufgehoben werden.

5.5.5. Hybrider Austausch von Providerbeschreibungen

Schon bei der Analyse des Anwendungsszenarios (Kap. 3.6) wurde erkennbar, dass wesentlich weniger Kontextsenken als -quellen existieren, und somit nur ein Bruchteil der Kontextquellen genutzt werden kann. Dies gilt nicht nur inhaltlich, sondern auch temporal und spatial, weil die Kontextsenken meist für begrenzte Zeit an den Kontextinformationen eines begrenzten räumlichen Bereiches interessiert sind. Vor diesem Hintergrund ist es notwendig die unnötige Übertragung von Providerbeschreibungen zu verhindern, d. h. die Übertragung von Beschreibungen, deren Provider generell oder momentan gar nicht von einem Consumer benötigt werden. Deshalb sind Alternativen zur gezeigten proaktiven Verteilung der Providerbeschreibungen zu diskutieren. Bevor dies jedoch erfolgt, sollen die wesentlichen Eigenschaften der proaktiven Verteilung von Providerbeschreibungen aufgezeigt werden.

Tabelle 5.4.: Zugriffsnachrichten zum Austausch von Kontextinformationen entfernter Knoten und ihrer Provider

Typ	Funktion	Inhalt	Behandlung
REQ	synchrone Abfrage von Kontext	Pattern	lokale Ermittlung des durch das Pattern spezifizierten Kontexts und Beantwortung mittels RESP-Nachricht
RESP	synchrone Auslieferung von Kontext	Pattern, {Kontext-element}	Übergabe des Kontexts an den lokalen Konsumenten
SUB	Registrierung für asynchrone Auslieferung von Kontext	Pattern	lokale Registrierung für Benachrichtigung bei Kontextänderung, und sofortige Beantwortung durch TICKET-Nachricht
TICKET	Bestätigung der Registrierung asynchroner Auslieferung	Pattern, Ticket	Registrierung für (a) spätere Weiterleitung des Kontexts an Konsumenten und (b) Verwendung in einer UNSUB-Nachricht
UNSUB	Beendigung asynchroner Auslieferung	Ticket	De-Registrierung der Auslieferung von Kontext
NOTIFY	asynchrone Auslieferung von Kontext	Ticket, {Kontex-tereignis}	Weiterleitung der Ereignisse an lokale Konsumenten (Zuordnung durch Ticket)

Bewertung der proaktiven Verteilung

Durch den proaktiven Charakter der Lösung verfügt jeder Client jederzeit über den globalen Zustand – die Beschreibungen aller verwendbaren Provider. Er kann sie bei Bedarf sofort nutzen. Die Selektion von Providern wird in diesem Fall folglich lokal auf dem Knoten durchgeführt, und benötigt somit nur wenig Zeit. Die Anfragelatenz, d. h. Verzögerung bei der Bedienung einer Consumer-Anfrage, der proaktiven Verteilung $t_{REQ_{pro}}$ entspricht folglich im Wesentlichen der *Selektionslatenz* t_{SEL}. Unter der Annahme $t_{SEL} \approx 10\mu s$ kann die Verzögerung vernachlässigt werden. Sie ist derart niedrig, weil die Selektion des geeigneten Providers lokal auf dem Gerät der Instanz des Context Service erfolgt.

Weil die Beschreibungen erst verteilt werden müssen, bevor sie genutzt werden können, spiegeln die lokal gespeicherten Beschreibungen nicht den aktuellen, globa-

len wider, sondern den Zustand, welcher beim Absenden der zugehörigen Nachrichten auf den entfernten den Knoten herrschte. Somit entsteht eine *Aktualisierungslatenz* $t_{UPD_{pro}}$ bei der Verteilung des globalen Zustandes. Sie setzt sich zusammen aus: (1) der Übertragung der Providernachrichten an den Server $t_{C/S}$, (2) ihrer Verarbeitung t_{CHK} und (3) der anschließenden Verteilung resultierender Providernachrichten an die Clients $t_{S/C}$. Selbst wenn man eine effiziente Verteilung an alle Clients mittels Multicast unterstellt, ergibt sich eine Aktualisierungslatenz (Gl. 5.10), welche die Größenordnung der durchschnittlichen Nachrichtenumlaufzeit (engl. RTT) hat (Annahme: $t_{RTT} \approx 100ms$).

$$t_{UPD_{pro}} = t_{C/S} + t_{CHK} + t_{S/C} = t_{RTT} + t_{CHK} \tag{5.10}$$

Die Aktualisierungslatenz kommt jedoch nur in zwei Fällen zum Tragen: (1) wenn die Providernachrichten den Ausfall eines selektierten Providers signalisieren, und (2) wenn sie die Verfügbarkeit eines alternativen Providers signalisieren. Wie oft diese Fälle eintreten, hängt somit einerseits von der dynamischen Verfügbarkeit jedes einzelnen Providers und von der Existenz alternativer bzw. redundanter Provider ab. Redundanz entsteht durch Überlappung der beobachteten Bereiche von Providern, und gilt vorwiegend für Quellen die niederwertigen Kontext liefern (z. B. Sensoren). Eine nähere Betrachtung der Ausfallwahrscheinlichkeiten wäre nur für ein konkretes Szenario möglich, und lieferte keine Aussage über die Allgemeinheit.

Der Kommunikationsaufwand für die proaktive Verteilung ist sehr hoch, weil jede Zustandsänderung eines Providers in einer Nachricht an den Server resultiert, welche zur Aktualisierung aller Clients führen muss. Dieser *Aktualisierungsaufwand* $m_{UPD_{pro}}$ hat folglich eine Komplexität von $\mathcal{O}(n)$. Weil aber voraussichtlich nur ein Bruchteil der Clients bzw. ihrer Consumer die signalisierten Provider nutzt, ist die Effizienz dieser Aktualisierungsstrategie fragwürdig.

Tritt ein Client dem verteilten Context Service erst verspätet bei, oder ist er nach einer Verbindungsunterbrechung wieder erreichbar, so muss sein lokaler Zustand mit dem globalen Zustand synchronisiert werden. Dies erfolgt, indem er als Antwort auf seine Beitrittsnachricht eine Providernachricht mit allen verwendbaren Providern erhält. Aufgrund der Vielzahl existierender Provider ist diese Nachricht sehr groß. Gleiches gilt für den lokalen Platzbedarf und Verwaltungsaufwand der übertragenen Providerbeschreibungen. Jeder Client speichert jederzeit die Beschreibungen aller verwendbaren entfernten Provider, sowie seiner lokalen Provider. Die Speicherung der Informationen ist genauso ineffizient wie die ihnen voraus gehende Übertragung.

Somit hat die proaktive Verteilung nur einen Vorteil – eine sehr niedrige Latenz – aber zwei Nachteile – unnötig häufige Verteilung von Providernachrichten an unnötig viele Clients, sowie und deren Volumen beim Beitritt eines Clients. Beide Nachteile gelten unabhängig von der Nutzung des Systems durch Consumer, d. h. selbst wenn kein einziger Provider verwendet wird. Folglich ist der Aufwand im

„Leerlauf“ unvertretbar hoch.

Eigenschaften der reaktiven Abfrage

Alternativ zur proaktiven Verteilung werden die Providerbeschreibungen rein bedarfsabhängig ausgetauscht. Demzufolge werden Clients erst dann geeignete Provider beim Server erfragen, wenn sie den Informationsbedarf ihrer lokalen Consumer nicht mit ihren eigenen, lokalen Providern bedienen können. Eine Abfrage durch einen Client C_i führt zur Abfrage aller übrigen $n-1$ Clients durch den Server S, der Verarbeitung ihrer Antworten, und schließlich zur Übertragung der gesammelten Beschreibungen an den abfragenden Client.

Als Nebeneffekt steigt die *Anfragelatenz* $t_{REQ_{re}}$ – im Vergleich zur proaktiven Verteilung um die Nachrichtenumlaufzeit, sowie die *Aktualisierungslatenz* $t_{UPD_{re}}$ (siehe Gl. 5.11).

$$\begin{aligned} t_{REQ_{re}} &= t_{SEL} + t_{C_i/S} + t_{S/C_{n-1}} + t_{C_{n-1}/S} + t_{CHK} + t_{S/C_i} \\ &= t_{SEL} + 2 \times t_{RTT} + t_{CHK} \\ &= t_{SEL} + t_{RTT} + t_{UPD_{re}} \end{aligned} \tag{5.11}$$

Die Häufigkeit des Nachrichtenaustauschs im reaktiven Fall ist wesentlich niedriger als im proaktiven Fall, weil Providernachrichten nur dann ausgetauscht werden, wenn sie tatsächlich benötigt werden. Bei einer langfristigen Nutzung von Kontextinformationen im Rahmen von Abonnements ist dies weniger relevant, weil dabei zusätzlich zur Überwachung des Kontexts auch die genutzten, sowie alternative Provider überwacht werden müssen, um deren dynamische Verfügbarkeit behandeln zu können. Folglich werden Providernachrichten bei Abonnements über einen längeren Zeitraum ausgetauscht – die Ersparnis wird während der Dauer der Abonnements reduziert.

Generell werden Nachrichten jedoch nur an diejenigen Clients geschickt, welche sie tatsächlich benötigen – nicht pauschal an alle Clients. Bei der quantitativen Betrachtung (Kap. 3.5.2) wurde von der Existenz von ca. 21 Mio. Kontextquellen, sowie 125.000 Kontextsenken ausgegangen. Das ist ein Übergewicht der Quellen gegenüber den Senken von mehreren Größenordnungen (100:1). Nimmt man an, dass jede Senke zu jeder Zeit die Informationen mehrerer Quellen nutzt (z. B. 1:10), so ist trotzdem davon auszugehen, dass aufgrund des großen Übergewichts der Quellen die Mehrheit von ihnen ungenutzt bleibt (ca. 90%). Die Orientierung am Bedarf der Kontextsenken verspricht somit eine signifikante Reduktion des Kommunikationsaufwandes um bis zu zwei Größenordnungen. Nicht zuletzt enthalten die Nachrichten nur Beschreibungen von benötigten oder alternativen Providern. Dadurch wird die Übertragung von ungenutzten Beschreibungen stark reduziert, sodass die zugehö-

rigen Providernachrichten bei gleich bleibender Übertragungsrate in kürzerer Zeit übermittelt werden können.

Trotz des selteneren Nachrichtenaustauschs ist der Kommunikationsaufwand der reaktiven Abfrage hoch. Fragt ein Client benötigte Provider beim Server ab, dann führt der Server daraufhin Anfragen bei allen übrigen Clients durch, welche ihrerseits jeweils eine Antwort übermitteln. Anschließend überträgt der Server eine einzelne Nachricht mit den zusammengefassten Informationen. Jede reaktive Abfrage führt also zu $2n$ Nachrichten, während die proaktive Verteilung nur n Nachrichten benötigt. Die Komplexität beider Ansätze ist gleich – $\mathcal{O}(n)$.

Tabelle 5.5 stellt die Verzögerungen, Häufigkeiten und Volumina der beiden Ansätze gegenüber. Die Schätzwerte für die Anfragelatenz t_{REQ} wurden bereits erläutert. Stellen Consumer hohe zeitliche Anforderungen (/A 4/, S. 13), dann ist dieses Kriterium ausschlaggebend für den Einsatz der proaktiven Verteilung, weil $t_{REQ_{pro}}$ um ca. vier Größenordnungen kleiner ist, als $t_{REQ_{re}}$. Die Aktualisierungslatenz t_{UPD} ist in beiden Fällen ähnlich, wobei $t_{UPD_{re}}$ bereits in $t_{REQ_{re}}$ enthalten ist, und somit im Gegensatz zu $t_{UPD_{pro}}$ nicht zusätzlich zur Anfragelatenz anfällt. Die Häufigkeit der Anfragen von Consumern f_{REQ} ist in beiden Fällen identisch, weil die Providerverwaltung für die Consumer transparent erfolgt, und keine Auswirkung auf sie zur Folge haben darf. Für die Aktualisierungshäufigkeit f_{UPD} und das -volumen v_{UPD} der dabei kommunizierten Providernachrichten werden hier ohne Beschränkung der Allgemeinheit Verhältniswerte angenommen. Dabei wird wie zuvor davon ausgegangen, dass ein Consumer die Informationen von ca. 10 Providern nutzt. Aufgrund des Übergewichts von Quellen zu Senken wird trotzdem nur ein Zehntel der Quellen genutzt. Dies führt wegen der Bedarfsorientierung der reaktiven Abfrage dazu, dass deren Aktualisierungshäufigkeit $f_{UPD_{re}}$ nur ein Zehntel von $f_{UPD_{pro}}$ beträgt. Bei Gleichverteilung der Kontexttypen auf die Provider sind die Senken auch nur an einem Zehntel der Kontexttypen interessiert. Daraus folgt, wiederum wegen der Bedarfsorientierung der reaktiven Abfrage, eine Reduzierung des Volumens der Providernachrichten im selben Verhältnis – $v_{UPD_{pro}} \approx 10 v_{UPD_{re}}$.

Tabelle 5.5.: Qualitativer Vergleich der proaktiven Verteilung und der reaktiven Abfrage von Providerbeschreibungen

Eigenschaft	**proaktive Verteilung**	**reaktive Abfrage**
Anfragelatenz t_{REQ}	$\approx 10\mu s$	$\approx 200ms$
Aktualisierungslatenz t_{UPD}	$\approx 100ms$	$\approx 100ms$
Anfragehäufigkeit f_{REQ}	1	1
Aktualisierungshäufigkeit f_{UPD}	10	1
Aktualisierungsvolumen v_{UPD}	10	1

Aufgrund der signifikanten Reduzierung des Kommunikationsaufwandes (Gl. 5.12) ist die reaktive Abfrage zu bevorzugen. Zusätzlich kann die mangelnde Skalierbarkeit (/A 17/, S. 54) der proaktiven Verteilung wesentlich schneller zur Verhinderung der Diensterbringung führen, als das mit der reaktiven Abfrage der Fall ist.

$$f_{UPD_{pro}} \times v_{UPD_{pro}} \approx 100 \times f_{UPD_{re}} \times v_{UPD_{re}} \tag{5.12}$$

Kombination der Ansätze

Wünschenswert ist eine Kombination der Vorteile beider Ansätze. Dabei benachrichtigen die Clients den Server weiterhin proaktiv über gestartete und gestoppte Provider, sodass er den gespeicherten globalen Zustand aktualisieren kann, ohne ihn jedoch pauschal an alle Clients zu verteilen. Die Aktualisierungslatenz $t_{UPD_{hy}}$ ist dadurch halb so groß wie bei der reaktiven Abfrage. Weil der Server den Zustand nicht mehr pauschal verteilt, müssen die Clients zum Zugriff auf die Providerbeschreibungen aktiv Anfragen an den Server stellen. Dieser kann die Anfragen sofort mit den gespeicherten Providerbeschreibungen bedienen, d. h. der Anfrageaufwand sinkt auf 2 Nachrichten – $t_{REQ_{hy}} = t_{RTT} \approx 100ms$. Der Aktualisierungsaufwand ist halb so hoch, weil aufgrund der eliminierten pauschalen Weiterleitung Providernachrichten der Clients nicht mehr implizit durch Providernachrichten des Servers bestätigt werden. Es wird folglich nur noch eine Providernachricht zur Aktualisierung an den Server geschickt.

Sind nämlich die Beschreibungen auf dem Server gespeichert, so kann eine reaktive Abfrage eines Clients sofort durch eine Antwort des Servers bedient werden. Es sind nur noch 2 Nachrichten erforderlich. Benachrichtigen die Clients den Server weiterhin proaktiv, so kann er die gespeicherten Providerbeschreibungen aktualisieren, ohne sie jedoch weiterzuleiten. Für jede Aktualisierung ist pro Client nur eine Nachricht erforderlich – zur Benachrichtigung des Servers.

Tabelle 5.6.: Kommunikationsaufwand der proaktiven Verteilung, der reaktiven Abfrage und des hybriden Austausches von Providerbeschreibungen (in Nachrichten pro Zustandsänderung)

Eigenschaft	**proaktive Verteilung**	**reaktive Abfrage**	**hybrider Austausch**
Anfrageaufwand m_{REQ}	0	$2 \times n$	2
Aktualisierungsaufwand m_{UPD}	$n + 1$	0	1
Gesamtaufwand m_{Σ}	$(n + 1) \times r$	$2 \times n$	$2 + r$

Durch die Kombination der beiden Ansätze wurde der Kommunikationsaufwand nennenswert gesenkt (Tab. 5.6), wenngleich die Latenzen nach wie vor ähnliche

Größenordnungen wie bei der reaktiven Abfrage aufweisen. Die Vermeidung des Anfrageaufwandes im proaktiven Fall ist dadurch zu erklären, dass die erforderlichen Informationen bereits durch die Aktualisierung zum Client übertragen wurden, und somit eine lokale Selektion ohne Kommunikation statt finden kann. Ähnlich entsteht im reaktiven Fall kein Aktualisierungsaufwand, weil die Aktualisierung jeweils implizit als Teil der Anfrage erfolgt. Für die Beurteilung des Gesamtaufwandes (siehe auch Gleichung 5.13) ist zu berücksichtigen, dass sich Aktualisierungen wesentlich häufiger als Providerabfragen ereignen. Das Verhältnis zwischen diesen beiden Ereignissen wird durch r repräsentiert, und gibt an, wie viele Aktualisierungen pro Providerabfrage statt finden ($f_{UPD} = f_{REQ} \times r \mid r \gg 1$). Außerdem gilt für die Anzahl der Knoten/Geräte n: $n \gg r$.

$$\begin{aligned} m_{REQ} \times f_{REQ} + m_{UPD} \times f_{UPD} = \\ m_{REQ} \times f_{REQ} + m_{UPD} \times (r \times f_{REQ}) = \\ (m_{REQ} + m_{UPD} \times r) \times f_{REQ} = m_{\Sigma} \times f_{REQ} \\ m_{REQ} + m_{UPD} \times r = m_{\Sigma} \end{aligned} \tag{5.13}$$

Providernachrichten für hybriden Austausch

Weil die Clients den Server auch im hybriden Fall proaktiv benachrichtigen müssen, wird die UPDPRO-Nachricht wiederverwendet (Tab. 5.3 auf Seite 121). Will ein Client die geeigneten Provider für eine konkrete Kontextinformation abfragen, schickt er eine REQPRO-Nachricht an den Server, die mit einer RESPPRO-Nachricht beantwortet wird. Jedoch müssen die Informationen dieser Nachricht schnell genutzt werden, d. h. die Kontextinformationen der Provider müssen zeitnah abgefragt werden, weil die dynamische Verfügbarkeit der Provider jederzeit zu Fehlern beim Kontextzugriff führen kann. Aus diesem Grund sind die zuvor genannten Nachrichten nicht für die Bedienung von Abonnements geeignet. Statt dessen verwendet ein Client eine SUBPRO-Nachricht, um Benachrichtigungen über die Verwendbarkeit von geeigneten Providern für konkrete Kontextinformationen zu erhalten. Zur Benachrichtigung verwendet der Server erneut die UPDPRO-Nachrichten. Beendet eine Consumer sein Abonnement, müssen die zugehörigen Kontextinformationen nicht mehr geliefert und deren Provider nicht mehr überwacht werden. Dies signalisiert der Client durch eine UNSUBPRO-Nachricht an den Server. Tabelle 5.7 auf der nächsten Seite zeigt eine Übersicht der soeben beschriebenen Providernachrichten und deren Inhalte.

Mögliche Ergänzungen

Die anschließend vorgestellten Konzepte sind theoretisch umsetzbar, aber bisher nicht in den Kontextdienst integriert.

Tabelle 5.7.: Providernachrichten für hybriden Austausch

Typ	**Funktion**	**Inhalt**	**Behandlung**
REQPRO	einmalige Abfrage von verwendbaren Providern	Pattern	Ermittlung der zum Pattern passenden Provider und Übertragung mittels RESPPRO-Nachricht
RESPPRO	Beantwortung von Providerabfragen	Pattern, {(Host, {Provider})}	Integration der Provider in lokale Verwaltung, sowie Zugriff auf ihre Kontextinformationen
SUBPRO	Überwachung verwendbarer Provider	Pattern	Registrierung für Benachrichtigung bei Änderung verwendbarer Provider, die zum Pattern passen
UNSUBPRO	Beendigung der Überwachung von Providern	Pattern	De-Registrierung der Benachrichtigung/Überwachung

Weil die Anfragelatenz $t_{REQ_{hy}}$ signifikant größer ist, als bei der proaktiven Verteilung, kann sie bei zeitkritischen Anwendungen – z. B. adaptive Nutzerinteraktion – problematisch sein. Deshalb kann die Schnittstelle des Context Service derart erweitert werden, dass Consumern beim synchronen Kontextzugriff die Möglichkeit gegeben wird, ein Zeitlimit für die Bedienung den Abfrage zu spezifizieren. Verstreicht diese Zeit ohne dass ein geeigneter Provider selektiert und seine Kontextinformationen abgefragt wurden, so wird eine Fehlermeldung erzeugt. Diese sollte identisch mit den Fehlermeldungen sein, welche erzeugt werden, wenn keine geeigneten Provider verwendbar sind.

Dieser Mechanismus kann lokal auf jeder Instanz des Context Service realisiert werden. Eine Änderung oder Erweiterung der Nachrichten ist dafür nicht notwendig. Dies wäre ohnehin nur dann möglich, wenn eine global einheitliche Zeit auf allen Knoten des Context Service existieren würde. Dann könnte die Verarbeitung einer Nachricht abgebrochen werden, wenn der Server erkennt, dass deren Zeitlimit erreicht/überschritten ist. Um eine global einheitliche Zeit zu realisieren, wäre jedoch die Existenz und gleichmäßige Verteilung von hinreichend vielen Global Positioning System (GPS)-Empfängern vorauszusetzen, deren Zeitinformationen mit dem Protokoll IEEE1588 [ET03] an die Knoten verteilt werden könnten. Im Rahmen dieser Arbeit wird von dieser Annahme Abstand genommen.

Um sogar Consumer mit sehr hohen zeitlichen Anforderungen (z. B. $t_{REQ} \ll$

$100ms$) zu unterstützen, ist zusätzlich zum hybriden Austausch der Providerbeschreibungen eine hybride Form des Kontextzugriffs vorstellbar. Dabei registriert der Consumer rechtzeitig vor dem eigentlichen Zugriff – frühestens sobald er aktiv wird – seinen Bedarf an Kontextinformationen. Der lokale Context Service beginnt entsprechend mit der Selektion und Überwachung geeigneter Provider beim Server. Führt der Consumer später den synchronen Kontextzugriff tatsächlich durch, so verfügt der Context Service bereits lokal über die Beschreibungen geeigneter Provider und kann mit geringer Latenz auf ihre Kontextinformationen zugreifen – $t_{REQ_{hy}} \approx 10\mu s$. Dazu wäre eine Erweiterung der Schnittstellen des Context Service und der Consumer notwendig (siehe Kap. 6.2.2). Letzterer würde sich als `HybridConsumer` mittels der `prepare(HybridConsumer)`-Methode bei der Zugriffskomponente registrieren. Diese ermittelte mit Hilfe seiner `consumes()`-Methode die von ihm konsumierten Kontextinformationen und löst die Suche und anschließende Überwachung geeigneter Provider aus. Der eigentliche Kontextzugriff wäre – wie bei jedem synchronen Consumer – per `request(ContextPattern)` möglich. Der Context Service müsste regelmäßig die `stillAlive()`-Methode des `HybridConsumer` aufrufen, um zu ermitteln, ob dieser noch aktiv wäre und somit die Überwachung der Provider noch erforderlich ist.

5.6. Dezentrale Providerverwaltung

Bisher wurde eine C/S-basierte Verwaltung der Provider diskutiert. Der eigentliche Kontextzugriff dagegen fand durch Punkt-zu-Punkt-Verbindungen direkt zwischen den Clients statt. Betrachtet man nur den letzten Aspekt, so wird der Einsatz der P2P-Kommunikation erkennbar. Weil aber die zentrale Providerverwaltung die notwendige Vorstufe zum dezentralen Kontextzugriff darstellt, d. h. der Server nach wie vor das zentrale Element der Kommunkationsarchitektur ist, handelt es sich de facto um ein *zentralisiertes P2P-System* [SW05]. Der Context Service ähnelt in dieser Form den ersten P2P-Systemen (z. B. Napster) und teilt mit ihnen die Nachteile, welche aus dem Einsatz eines einzelnen zentralen Servers erwachsen. Der stellt einen Engpass dar und limitiert die Verfügbarkeit und Skalierbarkeit des Systems. Außerdem ist er eine zentrale Schwachstelle, welche die Robustheit des verteilten Context Service gegenüber Ausfällen und Verbindungsabbrüchen – speziell des Servers – gefährdet.

Unabhängig von der Wahl zwischen proaktiver Verteilung, reaktiver Abfrage oder hybridem Austausch von Providerbeschreibungen sind folglich Skalierbarkeit und Robustheit des zentralen Servers als problematisch anzusehen. Eine Verbesserung der Robustheit versprechen P2P-Netzwerke in denen potentiell jeder Knoten* die

*Peer (dt. Gleichberechtigter) – ein Knoten, welcher wechselweise oder parallel die Rolle von Client und Server einnehmen kann.

Rolle des Servers übernehmen kann. Setzt man ein so genanntes reines bzw. *pures P2P-System* ein, welches auf jegliche zentrale Komponente verzichtet, verschärft sich das Problem der Skalierbarkeit dahingehend, dass bei hybridem Austausch des globalen Zustandes die REQPRO-, SUBPRO- und UNSUBPRO-Nachrichten nicht an einen einzigen Server, sondern an alle übrigen $n-1$ Peers verteilt werden müssen. Der Anfrageaufwand hat bei dieser Flutung des Netzwerkes eine Komplexität von $\geq \mathcal{O}(n^2)$ [WGR05], weil jeder Peer auch entsprechende Antworten – RESPPRO- und PRO-Nachrichten – verschickt. Weil bereits der Kommunikationsaufwand der zentralisierten P2P-Lösung Anlass zur Verbesserung gab, ist dessen weitere Erhöhung unvertretbar. Es muss ein alternatives P2P-System gefunden werden, welches für den hybriden Austausch von Providerinformationen geeignet ist.

5.6.1. Hybrides Peer-to-Peer-Netzwerk

Beim Einsatz in dem beschriebenen Anwendungsszenario (Kap. 3) zeigen pure P2P-Systeme einen weiteren Nachteil – sie unterstellen die Gleichartigkeit aller Knoten. Handelt es sich bei ihnen tatsächlich um Geräte mit ähnlichen Ressourcen und Voraussetzungen, so ist diese Gleichbehandlung unproblematisch. Im vorliegenden Szenario besteht jedoch eine große Heterogenität in der Art der Geräte und ihrer Nutzung bzw. Verfügbarkeit – Dauerbetrieb vs. kurzzeitiger, bedarfsabhängiger Betrieb. Ignoriert man diese Tatsache, kann es zur Beeinträchtigung des Systems führen [Dar05], weil die leistungsschwachen, kurzzeitig verfügbaren Geräte nicht in der Lage sind, ihren Anteil zur Aufrechterhaltung des Systems beizutragen. Die übrigen Geräte müssen aufgrund der Fluktuation der unzuverlässigen Geräte einen erhöhten Wartungsaufwand betreiben, um die Konsistenz des Netzwerkes und des globalen Zustandes wiederherzustellen. So genannte hybride P2P-Systeme stellen viel versprechende Kombinationen unterschiedlicher P2P-Ansätze dar. Einzelne Vertreter dieser Systeme berücksichtigen die Heterogenität der beteiligten Knoten, oder tragen der physischen Topologie Rechnung, um z. B. eine Reduzierung des Kommunikationsaufwandes zu erreichen.

Aufgrund der Vielfalt existierender Systeme und ihrer unterschiedlichen Einsatzgebiete wird nachfolgend auf eine Betrachtung einzelner Ansätze verzichtet. Statt dessen werden die Kriterien für die Auswahl eines solchen Systems diskutiert und schrittweise kombiniert, sodass ein geeignetes hybrides P2P-System entsteht.

Ein ideales hybrides P2P-System für den Context Service sollte die heterogene *Leistungsfähigkeit* und *Verfügbarkeit* der Peers berücksichtigen. Leistungsschwache, kurzzeitig verfügbare Peers sollten weitgehend vom Betrieb des Systems befreit werden, und statt dessen vorwiegend als Clients betrieben werden. Das bedeutet, dass die Knoten des Netzes in zwei Klassen unterteilt werden. Diejenigen Peers, welche die Rolle eines Clients einnehmen werden nachfolgend *Edge Peers* genannt, wogegen ein *Super Peer* die Rolle eines Servers übernimmt.

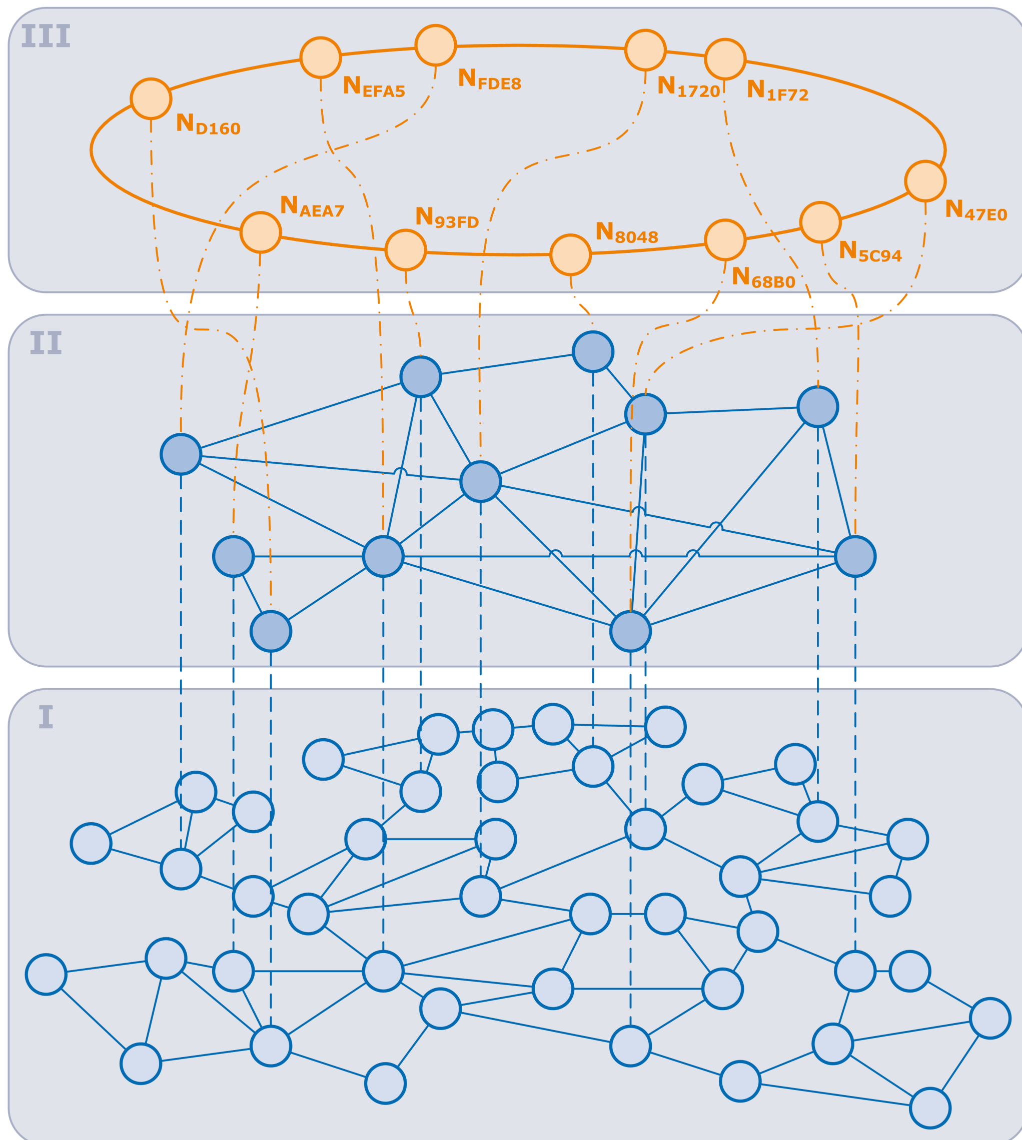

Abbildung 5.15.: Korrelation der Topologien zwischen (I) physischem, (II) Super-Peer- und (III) Überlagerungsnetzwerk (angelehnt an [WGR05])

Abbildung 5.15 auf der vorherigen Seite zeigt den möglichen Aufbau eines solchen hybriden P2P-Netzwerkes. Schicht I zeigt die Topologie des physischen Netzwerkes über das alle Peers miteinander kommunizieren. In dieser Darstellung wird kein optischer Unterschied zwischen drahtlosen und -gebundenen Verbindungen, sowie leistungsstarken und -schwachen Peers gemacht. Schicht II zeigt das physische Netzwerk der Super Peers, welche den Edge Peers den Zugang zum Netzwerk zur Verfügung stellen und das Netzwerk aufrecht erhalten. Für die Organisation der Super Peers wird ein Überlagerungsnetzwerk – Schicht III – auf Basis eines strukturierten* P2P-Netzwerkes eingesetzt.

Es wird eine strikte Trennung der Rollen von Edge Peers und Super Peers dahingehend vorgenommen, dass erstere Clients und letztere Server des verteilten Kontextdienstes sind. Ein Peer kann und muss ggf. seine Rolle wechseln. Jedem Super Peer aus Schicht II ist eine Teilmenge der Edge Peers aus Schicht I zugeordnet, welche nach dem Betritt zum Netz ausschließlich mit diesem Super Peer kommuniziert. Neben der Kommunikation mit seinen Edge Peers tauscht jeder Super Peer auch noch Informationen mit den übrigen Super Peers aus. Für das hybride P2P-Netzwerk werden folglich zwei Protokolle benötigt. Das *Edge-Peer-Protokoll* $P2P_{Edge}$ wird zwischen Schicht I und II eingesetzt, während das *Super-Peer-Protokoll* $P2P_{Super}$ innerhalb von Schicht II eingesetzt wird, und dabei die Struktur von Schicht III realisiert.

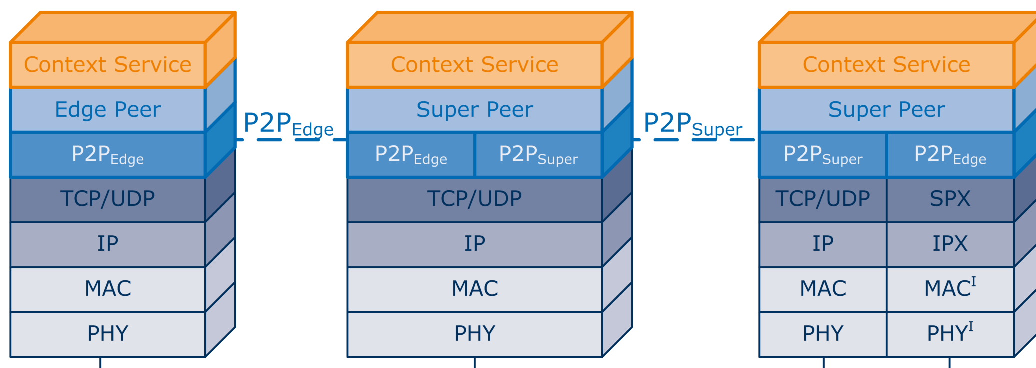

Abbildung 5.16.: Protokollstapel eines Edge Peers und zweier Super Peers (angelehnt an [Zim80])

Bezogen auf das ISO/OSI-Referenzmodell [Zim80] bilden die beiden Protokolle zusammen mit dem Context Service die anwendungsnahen Schichten V bis VII. Jeder Super Peer stellt ein *Application Level Gateway* zwischen seinem Teilnetz der Edge Peers und dem Netz der Super Peers dar. Aufgrund der dazu notwendigen

*Die Ringstruktur ist nur exemplarisch verwendet, weil sie eine einfachere Darstellung erlaubt.

Protokollumsetzung muss er beide Protokolle unterstützen und entsprechend zwei Protokollstapel besitzen (Abb. 5.16 auf der vorherigen Seite). Die unteren Schichten I bis IV sind an dieser Stelle von nachrangiger Bedeutung. Es sei jedoch bemerkt, dass bei einem Super Peer die Protokolle $P2P_{Edge}$ und $P2P_{Super}$ nicht notwendigerweise gemeinsam die gleiche Implementierung der Schicht IV nutzen müssen (Abb. 5.16, Mitte), sondern auch unterschiedliche Protokollstapel vorhanden sein können (Abb. 5.16, rechts).

5.6.2. Netzwerkverwaltung

Bevor Providerbeschreibungen oder Kontextinformationen ausgetauscht werden können, ist der Aufbau der Netzwerkschichten entsprechend Abbildung 5.15 auf Seite 131 sicher zu stellen, d. h. es ist eine Steuerung erforderlich, welche die Rolle – Edge Peer oder Super Peer – eines Peers kontrolliert. Sie muss dafür sorgen, dass in jedem Teilnetz ein Super Peer existiert, indem einer der Peers bedarfsabhängig die Rolle des Super Peers übernimmt. Damit diese Umschaltung in jeder Situation möglich ist, muss jede Context Service Instanz über eine Implementierung beider Protokolle – $P2P_{Edge}$ und $P2P_{Super}$ – verfügen.

Abbildung 5.17 auf der nächsten Seite zeigt die möglichen Zustände und Zustandsübergänge im Lebenszyklus eines Peers. Kurz nach seiner Inbetriebnahme befindet sich jeder Peer in einem *Initialisierungszustand* in dem er nur eingeschränkt benutzbar ist, d. h. der Context Service bedient nur lokale Consumer mit lokalen Providern. In diesem Zustand beginnt der Peer mit der Suche nach einem Super Peer in seinem lokalen Netz. Ist diese Suche erfolgreich, so meldet er sich bei dem Super Peer an, und geht anschließend in den *Edge-Peer-Betrieb* über, in welchem der Context Service und $P2P_{Edge}$ aktiv ist. Wird der Anmeldeversuch vom Super Peer abgelehnt, wird die Suche fortgesetzt. Nach Ablauf einer Suchdauer, geht der Peer in den *Super-Peer-Betrieb* über, weil davon auszugehen ist, dass in seinem Teilnetz noch keine anderen Peers – speziell Super Peers – vorhanden sind, oder vorhandene Super Peers bereits ausgelastet sind. Im Super-Peer-Betrieb wird die Suche nach Super Peers auf globaler Ebene fortgesetzt, um das Netz der Super Peers aufzubauen. Wurde ein Super Peer gefunden, sowie die Kopplung mit diesem erfolgreich durchgeführt, kann $P2P_{Super}$ aktiviert werden. Durch den Kopplungsmechanismus ist ein induktiver Aufbau des Super-Peer-Netzwerkes gewährleistet. Das Super-Peer-Protokoll wird erst deaktiviert, wenn der Peer keinen Kontakt mehr zu einem anderen Super Peer hat. Die Initialisierung beginnt erneut.

Zur Suche der Super Peers im lokalen und globalen Maßstab kann auf eine Vielzahl – z. T. erprobter – Ansätze zurückgegriffen werden, welche für die Suche von Knoten existieren. Das Protokoll zur Suche des Super Peers muss auch die Heterogenität der Peers berücksichtigen. Wird ein leistungsschwacher oder mobiler Peer zum Super Peer, weil er keinen Super Peer gefunden hat, so muss beim Beitritt

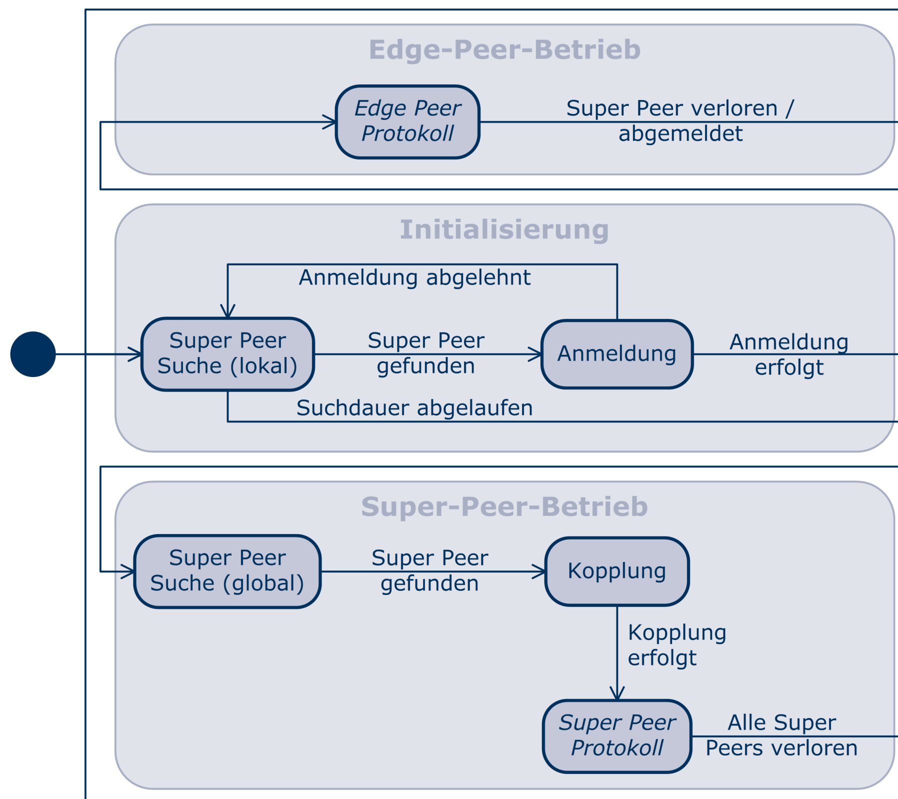

Abbildung 5.17.: Übergänge zwischen Edge- und Super-Peer-Betrieb, inkl. Aktivierung und Deaktivierung des Edge- bzw. Super-Peer-Protokolls

nachfolgender Peers zum Netzwerk ggf. die Rolle des Super Peers zu einem dieser leistungsfähigeren und stationären Peers wechseln. In lokalen Netzwerken mit besonders vielen Edge Peers ist der Einsatz mehrerer Super Peers wünschenswert, um die Skalierbarkeit und Verfügbarkeit des Netzwerkes – und somit des Kontextdienstes – zu gewährleisten. Für diese Probleme sind einerseits Aushandlungsprotokolle und andererseits Lastverteilungprotokolle entwickelt worden, welche, ebenso wie die Suchverfahren, im Rahmen dieser Arbeit nicht näher untersucht werden.

5.6.3. Edge-Peer-Betrieb

Weil die Rollenverteilung zwischen den Edge Peers eines Teilnetzes und ihrem Super Peer derjenigen zwischen Clients und Server aus der zentralen Verwaltung (Kap. 5.5) entspricht, können die bereits definierten Mechanismen mit ihren Nachrichten weitgehend wiederverwendet werden (Kap. 5.5.4 und 5.5.5). Die Providernachrichten für

den hybriden Austausch von Beschreibungen und die Zugriffsnachrichten können unverändert eingesetzt werden. Lediglich das Protokoll zum Austausch der Erreichbarkeitsnachrichten ist ggf. an das veränderte Protokoll zur lokalen Suche von Super Peers anzupassen.

5.6.4. Super-Peer-Betrieb

Als Super Peers sollen möglichst leistungstarke, stationäre Knoten ausgewählt werden, die mehr Arbeit beim Betrieb des Netzwerkes übernehmen können und zuverlässiger sind. Weil ein Super Peer für die Edge Peers seines Teilnetzes als Server fungiert, speichert er alle Beschreibungen ihrer Provider. Setzt man den hybriden Austausch dieser Beschreibungen ebenfalls zwischen den Super Peers ein, so hat dies eine Vervielfachung des Aktualisierungsaufwandes zur Folge, weil alle UPDPRO-Nachrichten der Edge Peers von ihrem jeweiligen Super Peer an die übrigen $s - 1$ Super Peers verteilt werden müssen. Somit reduziert sich zwar der Aktualisierungsaufwand im Vergleich zur proaktiven Verteilung um den Quotienten e^* weil nur noch s Super Peers aktualisiert werden müssen. Jedoch wächst der Aktualisierungsaufwand im Vergleich zum hybriden Austausch um den Faktor s der zu benachrichtigenden Super Peers. Folglich ist die Verteilung der Beschreibungen auf alle Super Peers keine adäquate Lösung.

Um die Beschreibungen geeigneter Provider im Bedarfsfall finden zu können, müssen zuerst diejenigen Super Peers gefunden werden, welche die Edge Peers der Provider verwalten. Weil jedoch der Super Peer der eine Anfrage eines Edge Peers bedient, a priori nicht weiß, welche weiteren Super Peers die gesuchten Providerbeschreibungen gespeichert haben, scheint das Durchsuchen der Menge aller $s - 1$ Super Peers unvermeidbar. Somit liegt der Anfrageaufwand mit rund s Providernachrichten wesentlich höher als der des hybriden Austauschs.

Anstatt ein unstrukturiertes Netzwerk von Super Peers zu betreiben, welches ineffizient aktualisiert und durchsucht werden muss, bietet sich der Einsatz eines strukturierten P2P-Überlagerungsnetzwerkes an in dem eine deterministische Suche möglich ist. Derartige Systeme sind unter den Namen *DHT*, *Decentralized Object Location and Routing (DOLR)* oder *Group Anycast and Multicast (CAST)* bekannt [DZD+03]. Ihnen allen ist gemein, dass die Vermittlung der ausgetauschten Nachrichten anhand spezieller Schlüssel erfolgt. Bei diesem Ansatz – *Key-Based Routing (KBR)* genannt – wird jeder Nachricht ein Schlüssel aus einem klar begrenzten numerischen Schlüsselraum zugeordnet [WGR05]. Jeder Peer erhält ebenfalls einen Schlüssel und einen Teil des Schlüsselraums, welchen er fortan verwaltet. Der Schlüsselraum ist meist ein eindimensionaler, algebraischer Ring (Chord [SMK+01], Pastry [RD01]), seltener ein n-dimensionaler Torus (Content-Addressable Network (CAN) [RFH+01]). Zur

$^*n = |\mathbb{N}|$ – Anzahl aller Peers bzw. Knoten $\mathbb{N}$; $s = |\mathbb{S}|$ – Anzahl der Super Peers $\mathbb{S}$; $e = n/s$ – durchschnittliche Anzahl der Edge Peers pro Super Peer

Vermittlung der Nachrichten verfügt jeder Peer über Routingtabellen, welche den Schlüsselraum effizient strukturieren – meist exponentiell. Die einzelnen Teile dieser Tabellen enthalten Einträge mit bekannten Peers in den zugehörigen Unterräumen des Schlüsselraumes. Mit dieser Strukturierung erreichen viele Vertreter der strukturierten P2P-Systeme einen Kommunikationsaufwand der Komplexität $\mathcal{O}(log\ n)$ für die Übertragung einer Nachricht zu einem konkreten Peer dessen Adresse dem sendenden Peer a priori nicht bekannt ist. Für die erfolgreiche Vermittlung von Nachrichten muss jeder Peer nur einen Bruchteil der übrigen Peers kennen – seine Routingtabellen haben ebenfalls eine Komplexität von $\mathcal{O}(log\ n)$ Einträgen.

Providersuche und Ereignisbehandlung

Für die Verwaltung der Provider wird eine dezentrale, verteilte Verwaltungsstruktur geschaffen – der *Distributed Provider Index (DPI)*. Ähnlich einer DHT bietet er lokalen Anwendungen – hier: der Instanz des Context Service – eine Menge von Dienstprimitiven an, welche auf Basis von KBR realisiert werden. Dieses Konzept geht auf einen Vorschlag zur Generalisierung strukturierter P2P-Netzwerke zurück [DZD+03]. Bevor jedoch die Dienstprimitive des DPI ausführlicher vorgestellt werden, soll in diesem Abschnitt auf die Anwendungsfälle eingegangen werden, welche der DPI unterstützen muss. Besondere Aufmerksamkeit wird auf die Zusammenhänge der Informationen des DPI gelegt.

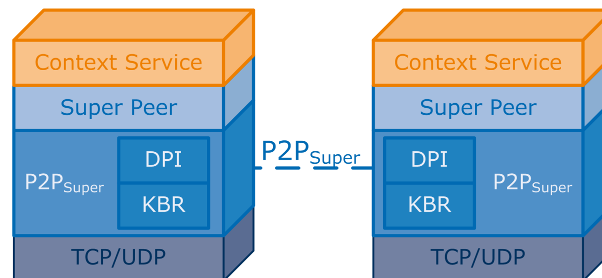

Abbildung 5.18.: Realisierung des Super-Peer-Protokolls durch den DPI auf Basis von KBR

Stellt ein Edge Peer eine Suchanfrage mittels einer REQPRO- oder SUBPRO-Nachricht, so greift sein Super Peer auf die lokal gespeicherten Providerbeschreibungen der Provider seiner Edge Peers zu. Zusätzlich sucht er mit den Primitiven `reqProviders` und `subProviders` auch im DPI nach geeigneten Providern. Diese Suche wird anhand der benötigten Kontextinformationen des Consumers, d. h. anhand der Consumed Patterns, durchgeführt. Folglich muss der Super Peer lokal und der DPI auf verteilter Ebene eine Abbildung der Menge von Context Pattern auf

die Menge von Providern vornehmen können (Gl. 5.14).

$$f_{CP-PRO} : \mathbb{CP} \mapsto \mathbb{PRO} \tag{5.14}$$

Neben `reqProviders` und `subProviders` dienen die Primitive `unsubProviders` und `notifyProviders` zur Steuerung des Zugriffs auf die Informationen des DPI.

Weil der DPI den globalen Zustand widerspiegelt, müssen jegliche Ereignisse die zur Veränderung der Verfügbarkeit von Providern führen an den DPI signalisiert werden. Wenn (a) ein Provider gestartet wird, (b) ein Edge Peer oder (c) sogar ein Super Peer dem Netzwerk beitritt, so müssen die Provider und Peers im DPI registriert werden. Analog müssen Provider und Peers aus dem DPI entfernt werden, wenn (d) ein Provider gestoppt wird, (e) ein Edge Peer oder (f) ein Super Peer ausfällt. Ist vorhersehbar oder geplant, dass ein Provider gestoppt wird oder ein Peer das Netzwerk verlässt, dann kann der betroffene Provider oder Peer selbst die Informationen für die Aktualisierung des DPI liefern. Fällt ein Provider oder Peer jedoch unvorhergesehen aus, so müssen die verbliebenen Peers über Informationen verfügen, welche zu einer entsprechenden Aktualisierung des DPI genutzt werden können. Weil die Behandlung unvorhersehbarer Ausfälle auch auf vorhersehbare Ereignisse anwendbar ist, werden letztere nicht näher betrachtet.

Signalisiert ein Edge Peer seinem Super Peer einen neuen Provider, so registriert dieser ihn im DPI mit dem Primitiv `addProvider()`. Analog wird der Beitritt eines Edge Peers mit `addEdgePeer()`, sowie eines Super Peers mit `addSuperPeer()` im DPI vermerkt. Während dieser Operationen müssen die Informationen im DPI abgelegt werden, welche einerseits die synchrone und asynchrone Suche von Providern erlauben, und andererseits das Entfernen von Providern und Peers gestatten, falls diese nicht mehr verfügbar sind. Die bestehenden Abhängigkeiten und daraus entstehende Abbildungen werden bei der Betrachtung der weiteren Ereignisse aufgezeigt. Sie sind die Grundlage für die Funktionsparameter und -ergebnisse des DPI-API, welches im folgenden Kapitel vorgestellt wird.

Als Reaktion auf den Ausfall eines Providers muss ihn sein Edge Peer aus seinen lokalen Verwaltungsstrukturen entfernen und den Ausfall an seinen Super Peer signalisieren. Dieser muss ebenfalls seine lokalen Verwaltungsstrukturen bereinigen und die Entfernung des Providers aus dem DPI veranlassen. Weil Provider von den Edge Peers, den Super Peers und dem DPI anhand ihrer Provided Pattern verwaltet werden, sind – zusätzlich zum Bezeichner des betroffenen Providers – weitere Informationen erforderlich. Mit ihnen muss die Feststellung der Provided Pattern des Providers möglich sein, anhand welcher der Provider registriert ist. Im Rahmen der Analyse infolge des Starts eines Providers speichert deshalb jeder Edge Peer die vollständige Beschreibung des Providers, sodass nachfolgend alle Informationen – inkl. der Provided Pattern – über den Provider auch ohne dessen Zutun dem Edge Peer zur Verfügung stehen. Dementsprechend verfügt jeder Edge Peer über die notwendigen Informationen um den Ausfall eines Providers zu behandeln. Er nimmt

dabei in Umkehrung von Gleichung 5.14 die Abbildung der Menge der Provider auf die Menge der Context Pattern vor (Gl. 5.15).

$$f_{PRO-CP} : \mathbb{PRO} \mapsto \mathbb{CP} \tag{5.15}$$

Die UPDPRO-Nachricht zur Signalisierung an den Super Peer enthält nur den Bezeichner des Providers, weil der Super Peer bereits – bei der Signalisierung des Starts des Providers – die vollständige Beschreibung des Providers erhalten hat, sodass der Super Peer die Abbildung (Gl. 5.15) selbst durchführen kann. Um den Provider aus dem DPI zu entfernen, nutzt der Super Peer dessen Primitiv `remProvider()`, welche die Assoziationen der einzelnen Provided Pattern $P \in \mathbb{P}_i$ zu ihrem Provider PRO_i aufhebt.

Fällt ein Edge Peer E_i aus, so muss dessen Super Peer seine lokalen Verwaltungsstrukturen und den DPI aktualisieren. Aufgrund der Verwaltung von Providern anhand ihrer Provided Pattern muss der Super Peer die Menge der Edge Peers $\mathbb{E}$ auf die Menge der Context Pattern $\mathbb{CP}$ abbilden. Diese Abbildung erfolgt zweistufig. Zuerst stellt der Super Peer die Menge der Provider $\mathbb{PRO}_{E_i}$ des ausgefallenen Edge Peers E_i fest, um anschließend mit Gl. 5.15 alle Provided Pattern $\mathbb{P}_{E_i}$ dieser Provider zu ermitteln.

$$\begin{aligned} f_{E-PRO} &: \mathbb{E} \mapsto \mathbb{PRO} \\ f_{E-CP} &: \mathbb{E} \mapsto \mathbb{PRO} \mapsto \mathbb{CP} \\ f_{E-CP} &= f_{E-PRO} \circ f_{PRO-CP} \end{aligned} \tag{5.16}$$

Mit diesen Informationen kann der Super Peer das Primitiv `remEdgePeer()` des DPI nutzen.

Der Ausfall eines Super Peers S_i wird von den übrigen Super Peers im Rahmen der Verwaltung des Super Peer Netzwerkes erkannt. Wie zuvor bei der Behandlung des Ausfalls eines Providers und eines Edge Peers ist eine Abbildung auf die Provided Pattern im DPI notwendig. Um diese Abbildung analog zur zweistufigen Abbildung f_{E-PRO} durchführen zu können, müssen die Informationen, welche für f_{E-PRO} und f_{PRO-CP} auf S_i gespeichert waren im DPI verfügbar sein. Weil jeder Super Peer Teil des DPI ist, ist es hinsichtlich des Gesamtdatenvolumens des DPI unerheblich, ob ein Super Peer die Informationen für f_{E-PRO} und f_{PRO-CP} lokal oder auf einem anderen Super Peer speichert. Wird dieser Ansatz verfolgt, so gehen beim Ausfall eines Super Peers mit hoher Wahrscheinlichkeit nicht die Informationen über dessen Edge Peers und deren Provider verloren, sondern die Informationen über die Edge Peers und Provider anderer Super Peers, welche noch Teil des intakten DPI sind. In vielen DHTs wird die dynamische Verfügbarkeit der Peers dadurch behandelt, dass jeder Peer in regelmäßigen Abständen die von ihm in der DHT abgelegten Inhalte erneut darin speichert. Auch auf den DPI ist dieses Vorgehen anwendbar.

Damit die verbliebenen Super Peers die Verwaltungsinformationen von S_i aus dem DPI entfernen können, müssen in ihm zusätzliche Informationen abgelegt sein, welche die Abbildung f_{S-E} gestatten. Sind diese Informationen vorhanden, steht einer dreistufigen Ermittlung der Edge Peers, Provider und Provided Pattern von S_i nichts im Wege (Gl. 5.17).

$$\begin{aligned} f_{S-E} &: \mathbb{S} \mapsto \mathbb{E} \\ f_{S-CP} &: \mathbb{S} \mapsto \mathbb{E} \mapsto \mathbb{CP} \\ f_{S-CP} &= f_{S-E} \circ f_{E-CP} \\ &= f_{S-E} \circ f_{E-PRO} \circ f_{PRO-CP} \end{aligned} \tag{5.17}$$

Zuerst stellt der DPI die Menge der Edge Peers die Menge $\mathbb{E}_{S_i}$ des ausgefallenen Super Peers S_i fest, dann die Provider $\mathbb{PRO}_{S_i}$ der Edge Peers $\mathbb{E}_{S_i}$ fest, um anschließend mit Gl. 5.15 alle Provided Pattern $\mathbb{P}_{S_i}$ dieser Provider zu ermitteln. Das Primitiv `remSuperPeer()` benötigt diese Informationen zur Beseitigung der Provider von S_i. Die Bereinigung des DPI wird durch einen der benachbarten Super Peers ausgelöst die S_i im Rahmen der Netzwerkverwaltung überwachen. Zur Koordination der beobachtenden Super Peers kann ein Protokoll verwendet werden, wie es zum selben Zweck in DHT-Implementierungen zum Einsatz kommt (z. B. bei [SMK+01]).

DPI-Primitive und Nachrichtenaustausch

Der DPI ist der Dienst des Super-Peer-Protokolls. Nach der Betrachtung der Anwendungsfälle des DPI sollen nun dessen Primitive, inklusive ihrer Parameter bzw. Ergebnisse und der Verwendung vorgestellt werden. Weil der DPI unter Verwendung von KBR definiert ist, sollen zuerst die genutzten Funktionen des API dieses Dienstes erläutert werden [DZD+03].

KBR-Datentypen Um KBR zu verwenden, sind nur wenige Datentypen notwendig. Dazu gehört der Schlüssel – `Key`. Er besteht aus einer Zeichenkette von 160 bit Länge. Ein `NodeHandle` repräsentiert einen Peer und besteht aus dessen Transportadresse* und seinem Peer-Bezeichner der vom Typ `Key` ist. Eine Nachricht – `Msg` – kann eine beliebige Menge von Daten enthalten.

KBR-Funktionen Um den DPI zu realisieren, werden folgende Funktionen des KBR verwendet:

- `void route(Key k, Msg m, NodeHandle hint)` kann vom DPI aufgerufen werden, um eine Nachricht `m` an denjenigen Peer zu übertragen, der für den

*Eine Adresse des Netzwerkes, welches dem Überlagerungsnetz zugrunde liegt – z. B. IP-Adresse und Port.

Schlüssel `k` zuständig ist, d. h. `k` liegt in dem Teil des Schlüsselraums den dieser Peer verwaltet. Der Parameter `hint` ist optional, und wird verwendet, wenn die Adresse des empfangende Peers bereits bekannt ist.

- `void deliver(Key k, Msg m)` wird vom KBR auf dem DPI aufgerufen, um die Nachricht `m` an ihn auszuliefern.

Mit diesen Funktionen und ihren Parametern kann die Kommunikation zwischen den Super Peers realisiert werden.

DPI-Datentypen Bei der Verwendung des DPI-API werden folgende Datentypen benötigt:

- `ContextPattern` repräsentiert ein einzelnes Context Pattern – provided oder consumed
- `ProviderId` identifiziert einen Provider und besteht aus
  - einem Namen und
  - einer optionalen `HostId` des Edge Peers
- `HostId` bezeichnet einen Edge oder Super Peer und besteht aus:
  - der Transportadresse des Peers und
  - dem optionalen Schlüssel bzw. Key, falls es sich um eine Super Peer handelt
- `ProviderInfo` enthält die vollständige Beschreibung eines Providers mit:
  - seinen Provided und ggf. Consumed `ContextPattern`, sowie
  - seinem `ProviderId`
- `EdgePeerInfo` repräsentiert einen Edge Peer mit:
  - seiner `HostId` und
  - den `ProviderInfo` seiner Provider

Ausdrücke in geschweiften Klammern stellen beliebig mächtige Mengen von Informationen desselben Typs dar. Mit runden Klammern und Kommata werden n-Tupel repräsentiert. Das API bietet folgende Funktionen, welche von der lokalen Instanz des Context Service zum Zugriff auf die Informationen des DPI genutzt werden können.

Die folgenden vier Funktionen `reqProviders`, `subProviders`, `notifyProviders` und `unsubProviders` werden von den Kontextdienstinstanzen der Super Peers beim

synchronen und asynchronen Zugriff auf Providerbeschreibungen aufgerufen. Sie alle nutzen die Abbildung f_{CP-PRO} (Gl. 5.14). Alle weiteren Funktionen dienen der Aktualisierung des globalen Zustandes, der die Providerbeschreibungen enthält und werden von dem Peer aufgerufen, welcher die Änderung feststellt die die Aktualisierung erforderlich macht (z. B. Provider ausgefallen).

`{ProviderInfo} reqProviders(ContextPattern cp)` *Suche nach Providern die konkrete Kontextinformationen liefern können* : Die Anfrage wird zu dem Super Peer weitergeleitet, der die Provider verwaltet, die die per ContextPattern spezifizierten Kontextinformationen liefern können. Der Super Peer antwortet mit den vollständigen Beschreibungen aller geeigneten Provider. Sie werden an den Context Service ausgeliefert.

Die Nachricht zur Übertragung der Anfrage besteht aus dem Schlüsselwort „REQPRO“, `cp` das den benötigten Kontext beschreibt und dem `NodeHandle S` des sendenden Super Peers. Außerdem wird ein Schlüssel `k` bestimmt, indem die Hash-Funktion SHA-1 [oSN02] auf `cp` angewendet wird. Die Nachricht wird durch den KBR-Aufruf `route(k, (REQPRO, cp, S), NULL)` an den Super Peer übertragen, in dessen Teil des Schlüsselraums `k` liegt. Für die Antwort erzeugt dieser eine Nachricht mit dem Schlüsselwort „RESPPRO“ und den `ProviderInfo pro` aller geeigneten Provider. Diese Nachricht wird mit `route(NULL, (RESPPRO, {pro}), S)` zurück an `S` übertragen.

`{ProviderInfo} subProviders(ContextPattern cp)` *Startet Abonnement von Providern die konkrete Kontextinformationen liefern können* : Die Anfrage wird wie bei `reqProviders()` vermittelt, jedoch vom empfangenden Super Peer abweichend verarbeitet. Dieser registriert den anfragenden Super Peer und das ContextPattern zur Überwachung von Providern, sodass nachfolgende Änderungen der verwalteten Provider, welche das ContextPattern bedienen können, an den anfragenden Super Peer weiter geleitet werden können. Abschließend wird ein initialer Aufruf von `notifyProviders` durchgeführt (siehe nächste Funktion).

Die Nachricht zum Start eines Abonnements ähnelt der von `reqProviders`, und wird mittels `route(k, (SUBPRO, cp), NULL)` an den zuständigen Super Peer übertragen.

`void notifyProviders({ProviderInfo pro}, {ProviderId pid})` *Signalisiert die Verfügbarkeit von Providern, welche konkrete Kontextinformationen liefern können* : Eine Nachricht mit Beschreibungen der neu verfügbaren Provider, sowie den Bezeichnern der nicht mehr verfügbaren Provider wird an den Super Peer gesendet der ein Abonnement gestartet hat. Dort werden die Informationen an den Context Service weitergeleitet.

Im Gegensatz zu `reqProviders` enthalten die Nachrichten die als Reaktion auf `subProviders` versendet werden nicht nur die `pro` der verfügbaren, sondern auch die `pid` der nicht mehr verfügbaren Provider. Außerdem enthalten sie noch den `NodeHandle R` des benachrichtigenden Super Peers. Diese Informationen werden per `route(NULL, (UPDPRO, {pro}, {pid}, R), S)` an den Super Peer gesendet der die Benachrichtigungen abonniert hat.

`void unsubProviders(ContextPattern cp)` *Ende des Abonnements von Providern, welche konkrete Kontextinformationen liefern können* : Nach der Weiterleitung der Anfrage zu dem Super Peer, welcher die Provider des ContextPattern verwaltet, entfernt dieser die Registrierungsinformationen des Abonnements und stoppt damit die Benachrichtigungen über neue oder ausgefallene Provider.

Die Nachricht zum Beenden eines Abonnements ähnelt der zum Start desselben – `route(NULL, (UNSUBPRO, cp), R)`. Neben dem abweichend definierten Schlüsselwort wird statt des Schlüssels der `NodeHandle R` verwendet. Weil es sich dabei um dasselbe `R` handelt, welches bei `notifyProviders` stets übermittelt wird, ist das Ziel der Übertragung bereits bekannt.

`void addProvider(ProviderInfo pro)` *Registrierung eines Providers* : Bei der Registrierung eines neuen Providers im DPI müssen die Informationen abgelegt werden, welche das Auffinden des Providers im Rahmen von `reqProviders` und `subProviders` erlauben (Gl. 5.14). Sie müssen weiterhin das Entfernen des Providers bei dessen Ausfall gestatten (Gl. 5.15). Deshalb wird aus `pro` einerseits die Menge der Provided Pattern $\mathbb{P}$, und andererseits die `HostId hid` isoliert. Diese Informationen werden benutzt, um den Provider unter jedem seiner Provided Pattern und seinem Edge Peer zu registrieren, sowie seine Providerbeschreibung abzuspeichern.

Um die Zugehörigkeit des Providers zu seinem Edge Peer im DPI zu speichern, wird mittels der Hash-Funktion über `hid` der Schlüssel `k` ermittelt. Dieser ermöglicht den Aufruf `route(k, (ADDPRO2EDGE, pro), NULL)`. Der empfangende Super Peer extrahiert `hid` aus `pro`, und speichert `pro` in der Menge der Provider des Edge Peers `hid` ab.

Für jedes Context Pattern `cp` aus $\mathbb{P}$ wird ebenfalls per Hash-Funktion ein Schlüssel `k` bestimmt, und der Aufruf `route(k, (ADDPRO, cp, pro), NULL)` durchgeführt. Der zuständige Super Peer speichert `pro` unter `cp` ab, sodass der Provider im Rahmen von `reqProviders`, `subProviders` und `unsubProviders` auffindbar ist.

Zur Speicherung der Providerbeschreibung wird ein Schlüssel `k` durch Anwendung der Hash-Funktion auf `pid` ermittelt, und anschließend `route(k, (ADDPROINFO, pro), NULL)` aufgerufen. Der empfangende Super Peer speichert `pro` unter `pid` ab.

`void remProvider(ProviderId pid)` *De-Registrierung eines ausgefallenen Providers* : Fällt ein Provider aus, so müssen dessen Spuren aus dem DPI beseitigt werden. Dazu wird die Providerbeschreibung `pro` ermittelt – und gleichzeitig aus dem DPI gelöscht. Mit ihr können die übrigen Informationen gelöscht werden, d. h. die Registrierungen des Providers unter seinen Provided Pattern (Gl. 5.15) und seinem Edge Peer.

Zuerst wird aus `pid` mit Hilfe der Hash-Funktion der Schlüssel `k` bestimmt, um `route(k, (REMPROINFO, pid, S), NULL)` aufrufen zu können. Der empfangende Super Peer löst die zu `pid` gehörige Providerbeschreibung, sendet sie jedoch an den aufrufenden Super Peer `S` mittels `route(NULL, (PROINFO, pro), S)`.

Danach wird für jedes Provided Pattern `cp` des ausgefallenen Providers der Schlüssel `k` bestimmt, und `route(k, (REMPRO, cp, pro), NULL)` aufgerufen werden. Der empfangende Super Peer löscht die Assoziation von `cp` zu `pro` aus seinem Datenbestand.

Parallel dazu wird aus `pro` die `hid` des Edge Peers isoliert und deren Schlüssel `k` bestimmt. Der Aufruf von `route(k, (REMPRO2EDGE, pro), NULL)` führt auf dem zuständigen Super Peer dazu, dass die Assoziation des Edge Peers zum ausgefallenen Provider gelöscht wird.

`void addEdgePeer(EdgePeerInfo edg)` *Registrieren eines neuen Edge Peers* : Tritt einem Super Peer ein neuer Edge Peer bei, so ruft der Super Peer diese Funktion auf, um den Edge Peer im DPI zu registrieren. Im Gegensatz zu Providern ist die Suche nach Edge Peers nicht notwendig, um auf Kontextinformationen zuzugreifen. Der DPI benötigt folglich nur Informationen die das Bereinigen des DPI ermöglichen. Durch Aufruf von `addProvider` mit jeder Providerbeschreibung in `edg` können die Assoziation eines Edge Peer zu seinen Providern im DPI registriert werden. Dadurch wird f_{E-PRO} (Gl. 5.16) ermöglicht. Darüber hinaus muss für f_{S-E} (Gl. 5.17) die Assoziation des Super Peers zu einem neuen Edge Peer aufgebaut werden.

Bisher wurde eine Assoziation von einem übergeordneten Objekt (z. B. Edge Peer) zu einem untergeordneten Objekt (z. B. Provider) dadurch repräsentiert, dass im DPI unter dem Bezeichner des übergeordneten Objektes die Beschreibung des untergeordneten Objektes abgespeichert wurde. Zur Übertragung der zugehörigen Nachricht wurde aus dem Bezeichner des übergeordneten Objektes (z. B. dem `hid` des Edge Peers) ein Schlüssel `k` berechnet. Wendet man dieses Vorgehen auf die Assoziation zwischen Super Peer und Edge Peer an, so stellt man fest, dass die Ermittlung von `k` unnötig ist*, und folglich der Super Peer die Nachricht an sich selbst senden würde.

Weil die herkömmliche Speicherung der Assoziation des Super Peers zu dem neuen Edge Peer im DPI dazu führt, dass diese Informationen auf dem betroffenen

*`k` ist bereits Teil der `HostId` des Super Peers, welche die Form eines `NodeHandle` hat.

Super Peer selbst gespeichert werden, sind diese Informationen beim Ausfall des Super Peers ebenfalls nicht verfügbar. Eine Bereinigung des DPI ist nicht möglich weil durch die fehlenden Informationen die Abbildung f_{S-E} nicht möglich ist, und somit weder die Edge Peers des Super Peers noch deren Provider festgestellt werden können (Gl. 5.17). Deshalb wird die Registrierung durch Verteilung der Informationen an die benachbarten Super Peers vorgenommen. Sie sind es auch, welche den Ausfall des Super Peers erkennen können, und die Informationen über dessen Edge Peers benötigen. Folglich führt der Super Peer mehrere Aufrufe von `route(NULL, (ADDEDGE2SUPER, hid, S), N)` durch, wobei `hid` die `HostId` des Edge Peers ist, und `N` jeweils den Wert des `NodeHandle` eines Nachbarn des Super Peers annimmt. Jeder Nachbar speichert lokal die Assoziation zwischen dem `NodeHandle S` des Super Peer und der `HostId hid` des Edge Peers ab.

Abschließend wird die Funktion `addProvider` für jede Providerbeschreibung aus `edg` aufgerufen, welche die Registrierung der ebenfalls neuen Provider zur Folge hat.

`void remEdgePeer(HostId hid)` *Entfernen eines Edge Peers* : Zur Bereinigung des DPI infolge des Ausfalls eines Edge Peers, muss festgestellt werden, welche Provider zu dem betroffenen Peer gehören. Diese müssen ebenso aus dem DPI entfernt werden wie die Assoziation des Edge Peers zu seinem Super Peer.

Mit der Hash-Funktion kann aus `hid` der Schlüssel `k` berechnet werden, der für den Aufruf von `route(k, (REMEDGEINFO, hid, S), NULL)` benutzt wird. Der zuständige Super Peer entfernt den mit `hid` bezeichneten Edge Peer aus den Verwaltungsstrukturen und übermittelt die Beschreibungen der Provider des Edge Peers mittels `route(NULL, ({pro}), S)` zurück an den Super Peer.

Der Super Peer extrahiert aus jeder Providerbeschreibung die `ProviderId pid` und ruft jeweils `remProvider(pid)` damit auf.

`void addSuperPeer()` *Registrieren eines Super Peers* : Die Registrierung nimmt jeder Super Peer beim Beitritt zum Super-Peer-Netzwerk selbst vor. Hat er zu diesem Zeitpunkt bereits Edge Peers, so ruft er mit jeder ihrer `EdgePeerInfo edg` die Funktion `addEdgePeer(edg)` auf. Durch die resultierenden Registrierungen werden bereits alle erforderlichen Informationen im DPI abgelegt.

`void remSuperPeer(HostId hid)` *Entfernen eines Super Peers* : Wie bereits erwähnt, kann der Ausfall eines Super Peers `hid` nur von seinen Nachbarn erkannt werden. Stellt ein Nachbar den Ausfall fest, so benachrichtigt er die übrigen Nachbarn und führt die Bereinigung des DPI durch.

Um die gleichzeitige Behandlung des Ausfalls durch alle Nachbarn zu verhindern, sendet jeder Nachbar an die übrigen, ehemaligen Nachbarn des Super Peers eine ent-

sprechende Nachricht – `route(NULL, (REMSUPERINFO, hid), N)`. Alle empfangenden Super Peers beseitigen lokal die zu `hid` gehörigen Registrierungsinformationen.

Die Nachricht des ersten Nachbars führt dazu, das alle übrigen Nachbarn lediglich ihre lokalen Informationen bereinigen. Er selbst ruft `remEdgePeer(hid)` mit den `hid` der Edge Peers des ausgefallenen Super Peers auf, bevor er seine eigenen lokalen Informationen bereinigt.

Sollte ein Super Peer planvoll das Netzwerk verlassen, so ruft er die Funktion auf sich selbst auf und veranlasst damit die Bereinigung.

5.6.5. Aufwandsbetrachtungen

Zum Vergleich mit der zentralen Verwaltung soll der Kommunikationsaufwand für die Aktualisierung des globalen Zustandes und für die Anfrage nach Providerbeschreibungen untersucht werden.

Kommunikationsaufwand – gesamt

Fällt ein Provider aus, so benachrichtigt dessen Edge Peer den übergeordneten Super Peer, welcher den Provider aus dem DPI entfernt. Für den vollständigen Vorgang sind $2 + (p + 2) \times (log\, s)$ Übertragungen* notwendig (siehe `remProvider`). Wird ein Provider verfügbar, ist der Aufwand ähnlich hoch – $1 + (p + 2) \times (log\, s)$ (siehe `addProvider`). Der mittlere Aktualisierungsaufwand des hybriden P2P-Netzwerkes $m_{UPD_{p2p}} = 1 + (p+2) \times (log\, s)$ ist signifikant höher als der des C/S-Netzwerkes mit hybridem Austausch $m_{UPD_{hy}} = 1$.

Bei einer Suchanfrage nach einem geeigneten Provider kontaktiert ein Edge Peer nicht nur seinen Super Peer, sondern letzterer greift auch auf den DPI zu (siehe `reqProviders`). Der Anfrageaufwand $m_{REQ_{p2p}}$ liegt mit $2 + log\, s$ erneut signifikant über $m_{REQ_{hy}} = 2$.

Betrachtet man den Kommunikationsaufwand der für die Bedienung der Consumer, sowie die Behandlung der Zuständsänderungen der Provider eines Knotens erforderlich ist, so birgt der dezentrale Ansatz keinen Vorteil. Diese Tatsache wird auch nicht durch die Ausweitung der Betrachtung auf den Kommunikationsaufwand im gesamten System beeinflusst, weil die Anzahl der Knoten, Quellen und Senken, welche den Aufwand verursachen, immer konstant bleibt.

Kommunikationsaufwand – pro Server

Der Schritt zu einer dezentralen Lösung wurde jedoch nicht wegen des Kommunikationsaufwandes gemacht, sondern aufgrund der mangelnden Skalierbarkeit des

* s ist die Anzahl der Super Peers; p ist die Anzahl der Provided Pattern des Providers

zentralen Servers. Demzufolge ist die Verteilung des gesamten Kommunikationsaufwandes auf den einzelnen Server bzw. Super Peer zu untersuchen. Diese Betrachtung spiegelt auch das Verhältnis zwischen Kontextquellen *pro* und Kontextsenken *con* wider. Während der serverbezogene Anfrageaufwand (Gl.5.18) von Ersteren abhängig ist, wirken sich Letztere auf den serverbezogenen Aktualisierungsaufwand (Gl.5.19) aus.

$$m_{REQ/S} = \frac{m_{REQ} \times con}{s} \tag{5.18}$$

$$m_{UPD/S} = \frac{m_{UPD} \times pro}{s} \tag{5.19}$$

Anfrageaufwand pro Server Betrachtet man den Kommunikationsaufwand eines einzelnen Super Peers, so wird erkennbar wie weit sich die Last gegenüber der eines zentralen Servers reduziert. Stellen alle 125.000 Kontextsenken des Anwendungsszenarios* eine Suchanfrage an einen zentralen Server, so muss dieser ein Anfrageaufwand von 250.000 Nachrichten bewältigen(Gl. 5.20).

$$\begin{aligned} m_{REQ/S_{hy}} &= \frac{m_{REQ_{hy}} \times con}{s} \\ &= \frac{2 \times 125.000}{1} \\ &= 250.000 \end{aligned} \tag{5.20}$$

Bei der dezentralen Verwaltung verteilt sich der Aufwand auf die Super Peers. Geht man davon aus, dass ein Super Peer nur einhundert Edge Peers ($e = 100$) verwaltet, so entfällt bei gleichzeitiger Anfrage aller Kontextsenken auf jeden der 7.400 Super Peers ein Anfrageaufwand von rund 13 Nachrichten (Gl. 5.21). Somit ist die Anfragelast um vier Größenordnungen kleiner, als die des zentralen Servers.

*siehe Kap. 3.5.2 und Tab. 3.6 auf Seite 55

$$
\begin{aligned}
m_{REQ/S_{p2p}} &= \frac{m_{REQ_{p2p}} \times con}{s} \\
&= \frac{(2 + ld\, s) \times 12.500}{s} \\
&= \frac{(2 + ld\,(\frac{n}{e})) \times 12.500}{\frac{n}{e}} \\
&= \frac{(2 + ld\,(\frac{740.000}{100})) \times 12.500 \times 100}{740.000} \\
&= \frac{(2 + ld\, 7.400) \times 125}{74} \\
&\approx 13
\end{aligned}
\tag{5.21}
$$

Um von einer Lastreduzierung zu profitieren, sind noch weniger Super Peers erforderlich. Bereits beim Einsatz von zwei Super Peers hat jeder Super Peer eine niedrigere Anfragelast als der zentrale Server zu bewältigen (Gl. 5.22; Abb. 5.19 auf der nächsten Seite). Erhöht man die Anzahl der Super Peers, sinkt die Last weiter. In den folgenden Diagrammen (Abb. 5.19, 5.20) ist der Kommunikationsaufwand des hybriden P2P-Systems in Abhängigkeit von der Anzahl seiner Super Peers dem konstanten Aufwand des C/S-Systems gegenübergestellt.

$$
\begin{aligned}
m_{REQ/S_{hy}} &= m_{REQ/S_{p2p}} \\
\frac{m_{REQ_{hy}} \times con}{1} &= \frac{m_{REQ_{p2p}} \times con}{s} \\
m_{REQ_{hy}} &= \frac{m_{REQ_{p2p}}}{s} \\
2 &= \frac{2 + ld\, s}{s} \\
2 \times s - 2 &= ld\, s \\
s &= 1
\end{aligned}
\tag{5.22}
$$

Aktualisierungsaufwand pro Server Die Schwelle für die Lastreduzierung des Aktualisierungsaufwandes ist höher (Gl. 5.23; Abb. 5.20 auf Seite 149)* als die des Anfrageaufwandes. Erst wenn mindestens 18 Super Peers betrieben werden, muss jeder dieser Super Peers weniger Last bewältigen, als der zentrale Server.

*Es wird angenommen, dass jeder Provider zwei Provided Pattern unterstützt ($p = 2$).

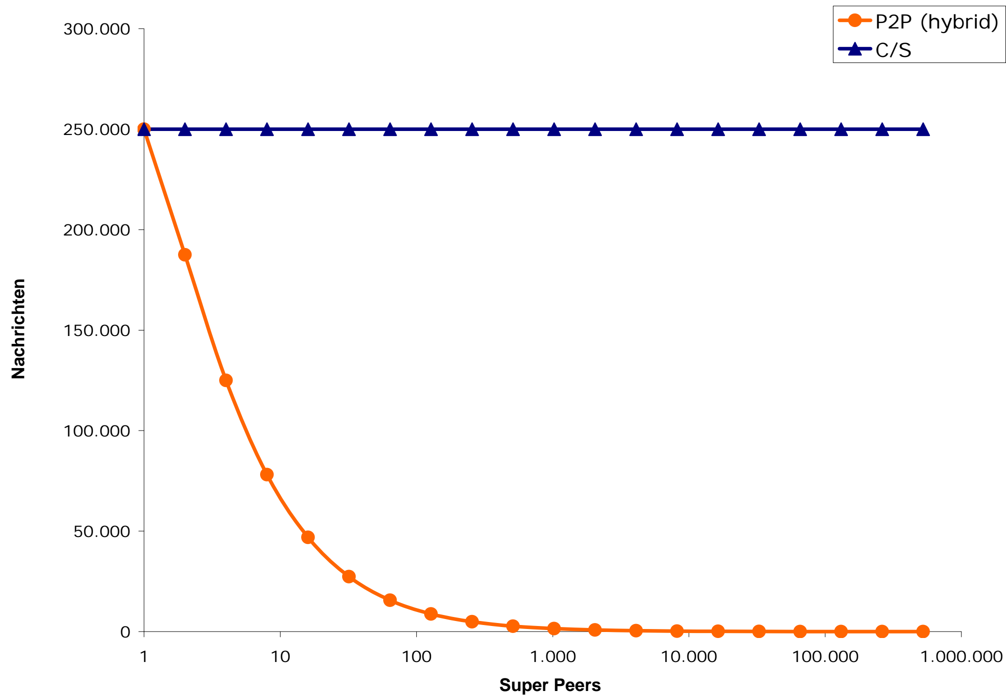

Abbildung 5.19.: Anfrageaufwand der dezentralen Verwaltung $m_{REQ_{p2p}}$ in Abhängigkeit von der Anzahl der Super Peers verglichen mit dem Anfrageaufwand der zentralen Verwaltung $m_{REQ_{c/s}}$

$$\begin{aligned}
m_{UPD/S_{hy}} &= m_{UPD/S_{p2p}} \\
\frac{m_{UPD_{hy}} \times pro}{1} &= \frac{m_{UPD_{p2p}} \times pro}{s} \\
m_{UPD_{hy}} &= \frac{m_{UPD_{p2p}}}{s} \\
1 &= \frac{1 + (p+2) \times (ld\, s)}{s} \\
s &= 1 + 4 \times (ld\, s) \\
\frac{s-1}{4} &= ld\, s \\
s &< 18
\end{aligned} \tag{5.23}$$

Ändert sich der Zustand aller 21 Millionen Kontextquellen aus dem bereits skizzierten Beispiel, so entfällt auf jeden der Super Peers eine Last von ca. 150.000

Nachrichten (Gl. 5.24). Gegenüber der Last des zentralen Servers von 21 Millionen Nachrichten ist dies eine Reduzierung um zwei Größenordnungen.

$$\begin{aligned} m_{UPD/S_{p2p}} &= \frac{m_{UPD_{p2p}} \times pro}{s} \\ &= \frac{(1 + (p + 2) \times (ld\, s)) \times 21.000.000}{s} \\ &= \frac{(1 + 4 \times (ld\, 7.400)) \times 21.000.000}{7.400} \\ &\approx 150.405 \end{aligned} \tag{5.24}$$

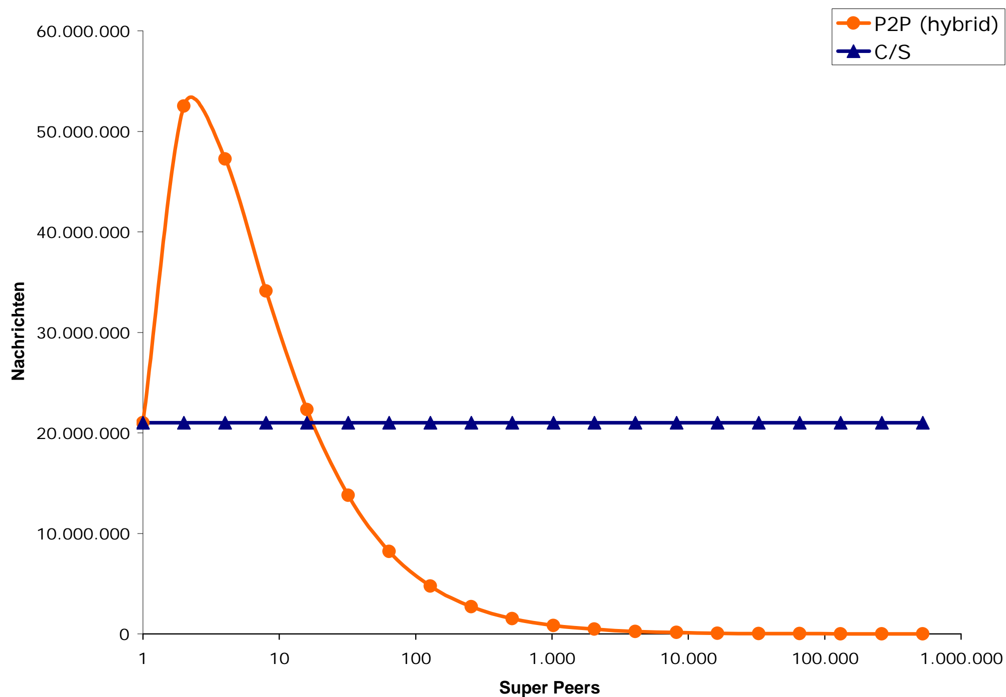

Abbildung 5.20.: Aktualisierungsaufwand der dezentralen Verwaltung $m_{UPD_{p2p}}$ in Abhängigkeit von der Anzahl der Super Peers verglichen mit dem Aktualisierungsaufwand der zentralen Verwaltung $m_{UPD_{c/s}}$

Fazit

Die Betrachtungen zeigen, dass die dezentrale Verwaltung zu einer signifikanten Erhöhung des Nachrichtenaufkommens führt. Wesentlich wichtiger ist die Tatsache,

dass sie eine weitaus höhere Reduzierung der Last des einzelnen Servers bzw. Super Peers zur Folge hat. Die Super Peers sind somit in der Lage, die Skalierbarkeit des Kontextdienstes zu verbessern, welche in erster Linie durch die Leistungsfähigkeit und die Ressourcen des zentralen Servers limitiert wird.

Abschließend sei bemerkt, dass im skizzierten Anwendungsszenario (Tab. 3.6 auf Seite 55) fast 1.000 Serverknoten vorhanden sind, welche den Betrieb eines Super Peers ohne weiteres erlauben. Über 6.000 weitere Knoten – Arbeitsplatzrechner – sind ebenfalls leistungsfähig und ggf. dauerhaft verfügbar. Es sind somit die praktischen Voraussetzungen dafür gegeben, nahezu 7.400 Super Peers zu betreiben und die genannten Größenordnungen der Lastreduzierung zu erreichen.

6. Realisierung und Evaluation

Die evaluierende Untersuchung des vorgestellten Context Service und der gewählten Lösungsansätze ist Gegenstand dieses Kapitels. Um die Bewertung vornehmen zu können, wurde ein Prototyp entwickelt, dessen Realisierung im Folgenden beschrieben wird (Kap. 6.1 bis 6.5). Auf die Integration des Context Service – als Teil einer Laufzeitumgebung für adaptive, kontextbezogene Software – in einen modellgetriebenen Software-Lebenszyklus, wird in Kapitel 6.6 eingegangen. Abschließend werden die Anforderungen der Kapitel 2 und 3 anhand eines kurzen Szenarios bewertet.

6.1. Realisierung des Context Service

Um den Context Service – entsprechend des Einsatzszenarios – auf mobilen und integrierten Geräten betreiben zu können, wurde er auf Basis der Programmiersprache Java realisiert. Für sie stehen entsprechende Ausführungsumgebungen (z. B. für PDA auf Basis von Microsoft Windows Mobile) zur Verfügung. Die daraus erwachsenden technischen Grundlagen werden nachfolgend beschrieben, bevor Details der Realisierung vorgestellt werden.

Aufgrund des beschränkten Platzes enthalten nachfolgend präsentierte Darstellungen nur die wichtigsten Elemente, welche im zugehörigen Text erläutert werden. Unwichtige Details werden ausgelassen. Treten Elemente in mehreren Darstellungen auf, so kann in einem Teil davon eine verkürzte Präsentation erfolgen. Für die

vorgestellten Klassendiagramme gelten folgende Konventionen:

- Bei Schnittstellen (Interfaces) sind Klassen- und Methodenbezeichner kursiv gesetzt. Weiterhin besitzen sie keine Attribute, deshalb werden die Methoden direkt unter dem Klassennamen aufgeführt.
- Bei abstrakten Klassen die die Klassennamen ebenfalls kursiv gesetzt. Für die Methodenbezeichner gilt dies nur dann, wenn die Methode abstrakt, d. h. definiert, aber nicht implementiert ist. Diese Klassen können Attribute besitzen und bestehen deshalb aus zwei Teilen – der obere kann Attribute, der untere dagegen Methoden enthalten.
- Die Klassen- und Methodenbezeichner regulärer Klassen sind im Standardschriftstil gesetzt. Sie können ebenfalls Attribute und Methoden besitzen.
- Zugriffsmodifikatoren für Attribute und Methoden sind nicht dargestellt.

Generell sind Klassen-, Schnittstellen- und Methodenbezeichner im Text in diktengleicher Schreibmaschinenschrift gesetzt.

6.1.1. Technische Voraussetzungen

Für die Realisierung des Context Service werden zwei grundsätzliche Bausteine benötigt: einerseits eine Java Laufzeitumgebung für mobile und integrierte Geräte und andererseits eine leichtgewichtige Komponentenplattform. Letztere erlaubt die dynamische Komposition von Softwaresystemen, und somit auch des Context Services, sodass dessen Komponenten bedarfsabhängig installiert werden können. Auf diesem Weg können sowohl die Ressourcen der Geräte geschont werden, als auch an das Nutzungsszenario angepasste Konfigurationen des Context Service verwendet werden.

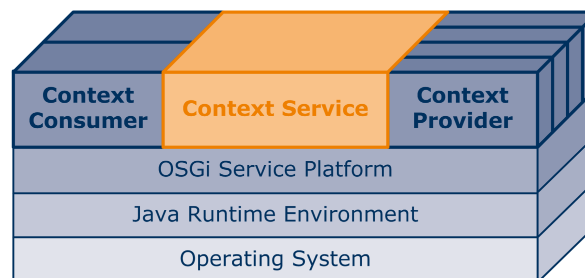

Abbildung 6.1.: Architektur des Laufzeitsystems

In Abbildung 6.1 ist eine Überblicksdarstellung der geschichteten Softwarearchitektur zu sehen. Ihre Basis wird durch das Betriebssystem gebildet, in dem die Java

Laufzeitumgebung ausgeführt wird, welche den Betrieb einer OSGi* Service Plattform gestattet. Die Bestandteile der obersten Schicht – der Context Service mit den Context Consumern und Context Providern† – greifen auf die Funktionalität der darunter liegenden Schichten zu.

Java Micro Edition Laufzeitumgebung

Für mobile und integrierte Geräte bietet Sun Microsystems, Inc. – Erfinder der Programmiersprache Java – die so genannte *Java Micro Edition (Java ME)* an. Teil der Spezifikation von Java ME ist zum einen die Ausführungsumgebung – auch Virtual Machine (VM) genannt – zum anderen der Funktionsumfang der standardmäßig verfügbaren Bibliotheken. Eine solche Ausführungsumgebung besteht aus mehreren Teilen – einer *Configuration*, einem *Profile* und beliebig vielen *Optional Packages*. Die Configuration spezifiziert die Eigenschaften der VM, sowie unverzichtbare grundlegende Bibliotheken. Das Profile definiert die standardmäßig verfügbaren Bibliotheken. Optional Packages stellen, wie der Name schon impliziert, optionale Teile dar, d. h. zusätzliche Bibliotheken, welche nur in bestimmten Einsatzfällen benötigt werden.

Die Spezifikation sieht zwei verschiedene Varianten der *Configuration* vor, welche für unterschiedlich leistungsfähige Geräte ausgelegt sind. Die *Connected-Limited Device Configuration (CLDC)*[Sun07] zielt auf leistungsschwächere, mobile Geräte ab, die nur über geringe Betriebsmittel (Rechenleistung, transienter und persistenter Speicher) und eine intermittierende Netzwerkverbindung verfügen. Dem gegenüber wurde die *Connected Device Configuration (CDC)* für leistungsstärkere Geräte entwickelt, die mehr Betriebsmittel und eine stabile Netzwerkverbindung besitzen. Letzteres Detail ist jedoch kein zwingender Grund (mehr) für die Trennung der beiden Ausführungsumgebungen. So sind bereits Smart Phones, d. h. „bessere“ Mobiltelefone, wie beispielsweise die Communicator-Modelle 9300, 9300i und 9500 von Nokia verfügbar, welche seitens des Herstellers mit einer CDC-basierten Laufzeitumgebung ausgeliefert werden. Die *Kilobyte Virtual Machine (KVM)* – VM der CLDC – ist stark eingeschränkt. So dürfen Programme nur wenige Kilobyte Hauptspeicher belegen. Der Bytecode der Software wird von der KVM im Gegensatz zu anderen VMs nicht auf Gültigkeit verifiziert. Folglich sind weitere Maßnahmen (z. B. Prä-Verifikation und kryptographische Signaturen) erforderlich, um die Validität von Software zu sichern, welche aus dem Netz geladen wird. Nicht zuletzt ist der Zugriff auf die Ressourcen des zugrunde liegenden Betriebssystems eingeschränkt. Dazu gehört, dass (a) die Software nur auf ihre eigenen Dateien zugreifen kann, und (b) die KVM das *Java Native Interface (JNI)* nicht unterstützt, mit dem der Zugriff auf

*Open Services Gateway Initiative (OSGi)

†Aufgrund des häufigen Gebrauchs der Begriffe Context Consumer und Context Provider werden diese nachfolgend nur Consumer und Provider genannt.

Betriebssystem- oder Treiberbibliotheken möglich wäre.

Für den Einsatz auf aktuellen mobilen und integrierten Geräten ist eine CDC-basierte Realisierung zu bevorzugen, weil deren VM einer VM der *Java Standard Edition (Java SE)* Version 1.4 entspricht. Sie legt der Software geringere Speicherbeschränkungen auf, unterstützt das JNI und bietet sogar Bytecode-Verifikation. Speziell die Nutzung von JNI kann bei der Entwicklung von Providern erforderlich werden, wenn die Daten von Sensoren oder anderen Informationsquellen nur durch Zugriff auf spezielle Treiber oder Betriebssystemfunktionen gewonnen werden können.

Aufbauend auf der CDC sind die verschiedenen Profile *Foundation Profile (FP)*, *Personal Basis Profile (PB)* und *Personal Profile (PP)* verfügbar – mit wachsendem Umfang und Funktionalität. Jedoch entspricht selbst die neueste Version – CDC + PP 1.1 – lediglich weitestgehend einer Java SE 1.4.2, welche seit Dezember 2006 nicht mehr aktuell ist, und ab Oktober 2008 von Sun Microsystems offiziell nicht mehr unterstützt werden wird*. Deshalb erfolgt die Realisierung des Context Service, zur Vermeidung umfangreicher Portierungsarbeiten der Bibliotheken benötigter Technologien, vorwiegend auf Basis einer Java SE 5.0. Die Ressourcen der mobilen Geräte stellen bei diesen Überlegungen kein Problem dar. Diejenigen Komponenten deren Anforderungen nicht über CDC + PP 1.1 hinausgehen, wurden in einer entsprechenden Umgebung† getestet.

OSGi Service Platform

Um den Context Service so realisieren zu können, dass einfach an die heterogenen Anforderungen der zu erwartenden Szenarien angepasst werden kann, ist eine Komponentenplattform wünschenswert, welche eine lose Kopplung von Komponenten ermöglicht. Andernfalls müsste die Verwaltung der dynamisch verfügbaren Komponenten als Teil desselben Systems realisiert werden.

Mit der *OSGi Service Platform* existiert die Spezifikation [OSG07a] einer Komponentenplattform für integrierte und mobile Geräte. Komponenten – in OSGi als *Bundles* bezeichnet – können dynamisch installiert und de-installiert werden. Sie stellen ihre Funktionalität zur Verfügung, indem sie entsprechende Dienstobjekte bei der Plattform registrieren. Dabei erfolgt die Beschreibung der Dienste in erster Linie durch die Menge von Java-Klassen oder -Schnittstellen, welche das registrierte Dienstobjekt unterstützt. Zusätzlich kann es durch eine Menge von Name-Wert-Paaren beschrieben werden, deren Bedeutung und Wertebereiche jedoch vollständig der Domäne des Dienstes unterliegen. Bundles können nicht nur Dienste zur Verfügung stellen, sondern auch selbst nutzen. Sie können dazu sowohl zu einem konkreten Zeitpunkt die Menge verfügbarer Dienste abfragen, als auch diese Menge

*Bei Erstellung dieser Arbeit hatte die aktuelle Version der Java SE die Versionsnummer 6.0.
†IBM WebSphere Micro Environment 6.1 bzw. J9 VM 2.3

langfristig überwachen, sodass sie Benachrichtigungen über die De-/Installation von Diensten und die Veränderungen ihrer Beschreibung erhalten. Neben der Bereitstellung von Diensten können Bundles auch zur Bereitstellung von Bibliotheken genutzt werden. Eine OSGi Plattform verwaltet sogar unterschiedliche Versionen der installierten Bundles, sodass Bundles während des Betriebes aktualisiert werden können. Die OSGi Plattform sorgt dafür, dass Bundles die das aktualisierte Bundle vor dessen Aktualisierung genutzt haben, weiterhin die „alte“ Version verwenden, während Bundles die erstmalig das aktualisierte Bundle nutzen dessen „neue“ Version erhalten. Auf diesem Weg können Anwendungen dynamisch installiert und verwaltet werden.

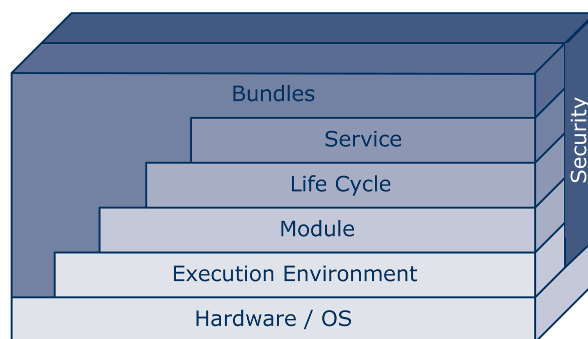

Abbildung 6.2.: Schichten der OSGi Service Platform (nach [OSG07a])

Abbildung 6.2 zeigt den Aufbau einer OSGi Plattform. Die Basis bildet immer das Betriebssystem, welches den Zugriff auf die Funktionen der Hardware des Geräts ermöglicht. Darüber befindet sich die Ausführungsumgebung, d. h. ein Java Runtime Environment (JRE) mit der VM – mindestens Java ME mit CDC. Die *Module*-Schicht enthält die Basisfunktionen der OSGi Plattform. Sie sorgt dafür, dass den Bundles die benötigten Packages (Bibliotheken) in den passenden Versionen zur Verfügung stehen. Darauf aufbauend steuert die *Life Cycle*-Schicht den Lebenszyklus der Bundles. Sie ist für Installation, Start, Stopp, Update, De-Installation und Überwachung der Bundles zuständig. Die *Service*-Schicht ermöglicht die Kommunikation zwischen den Bundles durch einen speziellen Ereignisdienst (Event Service). Weiterhin ist sie für das Binden der Dienste von Bundles anhand ihrer Schnittstellendefinition und Beschreibung zuständig. Zusätzlich zu diesen Grundfunktionen sind in dieser Schicht bereits einige Basisdienste definiert [OSG07b], welche von einer OSGi Plattform angeboten werden können, und die Realisierung von anwendungsspezifischen Bundles vereinfachen. In dieser Arbeit wird nur eine kleine Auswahl davon genutzt (Abb. 6.3 auf Seite 158). Die *Bundles* bilden die oberste Schicht, können aber nicht nur auf die Service-Schicht, sondern auch auf die anderen Schich-

ten zugreifen. Orthogonal zu allen Schichten befindet sich die *Security*-"Schicht", welche auf Basis von Java Sicherheitsmechanismen realisiert ist, und diese um OSGi-spezifische Funktionen erweitert. Sie ist für alle Schichten zugreifbar, und steuert deren Sicherheitsrelevante Abläufe.

Integration des Context Service in die OSGi Service Platform

Die Realisierung des Context Service findet aus mehreren Gründen auf Basis von OSGi statt:

1. Die Verwaltung der Komponenten/Bundles ermöglicht die einfache Unterstützung unterschiedlicher Nutzungsszenarien durch *angepasste Konfigurationen* des Context Service – z. B. Betrieb einer lokalen, selbstständigen Context Service Instanz ohne Kommunikationsfunktionalität. Somit können einerseits die Ressourcen der zugrunde liegenden Geräte geschont werden, und andererseits alternative Realisierungen von Komponenten mit derselben Schnittstelle einfach ausgetauscht werden (z. B. Peer-to-Peer- statt C/S-Kommunikation).

2. Die Signalisierungs- und Verwaltungsmechanismen der OSGi Service Platform erleichtern die Implementierung von Diensten, welche *dynamisch gesucht* und *überwacht* werden müssen, weil sie die Funktion eines Verzeichnisdienstes für OSGi-Bundles zur Verfügung stellen.

3. Die Eignung von OSGi für integrierte Geräte ebnet nicht nur den Weg zur Unterstützung ähnlich leistungsbeschränkter mobiler Geräte, sondern bringt auch *technologische Unterstützung* mit sich, welche für den Zugriff auf Sensoren oder Gebäudeautomationssysteme nützlich sein können. So definiert die OSGi-Spezifikation einen Dienst zum Zugriff auf UPnP*-Geräte.

Angepasste Installationen des Context Service müssen manuell zusammengestellt werden, d. h. der Administrator des jeweiligen Gerätes installiert die benötigte Auswahl der Komponenten in Abhängigkeit des Nutzungsszenarios. Jedoch ermöglicht OSGi die Entwicklung flexibler Komponenten, welche weitere optionale Komponenten nur dann nutzen, wenn diese auch über die Plattform erreichbar sind. Bei der Realisierung der Context Service Komponenten besteht folglich eine der Schwierigkeiten in der Aufteilung der Funktionen auf weitgehend voneinander unabhängige und ggf. austauschbare Einheiten. Zukünftig ist vorstellbar, dass fehlende Komponenten dynamisch und automatisch installiert werden können. Jedoch wäre dafür nicht nur eine entsprechende Modellunterstützung erforderlich, welche spezifiziert, unter welchen Bedingungen welche Komponenten woher und wie zu installieren sind,

*Universal Plug and Play (UPnP)

sondern auch eine entsprechende Komponente, die dieses Modell interpretiert/auswertet, um die gewünschte Funktion zu erbringen. Eine entsprechende Implementierung benötigt Rechenzeit und transienten Speicherplatz, sowie Netzwerkressourcen, welche auf mobilen und integrierten Geräten äußerst knapp sind. Persistenter Speicherplatz hingegen ist dank Flash-Speichermedien mit mehreren Gigabyte Kapazität weniger problematisch. Folglich ist eine automatische Lösung zwar wünschenswert und möglich, jedoch unter dem Gesichtspunkt der Leistungsfähigkeit des Systems voraussichtlich verzichtbar. OSGi selbst bietet bereits Mittel zur bedingten und automatischen Installation von Bundles. Jedoch wurden diese bisher nicht näher untersucht. Die nachfolgend beschriebene Architektur ermöglicht durch ihre Modularisierung eine spätere Umstellung des Context Service auf eine effizientere und möglichst automatische Konfiguration.

Dynamische Suche und Überwachung ist nicht nur für diejenigen OSGi-Bundles notwendig, welche die Komponenten des Context Service bilden. Ihre möglichen Konfigurationen sind während der Laufzeit voraussichtlich weitgehend statisch, weil der Context Service einen Basisdienst darstellt, welcher jederzeit für Consumer nutzbar sein sollte. Wesentlich dynamischer ist die Verfügbarkeit der Kontextquellen und entsprechend auch ihrer Provider. Deshalb werden die Schnittstellenimplementierungen der Provider als Dienste* in der OSGi Service Platform registriert, und ggf. sogar deregistriert. Mit Hilfe so genannter `ServiceTrackerCustomizer`, welche der Context Service – genauer gesagt der `ContextBroker` – bei der OSGi Service Platform registriert, erhält er Benachrichtigungen, wenn Providerdienste registriert und deregistriert werden. Wie dies im Detail funktioniert, wird in Kapitel 6.3 beschrieben. Auf die gleiche Weise überwachen die Komponenten des Context Service gegenseitig ihre Verfügbarkeit, sodass andere Komponenten genutzt werden, sobald sie gestartet sind.

6.2. Context Service Architektur

In Abbildung 6.3 auf der nächsten Seite sind die Bundles der einzelnen Komponenten des Context Service dargestellt. Außerdem enthält sie diejenigen Bundles der OSGi Service Platform, welche vom Context Service genutzt werden. Die Bundles der Consumer und Provider werden ebenfalls in der OSGi Service Platform ausgeführt, jedoch besitzen Provider (Kap. 6.3) und Consumer (Kap. 6.4) häufig Bestandteile die außerhalb der OSGi Service Platform realisiert sind – die eigentlichen Kontextsenken und Kontextquellen.

*Nachfolgend auch Providerdienste genannt.

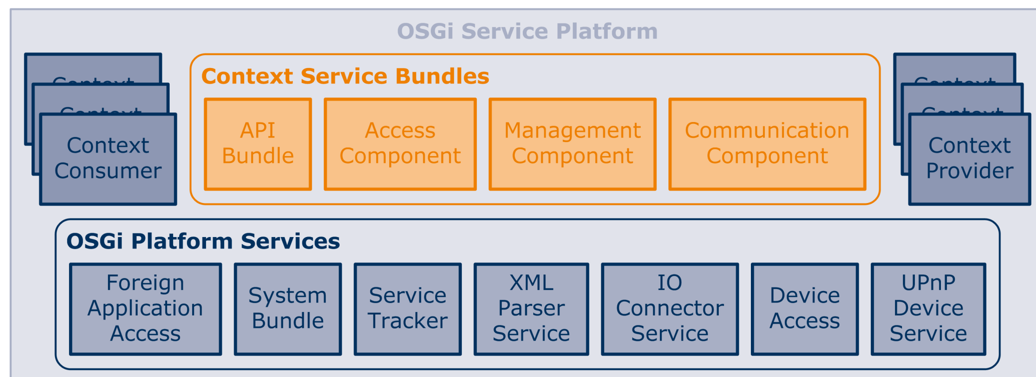

Abbildung 6.3.: Bundles der Consumer, des Context Service und der Provider, sowie genutzte Bundles der OSGi Service Platform

6.2.1. Mögliche Konfigurationen des Context Service

Für den Betrieb des eigentlichen Context Service stehen vier Bundles zur Verfügung:

- Das *API Bundle* `de.tud.mocawi.api`* beinhaltet die Klassen und Schnittstellen, welche für die Kooperation der übrigen Komponenten, sowie die Entwicklung von Providern und Consumern erforderlich sind. Die Bestandteile des Bundles werden bei der Betrachtung der übrigen Komponenten erläutert.
- Das Bundle `de.tud.mocawi.access` stellt die *Zugriffskomponente* (Kap. 6.2.2) bereit die den Consumern Zugriff auf Kontextinformationen von Providern erlaubt.
- Im Bundle `de.tud.mocawi.broker` befindet sich die *Verwaltungskomponente* (Kap. 6.2.3) die alle lokalen Provider verwaltet.
- Und die *Kommunikationskomponente* (Kap. 6.2.4) wird durch das Bundle `de.tud.mocawi.comm` realisiert.

Es sind verschiedene Konfigurationen des Context Service vorstellbar (Abb. 6.4 auf der nächsten Seite). Bisher wurde häufig von einem vollständigen Context Service – Konfiguration A – ausgegangen, bei dem alle Komponenten, sowie Provider und Consumer in einer OSGi Service Platform ausgeführt werden, und der über die Kommunikationskomponente an entfernte Context Service Instanzen gekoppelt ist. Sollte die Anbindung entfernter Instanzen nicht erforderlich oder möglich sein,

*Mobile Context-Awareness Infrastructure (MOCAWI) ist die Bezeichnung des Context Service.

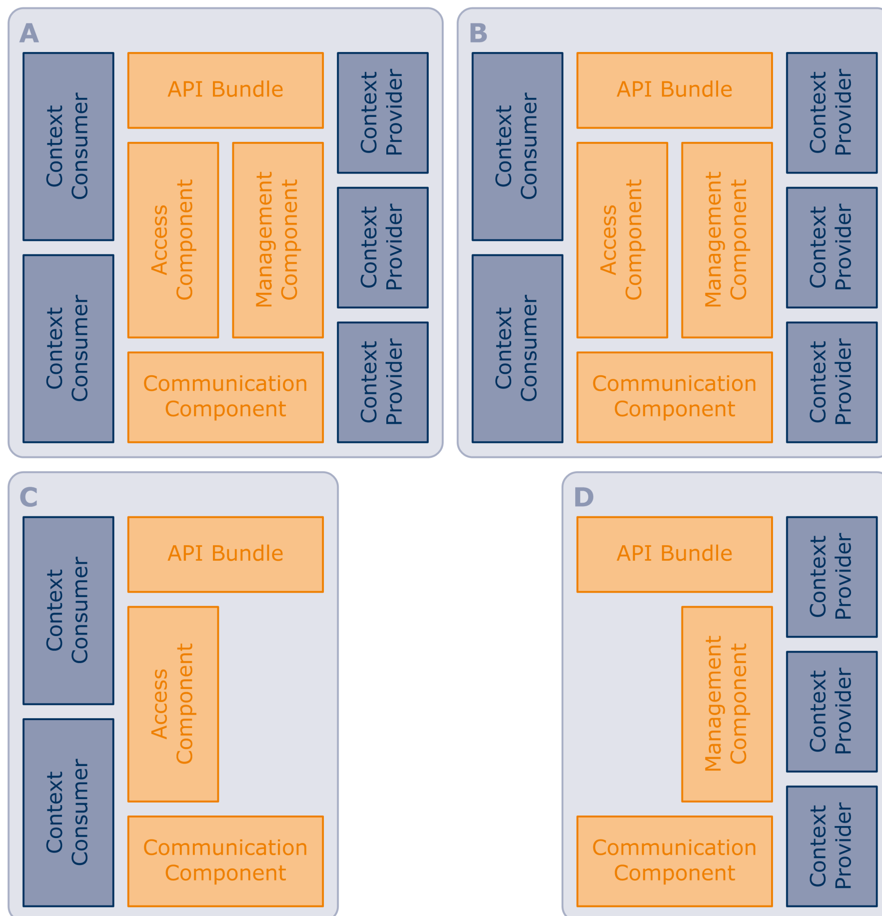

Abbildung 6.4.: Sinnvolle Konfigurationen des Context Service A) vollständig und verteilt; B) vollständig, nur lokal; C) nur Kontextnutzung und D) nur Kontextbereitstellung

so kann die Kommunikationskomponente entfernt werden – Konfiguration B. Sind dagegen lokal keine Provider vorhanden, so ist die Kommunikationskomponente erforderlich, um die Consumer mit den Kontextinformationen entfernter Provider zu bedienen – Konfiguration C. Ebenso ist die Kommunikationskomponente notwendig, wenn lokal keine Consumer existieren, die Kontextinformationen der lokalen Provider jedoch von entfernten Consumern benötigt werden, und entsprechend verfügbar gemacht werden müssen – Konfiguration D. Andere Konfigurationen sind nicht sinnvoll einsetzbar.

Neben den grundsätzlich möglichen Konfigurationen des Context Service sind auch unterschiedliche Versionen der einzelnen Komponenten einsetzbar. Natürlich immer unter der Voraussetzung, dass sie über die gleichen Schnittstellen und Klassen kooperieren, d. h. auf der gleichen Version des API Bundles basieren, das somit Teil einer jeden Konfiguration ist. Unterschiedliche Versionen dieses zentralen Bundles sind seltener zu erwarten – lediglich bei grundlegenden Änderungen/Erweiterungen der Funktionalität des Context Service. Dagegen ist jedoch die Nutzung alternativer Kommunikationstechnologien durch entsprechende Versionen der Kommunikationskomponente vorstellbar – z. B. Austausch der C/S-basierten Kommunikation durch den hybriden Peer-to-Peer-Ansatz.

Hinweis 1. *Consumer- und Providerimplementierungen sollten die schwankende Verfügbarkeit der Context Service Komponenten aufgrund wechselnder Konfigurationen behandeln, um die dynamische Rekonfiguration zukünftiger Context Service Implementierungen bzw. der OSGi Service Platform zu unterstützen. Dem entsprechend müssen sie die Dienste beobachten, welche die `QueryAndSubscribe`-, `ContextService`- und `ContextBroker`-Schnittstellen unterstützen.*

Die Nutzung der Bundles und Schnittstellen der OSGi Service Platform wurde bereits angesprochen und wird nachfolgend bei der Betrachtung der einzelnen Komponenten des Context Service ggf. genauer erläutert.

6.2.2. Zugriffskomponente – `de.tud.mocawi.access`

Zentraler Bestandteil der Zugriffskomponente (Abb. 6.5 auf der nächsten Seite) ist der `ContextManager`. Er implementiert die `QueryAndSubscribe`-Schnittstelle. Rufen Consumer die Methoden dieser Schnittstelle auf, um auf Kontextinformationen zuzugreifen, so führt der Manager stellvertretend für sie den Zugriff auf die Provider durch. Er steuert dabei auch die Aktivierung und Passivierung der Provider. Bei der Bedienung von Abonnements behandelt er die dynamische Verfügbarkeit von Providern durch Selektion alternativer Provider.

Im Zuge der Bedienung von Anfragen und Abonnements muss sich der Manager beim `ContextBroker` durch Aufruf seiner `subscribeProviders(ContextPattern,`

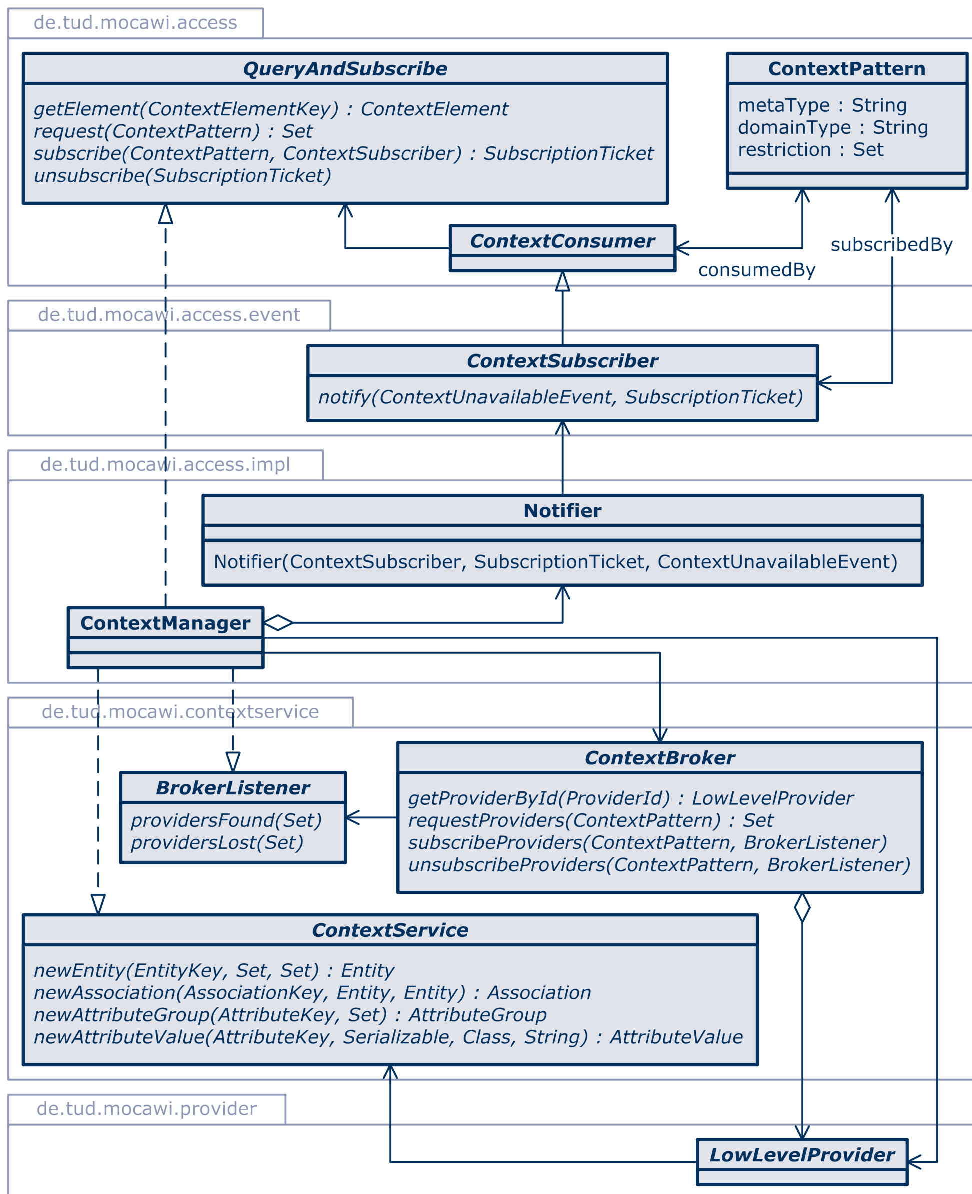

Abbildung 6.5.: Zentrale Klassen und Schnittstellen der Zugriffskomponente

`BrokerListener)`-Methode für Benachrichtigungen über nicht-verwendbare und verwendbare Provider registrieren. Als Voraussetzung dafür implementiert der Manager die `BrokerListener`-Schnittstelle. Sie bietet dem Broker die Methoden `providersFound(Set)` und `providersLost(Set)` zur Benachrichtigung. Beendet ein Consumer* ein Abonnement so prüft der Manager, ob die bisher abonnierten Kontextinformationen noch von einem anderen *Subscriber* abonniert sind. Ist das nicht der Fall beendet er den Zugriff auf den Provider durch Aufruf seiner `unsubscribe()`-Methode und auch die Überwachung alternativer Provider über die `unsubscribeProviders(ContextPattern, BrokerListener)`-Methode des Brokers.

Hinweis 2. *Weil die Zugriffskomponente von der Managementkomponente abhängig ist, registriert sie ihren eigenen Dienst der `QueryAndSubscribe`-Schnittstelle erst dann bei der OSGi Service Platform, wenn über diese der Dienst der `ContextBroker`-Schnittstelle verfügbar ist. Sobald diese nicht mehr verfügbar ist, wird auch die `QueryAndSubscribe`-Schnittstelle de-registriert.*

Greift ein Consumer über die Methode `QueryAndSubscribe.request(ContextPattern)` synchron auf Kontextinformationen zu, so sucht der Manager über die `requestProvider(ContextPattern)`-Methode des Brokers nach geeigneten Providern. Die Kommunikationskomponente unterstützt ebenfalls die `ContextBroker`-Schnittstelle und wird vom Manager genutzt, um Provider entfernter Context Service Instanzen zu finden. Die Kontextinformationen der gefundenen Provider werden abgefragt, ggf. gefiltert und anschließend als Ergebnis des Methodenaufrufs an die Consumer ausgeliefert.

Bei der Registrierung eines Abonnements durch den Subscriber mittels `QueryAndSubscribe.subscribe(ContextPattern, ContextSubscriber)` erfolgt die Suche, wie zuvor beschrieben, durch Abonnieren geeigneter Provider. Erhält der Manager eine entsprechende Benachrichtigung, so registriert er sich seinerseits bei den gelieferten Providern für Benachrichtigungen über Kontextänderungen. Wenn ein Provider eine solche Benachrichtigung vornimmt, filtert der Manager ggf. die Informationen und liefert sie anschließend an alle Subscriber aus, welche diese Information abonniert haben. Stellt der Manager über die Methode `providersLost(Set)` fest, dass keine Provider für die Bedienung eines Abonnements verfügbar sind, so signalisiert er dies den Subscribers durch Aufruf ihrer `notify(ContextUnavailableEvent, SubscriptionTicket)`-Methode mit einem Event dessen Schlüssel leer, d. h. `null`, ist.

*Consumer die per Abonnement, d. h. asynchron, auf Kontext zugreifen, werden nachfolgend auch *Subscriber* genannt.

Ereignisse von Kontextelementen

Die Klasse `ContextUnavailableEvent` bildet die Oberklasse einer Hierarchie von Ereignissen, die für die Signalisierung von Änderungen der unterschiedlichen Kontextelementtypen existieren (Abb. 6.6). Ein `ContextUnavailableEvent` wird immer dann an einen Subscriber ausgeliefert, wenn von ihm abonnierte Kontextinformationen nicht mehr verfügbar sind – meist, weil der zugrunde liegende Provider nicht mehr verfügbar ist. Da dieser Kontext nicht mehr zugreifbar ist, enthält die Klasse lediglich dessen Schlüssel – zugreifbar über `getElementKey()`. Ist nicht statt einer einzelnen Information die ganze Menge der abonnierten Informationen nicht-verfügbar, so wird ein Schlüssel mit dem Wert `null` übergeben. Während `EntityEvent` und `AssociationEvent` – durch die Methoden `isCreated()` und `isDestroyed()` – die Erzeugung oder Zerstörung einer Entität oder Assoziation signalisieren können, zeigt ein `AttributeValueEvent` immer die Änderung eines Attributs an, dessen neuer Wert ohne Umwege mittels `getNewValue()` zur Verfügung steht.

Hinweis 3. *Es liegt in der Verantwortung des Subscribers, Kontextereignisse zeitnah zu verarbeiten. Andernfalls kann es dazu führen, dass der signalisierte Kontext zum Zeitpunkt seiner Verarbeitung durch nachfolgende Ereignisse bereits invalidiert ist.*

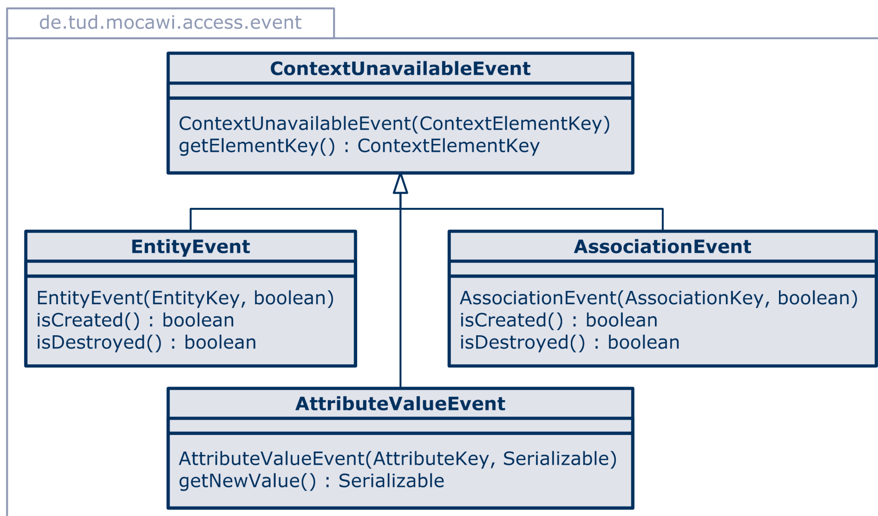

Abbildung 6.6.: Klassen für die Kontextereignisse der unterschiedlichen Typen des Metamodells

Robuste Benachrichtigung der Subscriber

Weil die Benachrichtigung der Subscriber durch synchrone Methodenaufrufe erfolgt, ist der Context Service darauf angewiesen, dass Subscriber die übergebenen Informationen schnell verarbeiten. Dies kann jedoch nicht immer sichergestellt werden, und das Attribut „schnell“ müsste quantifiziert werden. Darüber hinaus können Fehler innerhalb der Subscriber auftreten, deren Behandlung zu Verzögerungen führt. Deshalb erzeugt der Manager bei jeder Benachrichtigung einen separaten `Notifier`-Prozess, welcher den Aufruf der `notify(ContextUnavailableEvent, SubscriptionTicket)`-Methode beim Subscriber vornimmt, um den Context Service von diesem weitgehend zu entkoppeln. Natürlich sind nach wie vor Beeinträchtigungen des Context Service durch fehlerhafte oder ineffiziente Subscriber möglich. Benötigt ein Subscriber besonders viel Rechenzeit, so kann sich dies auch über den Scheduler der Virtual Machine auf den Context Service auswirken, weil alle Bundles einer OSGi Service Platform in derselben Virtual Machine Instanz ausgeführt werden. Abhilfe wäre durch entsprechende Echtzeitunterstützung [Sun04] möglich, deren Einsatz jedoch noch näher zu untersuchen wäre.

Repräsentation der Kontextinformationen zur Laufzeit

Für den Zugriff auf Kontextinformationen sind im API Bundle Schnittstellen für die drei Metamodelltypen definiert. Weiterhin stellt das API Bundle die Implementierungen dieser Schnittstellen bereit, welche von den Providern verwendet werden müssen, um ihre Kontextinformationen an den Context Service zu übergeben.

In Abbildung 6.7 auf der nächsten Seite sind die Schnittstellen der Kontextinformationen* dargestellt. Die meisten ihrer Methoden bieten lediglich die Semantik, welche bei der Diskussion des Metamodells (Kap. 5.3) bereits vorgestellt wurde. So verfügt jede Kontextinformation, d. h. jedes `ContextElement`, über einen eindeutigen Schlüssel – den `ContextElementKey`. Dieser enthält neben den Meta- und Domänentypen auch den typweit eindeutigen Bezeichner. Die darüber hinaus enthaltene `ProviderId` verweist auf den Provider der die Information geliefert hat, und dient lediglich zur Beschleunigung des Zugriffs auf diesen. Solange der Provider verfügbar ist, wird beim Zugriff auf die Informationen keine Suche nach anderen oder alternativen Providern durchgeführt.

Die eindeutigen Schlüssel der Kontextelemente werden durch die abstrakte Klasse `ContextElementKey` und ihre Unterklassen realisiert. Bei der Erzeugung von Schlüsseln der Schnittstellen `Entity` und `Association` müssen Domänentyp, Objektbezeichner und Providerbezeichner spezifiziert werden. Statt des Domänentyps ist für die Erzeugung von `AttributeKey`-Objekten der Schlüssel des übergeordneten Elternelements anzugeben. Aufgrund dieser Abhängigkeit erzeugt ein `AttributeKey`

*Nachfolgend z. T. auch *Kontextelemente* genannt.

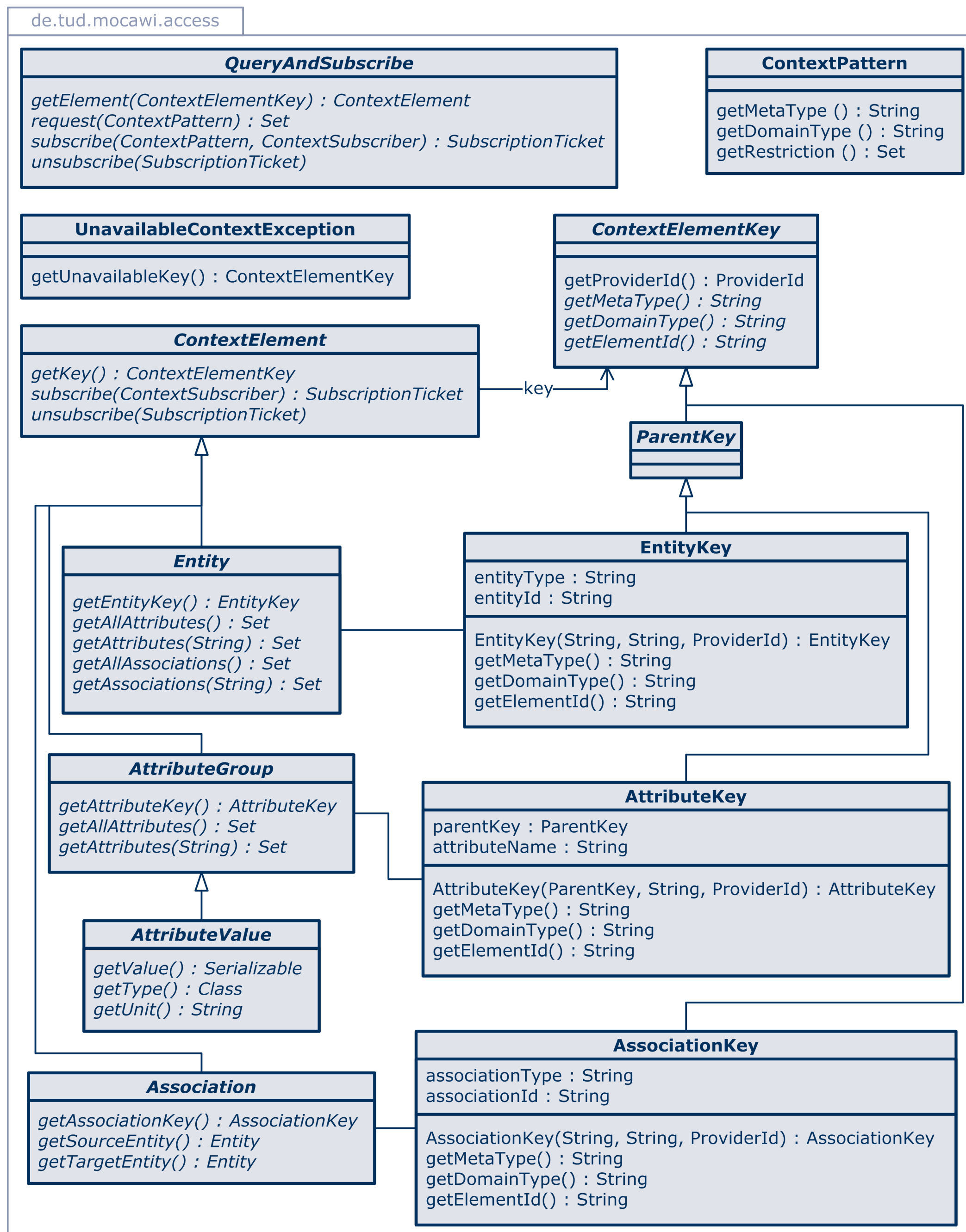

Abbildung 6.7.: Schnittstellen der Kontextelemente, sowie Klassen ihrer Schlüssel

die Rückgabewerte der Methoden `getMetaType()` und `getDomainType()` jeweils durch Aufruf derselben Methoden auf seinem übergeordneten `ContextElement` – dem Parent – und anschließende Ergänzung um den eigenen Objektbezeichner – den Attributnamen. Um die Relationen zwischen Entitäten und Attributen, sowie die Relationen zwischen Attributgruppen und Attributwerten in der Struktur ihrer Schlüssel zu realisieren, wurde zusätzlich die abstrakte Klasse `ParentKey` definiert. Weil nur `EntityKey` und `AttributeKey` Unterklassen dieser Klasse sind, können nur sie als Schlüssel des Elternelements bei der Erzeugung eines `AttributeKey` genutzt werden. Es entsteht somit eine rekursive, lineare Struktur, deren Anfang immer ein `EntityKey` bildet.

Hinweis 4. *Zur Separierung der Typen und Bezeichner des Parents und des Attributs wird in `AttributeKey` das Separator-Zeichen „.“ verwendet, welches folglich nicht in den Typen und Bezeichnern von Kontextelementen vorkommen darf. Beim Versuch dieses Zeichen zu verwenden liefern die Unterklassen von `ContextElementKey` entsprechende Fehlermeldungen.*

Die Schnittstellen der Kontextelemente werden durch eine Hierarchie von Datenobjektklassen implementiert (Abb. 6.8 auf der nächsten Seite), welche die zur Laufzeit genutzten Kontextinformationen repräsentieren. Provider können diese Objekte über die `ContextService`-Schnittstelle des Managers erzeugen indem sie die Methoden `newEntity`, `newAssociation`, `newAttributeGroup` und `newAttributeValue` aufrufen.

Consumer können die Methoden der Kontextelemente nutzen, um das zur Laufzeit bestehende Kontextmodell zu navigieren – z. B. zwischen Entitäten und Attributen, sowie zwischen mehreren Entitäten (über deren Assoziationen). Um die Navigation zu vereinfachen, können die Relationen auf Basis des Domänentyps der assoziierten Kontextelemente gefiltert werden. Die Methode `getAssociations(String)` eines `EntityDO` liefert die Untermenge der Assoziationen einer Entität, welche einen konkreten Domänentyp besitzen. Die vollständige Menge aller Assoziationen einer Entität kann mittels `getAllAssociations()` abgefragt werden. Ähnlich funktioniert die Nutzung von `getAttribute(String)`.

Wechsel von synchronem zu asynchronem Zugriff

Des weiteren unterstützen die Kontextelemente den Wechsel vom synchronen Informationszugriff zum asynchronen. Dies ist sinnvoll, wenn ein Consumer auf synchronem Weg eine Menge von Kontextinformationen abgefragt hat, und anschließend daraus durch Navigation den für ihn relevanten Teil selektiert hat, welchen er zukünftig beobachten möchte. Zu diesem Zweck bietet die `ContextElement`-Schnittstelle – analog zur `QueryAndSubscribe`-Schnittstelle – die Methoden `subscribe` und `unsubscribe` an. Subscriber können diese Methoden nutzen, um die Ereignisse eines

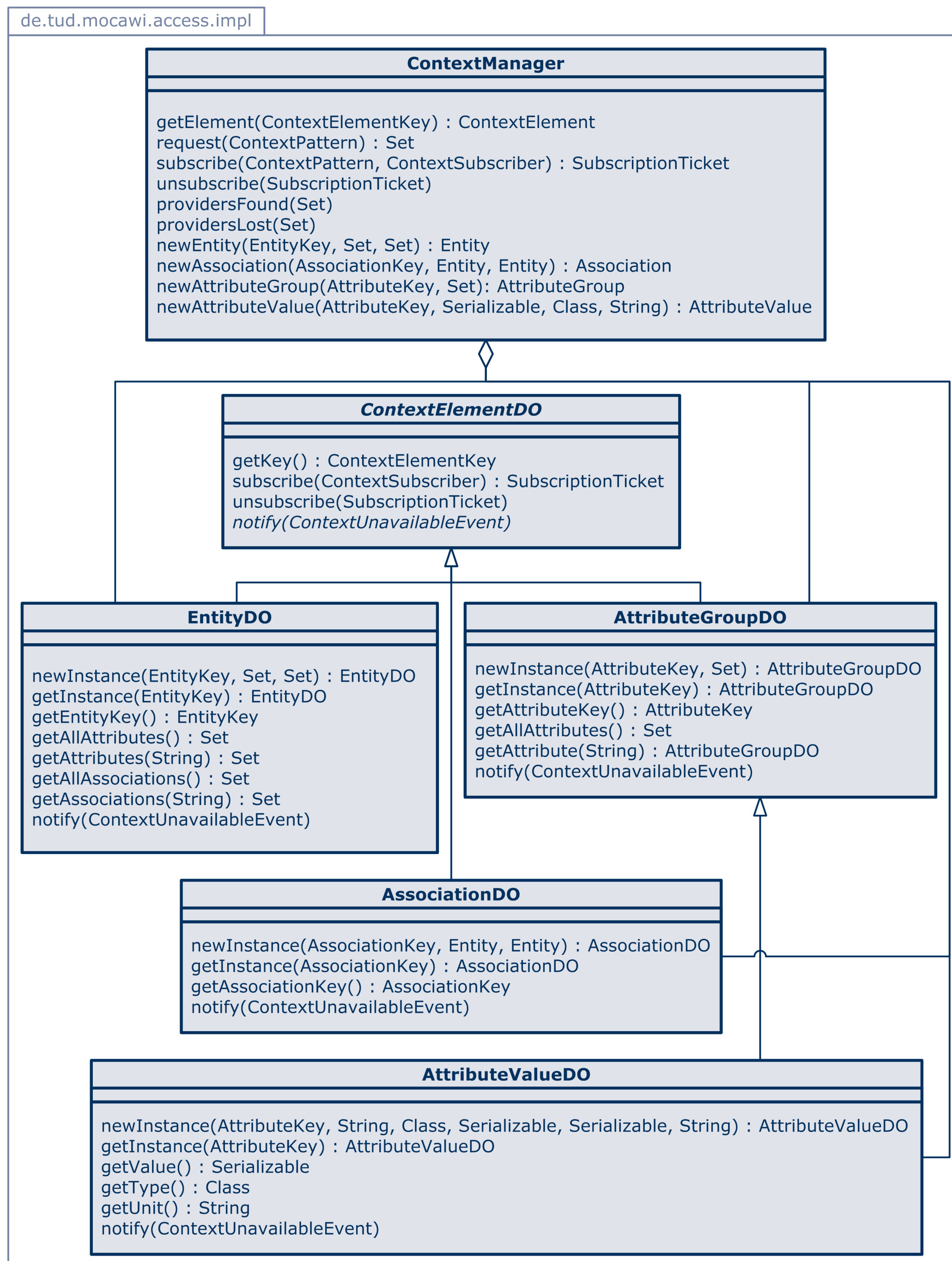

Abbildung 6.8.: Klassen der Datenobjekte der Kontextelemente

Kontextelements zu abonnieren bzw. das Abonnement zu beenden.

Dabei findet auch eine Propagierung der Ereignisse zwischen den Kontextelementen statt. `AssociationEvents` werden nicht nur an die zugehörigen `Association`-Objekte weitergeleitet, sondern auch an die `Entity`-Objekte von denen die Assoziation ausgeht. `AttributeValueEvents` werden zusätzlich zu ihren `AttributeValue`-Objekten auch an deren übergeordnete `AttributeGroup`- oder `Entity`-Objekte propagiert. Zusätzlich zur Propagierung der Kontextereignisse `AttributeEvent` und `AssociationEvent` zu den übergeordneten Kontextelementen ihrer Kontextelemente, findet bei allen `EntityEvents` und `AssociationEvents`, welche die Zerstörung ihres zugehörigen Kontextelements signalisieren, eine Propagierung zu den untergeordneten Kontextelementen statt.

Hinweis 5. *Um den asynchronen Zugriff auf einzelne Kontextelemente zu unterstützen (siehe nächster Abschnitt), müssen die Provider alle von ihnen synchron gelieferten Kontextinformationen aktualisieren indem sie deren `notify`-Methoden aufrufen.*

Hybrider Kontextzugriff

Für Consumer, welche hohe zeitliche Anforderungen an die Bedienung von synchronen Anfragen stellen (siehe Kap. 5.5.5), gibt es eine weitere Form des Kontextzugriffs. Dabei melden diese Consumer längere Zeit vor dem eigentlichen Kontextzugriff ihren Informationsbedarf an. Der Context Service sorgt daraufhin dafür, dass geeignete Provider überwacht werden, sodass bei der Durchführung des eigentlichen Zugriffs die Bedienungszeit reduziert wird.

Ein Consumer registriert sich beim Context Service durch Aufruf der Methode `QueryAndSubscribe.prepare(ContextConsumer)`. Weil die Schnittstelle der Consumer auch für die Realisierung von High-Level Providern genutzt wird, enthält sie bereits die `consumes()`-Methode mit welcher der Context Service ermitteln kann, welche Kontextinformationen der Consumer benötigt. Zusätzlich zu dieser Methode muss ein Consumer auch die `stillAlive()`-Methode unterstützen die der Context Service benutzt, um regelmäßig festzustellen, ob der Consumer noch existiert und folglich die Provider der benötigten Informationen nach wie vor überwacht werden müssen.

6.2.3. Verwaltungskomponente – `de.tud.mocawi.broker`

Damit Zugriffs- und Kommunikationskomponente ihren Consumern die Informationen lokaler Provider zur Verfügung stellen können, müssen die Provider verwaltet werden. Der `ContextBroker` erfüllt diese Aufgabe, indem er die Funktionalität erbringt, die schon zuvor in Kapitel 5.4 beschrieben wurde. Das beinhaltet in erster

Linie die Analyse der benötigten Kontextinformationen von High-Level Providern, um deren Verwendbarkeit zu ermitteln.

Von den vorgestellten Context Pattern Mengen – P_A, P_U, C_A und C_U – ist in der Praxis P_U nicht erforderlich. Zur Verwaltung der übrigen Mengen wird jeweils eine bi-direktionale Abbildung zwischen Pattern und Provider vorgenommen, weil einerseits ein Provider beliebig viele Pattern liefern bzw. konsumieren kann, und andererseits ein Pattern von beliebig vielen Providern geliefert bzw. konsumiert werden kann. Erkennbar sind diese Beziehungen durch die Relationen `providedBy` und `consumedBy` in den Abbildungen 6.5 auf Seite 161 und 6.9 auf der nächsten Seite. Realisiert wird die Abbildung durch `PatternProviderMapping` – eine interne Klasse des Brokers. Diese Klasse erleichtert auch die Analyse der Provider, weil sie beim Hinzufügen eines Providers und seiner Pattern sofort ermittelt, ob sich die Menge der verwalteten Pattern vergrößert hat, d. h. ob der Provider der erste ist, welcher mit einem bestimmten Pattern assoziiert ist. Weiterhin wird die Verkleinerung der Patternmenge beim Entfernen eines Providers und seiner Pattern ermittelt, d. h. ob ein Provider der letzte war, welcher mit einem konkreten Pattern assoziiert war.

Als Voraussetzung für die Analyse der Provider übernimmt der Broker auch die Überwachung der Provider. Sie erfolgt, wie bereits mehrfach erwähnt, durch Verwendung der `ServiceTracker`-Klasse mit einer Implementierung der `ServiceTrackerCustomizer`-Schnittstelle. Providerimplementierungen müssen ihre Providerdienste bei der OSGi Service Platform unter Angabe der Schnittstelle `LowLevelProvider` oder `HighLevelProvider` registrieren, um vom Broker gefunden zu werden.

6.2.4. Kommunikationskomponente – `de.tud.mocawi.comm`

Die Kommunikationskomponente ermöglicht die Kopplung der lokalen Context Service Instanz an entfernte Instanzen zur Erhöhung der Verfügbarkeit von Kontextinformationen. Das kann an ein paar Beispielen illustriert werden. Verfügt eine lokale Context Service Instanz über keine lokalen Provider, und somit auch keine Verwaltungskomponente (Abb. 6.4 auf Seite 159, Konfiguration C), so muss die Kommunikationskomponente den Zugriff auf entfernte Provider ermöglichen. Besitzt eine lokale Instanz dagegen keine lokalen Consumer und somit auch keine Zugriffskomponente (Abb. 6.4, Konfiguration D), so hat deren Betrieb nur dann Sinn, wenn die lokalen Kontextinformationen den Consumern entfernter Instanzen zur Verfügung gestellt werden. Während die Kommunikationskomponente im ersten Beispiel die Rollen der Verwaltungskomponente und der Provider übernimmt, sind es im zweiten die Rollen der Consumer und der Zugriffskomponente. Selbst wenn lokale Consumer und Provider vorhanden sind, führt die Kopplung mit anderen Context Service Instanzen zu einer Vergrößerung der verfügbaren Kontextinformationen.

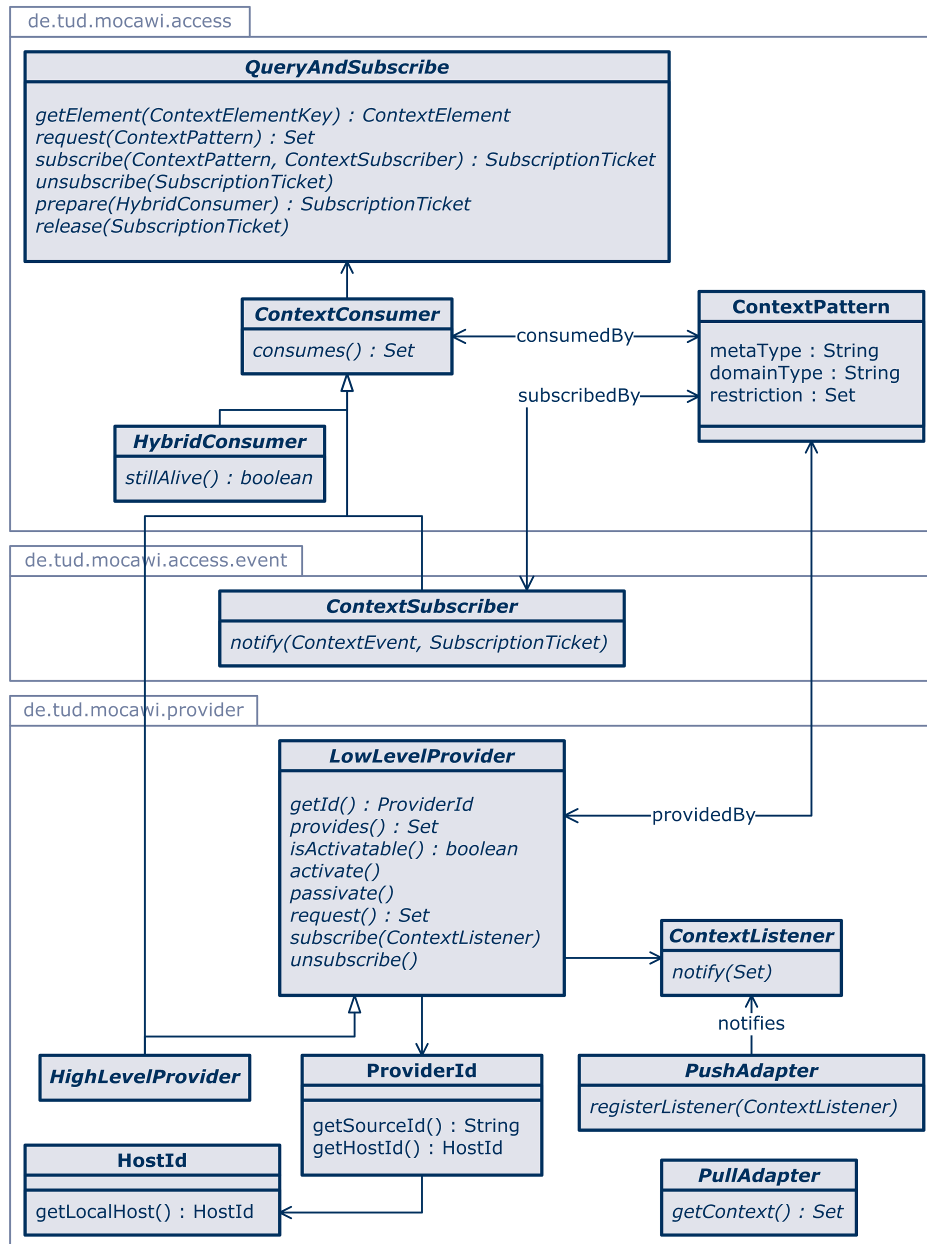

Abbildung 6.9.: Schnittstellen und Klassen zur Implementierung von Context Consumern und Context Providern

Emulation der Consumer

Weil Consumer nur über die Zugriffskomponente auf Kontext zugreifen, sind sie durch diese vollständig gekapselt und vor dem Rest des Systems verborgen. Folglich muss die Kommunikationskomponente keine entfernten Consumer nachbilden, sondern nur die Zugriffskomponenten der entfernten Context Service Instanzen.

Emulation der Zugriffskomponente

Die Zugriffskomponente wird sowohl von Consumern genutzt, um auf Kontext zuzugreifen, als auch von Providern, um die Objekte für die Datenhaltung der Kontextinformationen zu erzeugen. Ist keine lokale Zugriffskomponente vorhanden, unterstützt die Kommunikationskomponente neben dem Kontextzugriff auch die Erzeugung entsprechender Objekte.

Stellvertretend für den Manager der Zugriffskomponente implementiert die Kommunikationskomponente die Schnittstellen `ContextService` und `ContextListener`. Darüber hinaus verwaltet sie eine Menge interner, virtueller Consumer, welche die Übertragung der Anfrageergebnisse und Benachrichtigungen an die entfernten Consumer bzw. die entfernten Zugriffskomponenten übernehmen.

Emulation der Verwaltungskomponente

Der einzige Klient der Verwaltungskomponente ist die Zugriffskomponente, die geeignete Provider für die Bedienung von Anfragen und Abonnements selektieren muss. Die Kommunikationskomponente bietet stellvertretend die Provider aller erreichbaren, entfernten Verwaltungskomponenten an. Für eine nahtlose Integration implementiert die Kommunikationskomponente die `ContextBroker`-Schnittstelle.

Emulation der Provider

Im Gegensatz zu einem Consumer wird ein Provider durch die Verwaltungskomponente nicht vollständig gekapselt, weil die Zugriffskomponente die Verwaltungskomponente nur für die Selektion geeigneter Provider verwendet. Der Zugriff auf die Kontextinformationen erfolgt direkt durch Aufruf der selektierten Provider. Folglich bildet die Kommunikationskomponente auch Provider nach, indem sie bedarfsabhängig Stellvertreter für entfernte Provider bereitstellt, welche die Kopplung zu den eigentlichen Providern vornehmen.

Realisierung der Kommunikation

Die Kommunikationskomponente realisiert die zentrale Verwaltung mit hybridem Austausch der Providerbeschreibungen (Kap. 5.5.5). Eine weitere Version der Kommunikationskomponente, welche eine dezentrale Verwaltung gestattet, befindet sich

derzeit noch in Entwicklung. Sie nutzt Mechanismen des P2P-Frameworks JXTA zur automatischen Suche von Peers, zur Verwaltung des P2P-Netzwerkes und zum Nachrichtenaustausch zwischen einzelnen Peers. Aufgrund technischer Probleme und teilweise indeterministischen Verhaltens dieser Funktionen sind evaluierende Test zum aktuellen Zeitpunkt nicht möglich.

Sowohl zwischen den Clients und dem Server, als auch zwischen den Clients selbst, wird Socketkommunikation verwendet. Der *IO Connector Service* der OSGi Service Platform ermöglicht entsprechende Verbindungen unter Verwendung des *Generic Connection Frameworks* der Java ME Laufzeitumgebung. Die Nutzdaten der ausgetauschten Nachrichten liegen, wie bereits beschrieben, in XML-Format vor, welches mit Hilfe des *XML Parser Service* der OSGi Service Platform verarbeitet wird. Die Adresse des Servers ist zur Laufzeit statisch definiert, und wird aus einer Konfigurationsdatei im `de.tud.mocawi.comm`-Bundle des ausgelesen. Die Adressen der übrigen Clients erfährt ein Client implizit aus den Nachrichten des Servers.

Für alle drei nachzubildenden Komponententypen – Zugriffs- und Verwaltungskomponente, sowie Provider – werden Stellvertreter eingesetzt, die Rückaufrufe unterstützen und deshalb aus zwei Teilen bestehen. Der erste (Klient-lokale) Teil bietet die Schnittstelle der entfernten Komponente an, und leitet synchrone Aufrufe an diese, über den zweiten (entfernten) Teil, weiter. Greift der erste Teil asynchron auf Kontextinformationen oder verfügbare Provider zu, so wird neben der Weiterleitung an den zweiten Teil dieser selbst bei der entfernten Komponente für Benachrichtigungen registriert. Finden diese statt, so leitet er sie an den ersten Teil weiter, welcher sie an die lokalen Klienten ausliefern kann.

6.3. Implementierung der Context Provider

Wie schon mehrfach betont, sind Context Provider nicht Teil des Context Service, sondern stellen selbstständige Systeme dar. Folglich nimmt der Context Service lediglich ihre Verwaltung vor, um den Zugriff auf sie zu kontrollieren. Er greift dabei nur teilweise in ihren Lebenszyklus (siehe Kap. 5.4.3) ein – die meisten Zustandsänderungen ergeben sich aus den Charakteristika der Provider oder werden durch sie selbst beeinflusst. So entsprechen die Zustandsübergänge „starten" und „stoppen" (Abb. 5.8 auf Seite 106) der Registrierung und der Deregistrierung der Schnittstellen einer Providerimplementierung mit der OSGi Service Platform. Nur registrierte Provider sind für den Context Service auffindbar/sichtbar. Die Übergänge zu und zwischen den Zuständen „verwendbar" und „nicht verwendbar" ergeben sich aus den Informationsabhängigkeiten der Provider und ihrer Erfüllbarkeit, d. h. sie sind abhängig von der globalen Situation des gesamten Systems. Lediglich die Übergänge „aktivieren" und „de-aktivieren" werden durch den Context Manager kontrolliert, um die Provider bedarfsabhängig zu aktivieren, oder in Zeiten der Inaktivität zum

Zwecke der Ressourceneinsparung zu passivieren.

6.3.1. Aufgaben eines Providers

Eine Aufgabe einer Providerimplementierung wurde soeben aufgezeigt – die Registrierung und Deregistrierung des Providerdienstes in Abhängigkeit von der Erreichbarkeit/Verfügbarkeit der zugrunde liegenden Kontextquelle. Diese Aufgabe kann bei Low-Level Providern größeren Aufwand bergen, weil die Kontextquelle meist erst gefunden werden muss. Auch bei High-Level Providern können – neben den Informationsabhängigkeiten – technische Abhängigkeiten existieren, die Einfluss auf die Registrierung haben. Erfolgt z. B. die Ableitung der abstrakten Kontextinformationen in einem High-Level Provider durch einen logikbasierten Reasoner, so kann der Providerdienst nur dann registriert werden, wenn ein solcher Reasoner auch wirklich vorhanden ist. Aufgrund der Größe derartiger Systeme ist es unwahrscheinlich, dass der Reasoner Teil des Providers ist. Statt dessen greift der Provider lediglich auf einen verfügbaren Reasoner zu.

Die Registrierung kann zu unterschiedlichen Zeitpunkten erfolgen. Bestehen keine Abhängigkeiten eines Providers von weiteren Ressourcen (z. B. Kontextquellen, Diensten, etc.), so kann er sofort beim Start seines übergeordneten Bundles in der `start()`-Methode des so genannten `BundleActivator`s registriert werden. Dabei wird die Providerimplementierung initialisiert und anschließend unter Angabe der Klassennamen der implementierten Schnittstellen dem `BundleContext` übergeben (siehe Quellcode 6.1, Zeile 6). Der `BundleContext` stellt die Schnittstelle eines Bundles zur OSGi Service Platform dar. Da der Context Service keinen Gebrauch von Domänen- oder Provider-spezifischen Schnittstellen machen kann, genügt die Angabe der Schnittstellen `LowLevelProvider` und `HighLevelProvider` – je nach Art des Providers.

Hinweis 6. *Bei Providern die von Ressourcen (z. B. Quellen) abhängig sind, sollte in der `start()`-Methode lediglich die Suche und Überwachung der Ressourcen gestartet werden. Die Registrierung ihrer Providerdienste sollte jedoch situationsabhängig beim Auffinden der Ressourcen vorgenommen werden können.*

Quelltext 6.1: Registrierung eines High-Level Providers

```
1  public class ProviderActivator implements BundleActivator {

     public void start(BundleContext context) throws Exception↘
     → {
       LowLevelProvider pro = new ProviderImplementation();
5      String[] clazzes = new String[] {LowLevelProvider.class↘
       →.getName()};
```

```
      context.registerService(clazzes, pro, null);
   }

   public void stop(BundleContext context) throws Exception ↘
   →{
10 }
}
```

Es gibt keine Beschränkungen hinsichtlich der Anzahl von Providerdiensten, welche durch ein einzelnes Bundle bereitgestellt und entsprechend registriert werden können. Die Entscheidung über eine geeignete Granularität bei der Implementierung der Provider liegt vollständig bei den verantwortlichen Entwicklern. Es sind jedoch die Rahmenbedingungen bei der Implementierung von OSGi-basierten Anwendungen zu berücksichtigen [OSG07a].

Ein Providerdienst wird deregistriert, indem die Methode `unregister()` auf dem `ServiceRegistration`-Objekt aufgerufen wird, welches als Ergebnis der Registrierung zurückgeliefert wurde. Bei abhängigen Providern muss die Deregistrierung natürlich an die Verfügbarkeit der Ressourcen gekoppelt sein. Für Provider ohne Ressourcenabhängigkeiten macht eine Deregistrierung nur beim Stopp ihres übergeordneten Bundles Sinn. In diesem Fall sind aber keine Maßnahmen von Seiten des Bundles erforderlich, weil die OSGi Service Platform automatisch beim Stopp eines Bundles sämtliche seiner registrierten Dienste – also auch die Providerdienste – deregistriert.

Neben der automatischen Registrierung bzw. Deregistrierung seiner Schnittstelle muss jeder Provider in der Lage sein, sich selbst zu beschreiben, Kontextinformationen aus seiner zugrunde liegenden Kontextquelle bzw. seinem Ableitungsschema zu extrahieren, und diese dem Context Service zur Verfügung zu stellen. Die registrierten Schnittstellen liefern all diese Informationen bzw. unterstützen die erforderliche Funktionalität. Ihre Implementierung ist abhängig von der Technologie der Kontextquellen oder Ableitungsschemata. Welche Schnittstellen ein Provider dem Context Service anbieten kann, und ihre Funktionen erläutert der nächste Abschnitt.

6.3.2. Externe Schnittstellen

Um die Integration der Provider mit dem restlichen System zu erlauben, existieren mehrere Schnittstellen, welche von einer Providerimplementierung unterstützt und von den übrigen Komponenten genutzt werden müssen. So muss in jedem Fall die `LowLevelProvider`-Schnittstelle implementiert werden, und je nach Art des Providers auch die Schnittstellen `HighLevelProvider` und `ContextSubscriber` (siehe Abb. 6.9 auf Seite 170). Diese Schnittstellen sind in den Paketen `de.tud.mocawi.provider` und `de.tud.mocawi.access.event` definiert.

Die Pakete `de.tud.mocawi.provider` und `de.tud.mocawi.access` enthalten weiterhin die Klassen `ProviderId`, `HostId` und `ContextPattern` die für die Implementierung der Schnittstellen erforderlich sind, und nachfolgend in Kapitel 6.3.3 erläutert werden. Die Schnittstelle `ContextConsumer` fasst alle Arten von Context Consumern zusammen – synchrone und asynchrone, sowie Anwendungen und High-Level Provider – ist jedoch vorwiegend für letztere interessant, weil diese die Methode `consumes(Set)` implementieren müssen. Consumer, welche synchron auf Kontext zugreifen, erreichen dies durch aktiven Aufruf der Schnittstellen des Context Service. Deshalb ist keine Schnittstellenunterstützung auf Seiten dieser Consumer notwendig. Für Subscriber hingegen existiert die Schnittstelle `ContextSubscriber`, um die Benachrichtigung über Kontextänderungen zu ermöglichen.

de.tud.mocawi.provider.LowLevelProvider

Bei der Realisierung eines Low-Level Providers ist allein die Schnittstelle `LowLevelProvider` zu implementieren. Als Teil dessen ist für jeden Provider ein global eindeutiger Bezeichner zu erzeugen, welcher als Ergebnis der Methode `getId()` zurück geliefert wird. Welche Möglichkeiten es gibt, eine solche `ProviderId` zu erzeugen, wird in Kapitel 6.3.3 erklärt.

Um während der Bearbeitung der Anfragen und Abonnements der Context Consumer geeignete Context Provider binden zu können, muss der Context Service von jedem Provider PRO_i die Menge seiner lieferbaren Kontextinformationen wissen. Diese Menge P_i (siehe Kap. 5.4.1) muss ein Low-Level Provider bei Aufruf der Methode `provides()` in Form einer Menge (`java.util.Set`) zurück liefern, welche entsprechende Instanzen der Klasse `ContextPattern` enthält. Wie schon in Kapitel 5.3 beschrieben, besteht ein solches Pattern aus einem Metatyp, einem Domänentyp und einer Restriktion.

Zur Einsparung von Geräteressourcen sollen Provider bedarfsabhängig vor Ausführung von Anfragen und Abonnements aktiviert und nach deren Beendigung passiviert werden. Die Providerimplementierungen müssen während dieser Phasen Ressourcen allokieren bzw. freigeben. Damit der Context Service die Provider überhaupt derartig steuern kann, müssen sie geeignete Methoden implementieren. Es kann jedoch nicht vorausgesetzt werden, dass alle Provider im Bedarfsfall in der Lage sind, innerhalb kürzester Zeit ihre Ressourcen zu allokieren oder ihre Informationen bereitzustellen. So müssen beispielsweise High-Level Provider* die langfristig Kontextinformationen verarbeiten, um daraus gleitende Mittelwerte zu gewinnen, dauerhaft aktiviert sein. Aus diesen Gründen müssen Provider als Ergebnis der Methode `isActivatable()` mitteilen, ob sie dynamisch aktivierbar und passivierbar

*Bei der Beschreibung der `HighLevelProvider`-Schnittstelle werden die Gemeinsamkeiten von Low- und High-Level Providern gezeigt.

sind (`true`), oder ob sie dauerhaft aktiviert bleiben müssen (`false`). Im Lebenszyklus dieser letzten Gruppe von Providern verschmelzen die Zustände „verwendbar" und „aktiviert" (Abb. 5.8 auf Seite 106). Dynamisch aktivierbare Provider müssen bei Aufruf der Methode `activate()` in den Zustand „aktiviert" wechseln und bei Aufruf von `passivate()` wieder in den Zustand „verwendbar" zurückkehren.

Während der Bedienung synchroner Anfragen von Consumern muss der Context Service die aktuell gültigen Kontextinformationen von den ausgewählten Providern abfragen können. Die Providerimplementierungen müssen dies durch die Methode `request()` unterstützen, als deren Ergebnis sie die Menge (`java.util.Set`) aller aktuell von ihnen lieferbaren Kontextinformationen bereitstellen. Die Menge enthält Instanzen der Klassen `EntityDO`, `AttributeValueDO`, `AttributeGroupDO` und `AssociationDO` des Paketes `de.tud.mocawi.contextmodel`, welche durch die Providerimplementierung zu erzeugen sind. Dazu stellen diese Klassen entsprechende Fabrikmethoden zur Verfügung (Kap. 6.2.2).

Hinweis 7. *Die Ergebnismenge der `request()`-Methode muss alle Kontextinformationen enthalten, die in der Methode `provides()` als lieferbar angegeben wurden. Provider bei denen das nicht der Fall ist, entfernt der Context Service aus der Verwaltung, weil sie sich nicht konform zu ihrer Beschreibung verhalten.*

Abonnieren Subscriber Kontextinformationen beim Context Service, so führt dieser stellvertretend Abonnements bei geeigneten Providern durch. Dazu ruft er die Methode `subscribe(ContextListener)` auf, und registriert sich selbst bei den Providern, um Benachrichtigungen über geänderte Kontextinformationen zu erhalten. Weil die Kontextinformationen nicht direkt an die Subscriber ausgeliefert werden, ist der Context Service der einzig mögliche Abonnent. Somit müssen Provider zu jeder Zeit nur höchstens einen `ContextListener` benachrichtigen. Deshalb ist auch zur Beendigung eines Abonnements per `unsubscribe()` kein Argument erforderlich. Die Benachrichtigung über Kontextänderungen muss der Provider durch Aufruf von `notify(Set)` auf der `ContextListener` Schnittstelle vornehmen. Analog zur Ergebnismenge von `request()` müssen dabei die veränderten Kontextinformationen in Form einer Menge von Instanzen der Unterklassen von `ContextElementDO` übergeben werden. Diese Schnittstelle wird vom Context Service implementiert.

Hinweis 8. *Die Providerimplementierungen tragen dafür Sorge, dass direkt nach der Registrierung eines Abonnements eine erste Benachrichtigung erfolgt, in der die gleiche Menge von Kontextinformationen übertragen werden muss, wie sie beim zeitgleichen Aufruf von `request()` geliefert worden wäre.*

Dieses Verhalten ist notwendig, weil nicht vorausgesetzt werden kann, dass Abonnenten zum Zeitpunkt der Durchführung des Abonnements bereits den aktuellen Zustand des abonnierten Kontexts kennen, und somit bis zur nächsten Änderung

dieses Kontexts keinerlei Informationen über ihn besäßen. Aufgrund dieser Form der Synchronisation müssen nachfolgende Benachrichtigungen im Rahmen derselben Abonnements lediglich die Menge derjenigen Kontextinformationen enthalten, welche sich tatsächlich seit der letzten Benachrichtigung geändert haben.

Je nach Art der zugrunde liegenden Kontextquelle müssen bei der Realisierung von Low-Level Providern diejenigen Zugriffsformen emuliert werden (siehe Kap. 5.2.1), welche die Quelle nicht unterstützt. Dazu kann der Entwickler eines Providers auf die vorgefertigten Bausteine (`PullWorker`, `PushFilter`, `Comparator` und `Cache`) des Provider Frameworks zurückgreifen. Die ohnehin erforderliche Behandlung der Technologieabhängigkeit der Quellen geschieht durch Implementierung der Schnittstellen `PullAdapter` oder `PushAdapter`.

Quelltext 6.2 zeigt die beispielhafte Implementierung eines Low-Level Providers der einen Temperatursensor kapselt. Die Registrierung des Providerdienstes durch den Activator wurde bereits zuvor präsentiert. Teil dieser Registrierung ist die Instanziierung der Providerimplementierung durch Aufruf des Konstruktors der Beispielklasse `ProviderImplementation` (Zeilen 8 bis 13). In dieser Phase sollten die Objekte erzeugt werden, welche die statische Beschreibung des Providers repräsentieren. Über die Methoden `getId()`, `provides()` und `isActivatable()` (Zeilen 15 bis 25) stehen sie dem Context Service zur Verfügung. Aktiviert der Context Service den Provider durch Aufruf von `activate()` (Zeilen 27 bis 31) , dann sollte er benötigte Ressourcen allokieren – hier: Initialisierung einer `PullAdapter`-Implementierung zur Kapselung der Quelle und eines `PullWorker`s zur Emulation des Push-Zugriffs. Bei der Deaktivierung durch den Aufruf der `passivate()`-Methode (Zeilen 33 bis 37) werden diese Komponenten wieder beendet. Die Realisierung des Kontextzugriffs in den Methoden `request`, `subscribe` und `unsubscribe` erfolgt durch einfache Delegation der Aufrufe an die internen Komponenten des Providers.

Quelltext 6.2: Implementierung eines Low-Level Providers

```
1  public class ProviderImplementation implements ↘
   →LowLevelProvider {
     private ProviderId myId = null;
     private Set iProvide = null;
     private Properties = null;
5    private PullAdapter myAdapter = null;
     private PullWorker myWorker = null;

     public ProviderImplementation(Properties sourceFeatures) ↘
     →{
       this.myId = ProviderId.newId(HostId.getLocalHost());
10     this.iProvide = new HashSet();
       this.iProvide.add(new ContextPattern("AT", "room.↘
```

```
  →temperature", "room.id='INF3100'"));
    this.mySource = sourceFeatures;
  }

15  public ProviderId getId() {
    return this.myId;
  }

  public Set provides() {
20    return this.iProvide;
  }

  public boolean isActivatable() {
    return true;
25  }

  public synchronized void activate() {
    // intialize pull adapter and worker
    this.myAdapter = new TemperatureAdapter(this.mySource);
30    this.myWorker = new PullWorker(this.myAdapter);
  }

  public synchronized void passivate() {
    // free resources
35    this.myAdapter = null;
    this.myWorker = null;
  }

  public synchronized Set request() {
40    if (this.myAdapter != null) {
      // delegate call to pull adapter
      return this.myAdapter.getContext();
    }
  }
45
  public synchronized void subscribe(ContextListener ↘
  →listener) {
    if (this.myWorker != null) {
      // delegate call to pull worker
      this.myWorker.subscribe(listener);
50    }
```

```
  }

  public synchronized void unsubscribe() {
    if (this.myWorker != null) {
55    // delegate call to pull worker
      this.myWorker.unsubscribe();
    }
  }

60 // the adapter to the context source
  private class TemperatureAdapter implements PullAdapter {
    public TemperatureAdapter(Properties sourceFeatures) {
      ...
    }
65 }
}
```

Aufgrund der Komplexität und Technologieabhängigkeit kann hier nicht gezeigt werden, wie die `PullAdapter`-Implementierung auf den Sensor zugreift. Bei der Realisierung dieses Zugriffs können die Entwickler auch Funktionen der OSGi Service Platform nutzen, wie z. B. den *Device Access* der eine einheitliche Schnittstelle auf die Gerätehardware definiert. Diese Schnittstelle wird für verschiedene Technologien bereits unterstützt.

Die Vorgehensweise zur Erzeugung der Datenobjekte für die Repräsentation der Kontextinformationen ist Kapitel 6.2.2 zu entnehmen. Außerdem muss der Context Service jederzeit in der Lage sein, auftretende Fehler oder Ausnahmen zu behandeln, weil beim Zugriff des Adapters auf die Quelle und auch innerhalb der Providerimplementierung Fehler auftreten können. Dies zieht einerseits die Entfernung des fehlerhaften Providers aus der Verwaltung, und andererseits die Suche alternativer Provider, sowie ggf. die Erzeugung von Fehlermeldungen nach sich.

Hinweis 9. *Providerentwickler sollten die Providerimplementierung derart vornehmen, dass auftretende Fehler und Ausnahmen beim Zugriff auf die Kontextquelle weitestgehend Provider-intern behandelt werden. Nur so kann langfristige Verwendbarkeit des Providers gewährleistet werden, weil im Context Service aufgrund der Technologieabhängigkeit des Quellenzugriffs keine Fehlerbehandlung möglich ist.*

Hinweis 10. *Provider sollten derart implementiert sein, dass sie entweder fehlerfreies Verhalten des Context Service voraussetzen, oder fehlerhaftes Verhalten ignorieren – z. B. Zugriff auf Kontext im de-aktivierten/passivierten Zustand. Entsprechend ausgelöste Ausnahmen* würden vom Context Service als Fehler der Provider*

*Exceptions

fehlinterpretiert, was zu einer Aussonderung der betroffenen Provider führt.

de.tud.mocawi.provider.HighLevelProvider

Kapitel 5.4.1 hat gezeigt, dass eine einheitliche Beschreibung von Low- und High-Level Providern möglich ist (Gleichung 5.5 auf Seite 101). Alle vom Broker als verwendbar bewerteten High-Level Provider können folglich wie Low-Level Provider genutzt werden. Zusätzlich muss der Context Service jedoch ihre Informationsabhängigkeiten erfahren können. Deshalb erbt die `HighLevelProvider`-Schnittstelle nicht nur von der Schnittstelle `LowLevelProvider`, sondern zusätzlich auch von `ContextConsumer`.

Die Informationen über die von High-Level Providern benötigten Kontextinformationen müssen von ihren Implementierungen als Ergebnis der Methode `consumes()` geliefert werden. Analog zu `LowLevelProvider.provides()` ist das Ergebnis eines solchen Aufrufs eine Menge von Context Patterns – in diesem Fall C_i. Weil Context Patterns keinerlei Informationen über den zeitlichen Charakter des Kontextzugriffs (synchron oder asynchron) enthalten, kann der Context Service nur die Abhängigkeiten der High-Level Provider erkennen und ggf. behandeln. Er ist folglich nicht in der Lage, selbstständig die benötigten Kontextinformationen an die High-Level Provider zu liefern.

Hinweis 11. *Die Implementierungen von High-Level Providern müssen selbstständig auf Kontextinformationen zugreifen, weil die Art und Weise der Kontextnutzung zur Ableitung abstrakter Kontextinformationen sehr stark vom Ableitungsschema und eventuell von der Anwendungsdomäne abhängig sein kann. Somit können nur die Entwickler der High-Level Provider entscheiden, welche Zugriffsform in welcher Situation bzw. für welche Funktion geeignet ist.*

Müssen Provider asynchron oder hybrid auf Kontext zugreifen, so sind dafür zusätzlich die Schnittstellen `ContextSubscriber` und `HybridConsumer` zu implementieren, welche in den nächsten Abschnitten beschrieben werden.

de.tud.mocawi.access.event.ContextSubscriber

Neben den adaptiven, kontextbezogenen Anwendungskomponenten müssen auch manche High-Level Provider asynchron auf Kontextinformationen zugreifen. Dabei bestehen dieselben Anforderungen:

- Möglichkeit der Spezifikation der benötigten Kontextinformation
- Änderungsabhängige Benachrichtigung
- Unterstützung mehrerer Abonnements pro Provider

- Orts- und quellentransparenter Zugriff auf Kontextinformationen

Deshalb werden diese High-Level Provider wie Subscriber behandelt und müssen die `ContextSubscriber`-Schnittstelle implementieren, um sich bei der `QueryAndSubscribe`-Schnittstelle für die benötigten Kontextinformationen registrieren zu können. Was bei der Implementierung dieser Schnittstelle zu beachten ist, wird in Kapitel 6.4 erläutert.

de.tud.mocawi.access.HybridConsumer

Stellen High-Level Provider besonders hohe Anforderungen bezüglich der Antwortzeit bei der Bedienung synchroner Anfragen, so müssen sie die Schnittstelle `HybridConsumer` implementieren. Sie müssen dabei – wie alle Consumer – die Menge der von ihnen voraussichtlich konsumierten Kontextinformationen als Ergebnis der `consumes()`-Methode spezifizieren. Weiterhin müssen sie in der Implementierung der `stillAlive()`-Methode signalisieren, ob sie nach wie vor aktiv sind, und die angegebenen Kontextinformationen benötigen. Des weiteren muss die Implementierung der Schnittstelle durch die `prepare(HybridConsumer)`-Methode bei der `QueryAndSubscribe`-Schnittstelle registriert werden, um von der Beschleunigung der Selektion Gebrauch zu machen.

Der Manager ruft `consumes()` auf, um die benötigten Kontextinformationen zu ermitteln und entsprechend die Überwachung von Provider zu veranlassen. In regelmäßigen Abständen wird `stillAlive()` durch den Manager aufgerufen, sodass die Überwachung der Provider gestoppt werden kann, sobald der Consumer nicht mehr aktiv ist und den angeforderten Kontext nicht mehr benötigt.

Weil die Überwachung der Provider für das System kostspielig ist, ist es ratsam, die Granularität der Consumer möglichst klein zu wählen. Außerdem sollte die Registrierung rechtzeitig, d. h. wenn möglich wesentlich $\gg t_{RTT_{avg}}$ vor dem eigentlichen Kontextzugriff erfolgen, und sollte sobald wie möglich nach diesem durch Aufruf der `release(SubscriptionTicket)`-Methode der `QueryAndSubscribe`-Schnittstelle zurück genommen werden.

6.3.3. Weitere Klassen

Für die Implementierung der zuvor beschriebenen Schnittstellen werden zusätzliche Klassen benötigt, welche die Repräsentationen von Bezeichnern und Context Patterns gestatten. Diese Klassen werden anschließend vorgestellt. Die Klassen zur Repräsentation von Kontextinformationen sind bereits Gegenstand von Kapitel 6.2.2.

de.tud.mocawi.provider.ProviderId

Bei der Beschreibung der `LowLevelProvider`-Schnittstelle wurde gefordert, dass sich jeder Provider durch einen global eindeutigen Bezeichner identifizieren muss. Jedoch wurde nicht beschrieben, wie dieser zu wählen ist.

Aufgrund der unterschiedlichen Kontextquellen gibt es auch unterschiedliche Beziehungen zwischen einer Quelle und dem konkreten Gerät auf dem ihr Provider betrieben wird. Quellen die *lokal* auf einem Gerät vorhanden sind, sind nur *geräteabhängig* erreichbar (z. B. Quellen von Provider P und S, Abb. 5.1 auf Seite 90). Kann ein anderes Gerät keine Verbindung zu einem Gerät herstellen, so kann es auch dessen Quellen bzw. Provider nicht nutzen. Quellen die *entfernt* von einem Gerät existieren, können unterschiedlich erreichbar sein. Dabei gibt es Quellen, welche nur unter bestimmten Bedingungen – z. B. im Empfangsbereich der Bluetoothkommunikationseinrichtung des Gerätes – erreichbar sind (z. B. Quelle von Provider Q, Abb. 5.1). Andere sind dagegen nahezu immer erreichbar – z. B. ein öffentlicher Web Service im Internet (Quelle der Provider R und T, Abb. 5.1). Wesentlich ist, dass *entfernte* Quellen *geräteunabhängig* erreichbar sein können. Das gilt vor allem für *globale* Quellen.

Um bei der Konstruktion des Providerbezeichners (der `ProviderId`) – *geräteabhängige* und *-unabhängige* Quellen zu unterstützen, besteht eine `ProviderId` aus einem Quellen- und einem Gerätebezeichner (der `HostId`). Während der Gerätebezeichner ein systemspezifisches Format hat, ist der Quellenbezeichner eine Zeichenkette, welche beliebig bzw. Domänen-spezifisch definiert werden kann.

Die `ProviderId` von Providern die lokale Quellen kapseln, muss eine gültige `HostId` enthalten. Um die Implementierung solcher Provider zu erleichtern, existiert die Methode `HostId.getLocalHost()`, welche die `HostId` des lokalen Gerätes liefert. Sie sollte bei der Erzeugung einer *lokalen* `ProviderId` genutzt werden. Die Entscheidung, ob ein Provider lokale oder entfernte Quellen kapselt, ist von den Entwicklern des Providers zu treffen. Meist ist diese Entscheidung jedoch durch die Art der Quelle und somit durch die Anwendungsdomäne vorgegeben.

Der Quellenbezeichner muss in jedem Fall – egal ob lokale oder entfernte Quelle – Teil der `ProviderId` sein. Es gibt unterschiedliche Wege den Quellenbezeichner zu definieren: entweder manuell durch statische Definition vom Providerentwickler bzw. Konfiguration vom Provideradministrator beim Deployment, oder automatisch durch die Providerimplementierung.

Hinweis 12. *Die automatische Vergabe der Quellenbezeichner durch die Providerimplementierung ist zu bevorzugen, wenn eine Providerinstanz wechselnde Instanzen des gleichen Quellentyps kapselt. Das gilt primär für Kontextquellen die entfernt angebunden oder sogar global erreichbar sind – z. B. Bluetooth-GPS-Empfänger bzw. Web Service.*

Bei der Registrierung eines Providers mit der OSGi Service Platform ist eine gültige `ProviderId` erforderlich. Folglich muss auch der Quellenbezeichner vor der Registrierung definiert sein. Quellenbezeichner entfernter Quellen müssen global eindeutig sein. Die Bezeichner lokaler Quellen hingegen müssen nur lokal eindeutig sein. Diese Eigenschaft wird in der Implementierung der `ProviderId` durch Verwaltung aller lokal vergebenen `ProviderId`s in einer Tabelle gesichert, welche mit geringem Aufwand ($\mathcal{O}(log\ n)$) durchsucht werden kann. Zur Erzeugung der `ProviderId`s gibt es folgende Versionen der Methode `newId` für die unterschiedlichen Anwendungsfälle:

- `newId()` liefert eine `ProviderId` für eine globale Quelle deren Quellenbezeichner generiert ist.
- `newId(String)` liefert eine `ProviderId` für eine globale Quelle mit einem extern definierten Quellenbezeichner. Doppelte Vergabe von `ProviderId`s wird auf lokaler Ebene verhindert, sodass die Wiederverwendung eines Quellenbezeichners zu einer `java.lang.IllegalArgumentException` führt.
- `newId(HostId)` liefert eine `ProviderId` für eine lokale Quelle deren Quellenbezeichner ebenfalls generiert ist. Fehlerhafte `HostId`s werden mit einer `java.lang.IllegalArgumentException` zurück gewiesen.
- `newId(String, HostId)` liefert eine `ProviderId` für eine beliebige Quelle ohne den Quellenbezeichner zu generieren. Leere Zeichenketten für Quellenbezeichner werden mit einer `java.lang.IllegalArgumentException` zurück gewiesen.

Globale Eindeutigkeit einer `ProviderId` bzw. eines Quellenbezeichners kann ohne eine zentrale bzw. globale Instanz nicht realisiert werden. Diese Instanz könnte einerseits physisch existieren – z. B. in Form eines Knotens, welcher alle Bezeichner verwaltet und neue vergibt. Andererseits könnte sie durch ein globales Schema etabliert werden, welches den Namensraum auf verschiedene Instanzen aufteilt, die diese Teile verwalten. Die Schwierigkeit besteht bei der zweiten Möglichkeit jedoch wieder in der Zuweisung der Namensräume, welche durch eine zentrale Instanz erfolgen muss.

Im Context Service wird der zweite Ansatz für die Bezeichner lokaler Quellen genutzt. Dies geschieht derart, dass lokal eindeutige Quellenbezeichner mit der `HostId` zu einer global eindeutigen `ProviderId` kombiniert werden. Durch die lokale Verwaltung aller `ProviderId`s kann auch für entfernte Quellen lokale Eindeutigkeit ihrer Bezeichner gesichert werden. Jedoch ist bei der Generierung ihrer Quellenbezeichner in `ProviderId.newId()`, mangels Zugriffs auf eine zentrale Instanz, keine Prüfung der globalen Eindeutigkeit möglich. Um die Robustheit der einzelnen Context Service Instanzen sicher zu stellen, wird auf die Nutzung des Servers für diese Funktionalität verzichtet, weil die Verbindung zwischen Clients und Server nicht als stabil

angenommen werden kann. Statt dessen wird die Wahrscheinlichkeit doppelter vergebener Bezeichner dadurch reduziert, dass diese aus einer 128 Bit langen, gleich verteilten Pseudozufallszahl gebildet werden. Dieses Verfahren ist in Anlehnung an den RFC* 4122 [LMS05] der Internet Engineering Task Force (IETF) zur Erzeugung eines Universally Unique Identifier (UUID) realisiert.

Der erste Ansatz – Nutzung einer zentralen Instanz – wird ebenfalls verwendet, jedoch nicht für die Vergabe von Quellenbezeichnern, sondern nur für die Feststellung von uneindeutigen `ProviderId`s. Erhält der Server (Kap. 5.5.3) mehrere Providerbeschreibungen deren `ProviderId`s identisch sind, die sich jedoch in den übrigen Eigenschaften unterscheiden (d. h. P_i und C_i), so werden alle Provider mit derselben `ProviderId` als „nicht-verwendbar“ betrachtet und entsprechend an die Clients gemeldet. Für den Server ist nicht entscheidbar, welche Beschreibung oder welcher Bezeichner korrekt ist. Wenn der Server jedoch mehrere vollständig identische Providerbeschreibungen von unterschiedlichen Clients erhält, so bedeutet dies nur, dass eine entfernte oder globale Quelle zeitgleich von mehreren Providern gekapselt wird. Ist der Server nicht erreichbar, dann bleiben diese Inkonsistenzen unentdeckt und unbehandelt.

Hinweis 13. *Während der Lebenszeit eines Providers darf sich seine `ProviderId` nicht ändern. Providerimplementierungen, welche wechselnde Instanzen eines Quellentyps kapseln, müssen ihren Providerdienst beim Wechsel von einer Quelleninstanz zu einer anderen zuerst von der OSGi Service Platform abmelden und anschließend mit der Beschreibung der neuen Quelle (inkl. neuer `ProviderId`) wieder anmelden.*

Die Methode `getSourceId()` liefert bei allen Quellen den Quellenbezeichner zurück. Bei lokalen Quellen kann darüber hinaus die Methode `getHostId()` für den Zugriff auf den Gerätebezeichner genutzt werden. Für globale Quellen liefert diese Methode kein Ergebnis (`null`).

de.tud.mocawi.provider.HostId

Der Gerätebezeichner wurde bereits im vorangegangenen Kapitel angesprochen. Seine wesentlichste Eigenschaft besteht in der globalen Eindeutigkeit. Die Implementierung der `HostId` basiert auf der IP-Adresse des Gerätes, und setzt voraus, dass (a) das Gerät über eine Netzwerkschnittstelle verfügt, und (b) diese eine öffentliche IP-Adresse besitzt. Bei Geräten die keine solche Schnittstelle haben, besteht die Forderung der globalen Eindeutigkeit nicht, weil sie keine anderen Geräte erreichen können. Aufgrund des derart eingeschränkten „Ereignishorizonts“ sind alle lokal eindeutigen Bezeichner gleichzeitig global eindeutig. Probleme wie der Einsatz von Network Address Translation (NAT) oder die Auswahl der IP-Adresse auf Geräten mit mehreren Netzwerkschnittstellen werden aktuell nicht behandelt.

*Request for Comments (RFC)

Bei einem Wechsel der Kommunikationstechnologie von der C/S- hin zu einer Peer-to-Peer-Architektur auf Basis des JXTA-Projekts bietet sich die Nutzung der global eindeutigen `PeerId`s als Gerätebezeichner an.

de.tud.mocawi.access.ContextPattern

Die Bestandteile von Context Patterns wurden bereits in Kapitel 5.3 erläutert. Instanzen der Klasse `ContextPattern` repräsentieren diese Context Patterns. Im Gegensatz zu einem `ContextElementKey` der nur eine einzige Kontextinformation bezeichnet, ist es möglich, mit einem einzigen `ContextPattern` eine ganze Menge von Kontextinformationen zu beschreiben.

Während der Metatyp und der Domänentyp eines Patterns den Typ der Kontextinformation repräsentieren, müssen die übrigen Eigenschaften durch die Restriktion beschrieben werden. Seitens des Context Services sind keinerlei Eigenschaften definiert, geschweige denn eine Grammatik spezifiziert mit der Ausdrücke über den Eigenschaften möglich sind.

6.4. Realisierung von Consumern

Neben den bereits beschriebenen High-Level Providern enthält die Gruppe der Consumer auch die kontextbezogenen, adaptiven Softwaresysteme. Sie kapseln die Kontextsenken (engl. context sink), welche über den Context Service auf die Kontextquellen der Provider zugreifen. Während ein Provider einen Adapter zwischen Kontextquelle und Context Service darstellt, ist ein Consumer ein Adapter zwischen Kontextsenke und Context Service.

6.4.1. Integration der Kontextsenken

Weil es sich bei den Kontextsenken häufig um eigenständige, d. h. vom Context Service unabhängige, Software handelt, ist damit zu rechnen, dass nur wenige dieser Softwaresysteme auf Basis von Java oder sogar OSGi implementiert sind. Somit besteht eine Herausforderung in der Integration der Kontextsenken in den Context Service bzw. in die OSGi Service Platform. Nach erfolgter Integration ist der Zugriff von den Bundles der Kontextsenken auf den Context Service – und umgekehrt – theoretisch* problemlos möglich.

Die Aufgabe der Consumer besteht in der Abbildung des Informationszugriffs der Kontextsenken auf die Funktionalität und Informationsrepräsentation des Context Services. Als Teil dieser Kapselung der Kontextsenken wird auch die Kopplung zwischen Kontextsenke und Consumer verborgen. Zu deren Realisierung eröffnen sich

*Von Zugriffsbeschränkungen aus administrativen Gründen soll hier abstrahiert werden.

die in Abbildung 6.10 dargestellten Varianten. Teile die sich innerhalb des Rechtecks der OSGi Service Platform befinden, sind OSGi Bundles, alle anderen nicht. Welche Lösung gewählt wird, hängt von den Eigenschaften der Kontextsenken, sowie den jeweiligen Anforderungen ab und kann nur durch die Entwickler der Consumer entschieden werden.

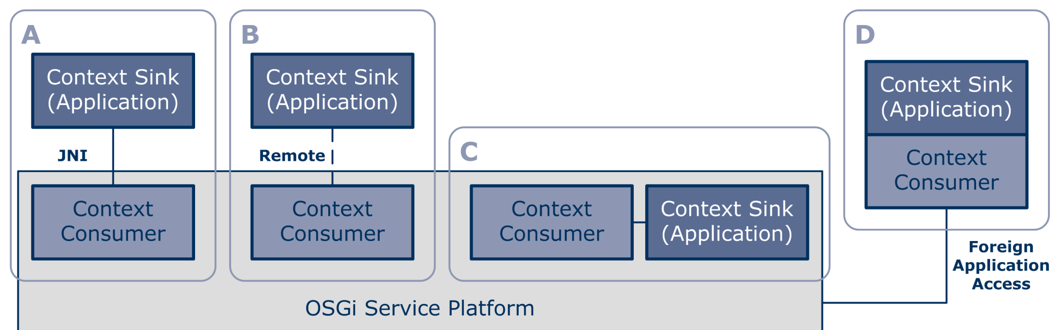

Abbildung 6.10.: Möglichkeiten der Kopplung von Context Consumer und Kontextsenke: A) JNI B) entfernte Kommunikation C) OSGi Bundle und D) Foreign Application Access

Variante A – Java Native Interface

Eine Kontextsenke die nicht in Java implementiert ist, sondern eine native (Betriebssystem-spezifische) Anwendung ist, kann mittels JNI auf den Consumer zugreifen – und umgekehrt. Details zur Verwendung des JNI sind der Dokumentation [Lia99] zu entnehmen.

Variante B – Entfernte Kommunikation

Alternativ zur Verwendung des JNI kann auch entfernte Kommunikation eingesetzt werden. So könnte der Consumer z. B. einen Web Service bereitstellen auf den die Kontextsenke zugreifen kann. Führt die Senke Abonnements durch, so muss sie selbst eine entsprechende Schnittstelle anbieten, um benachrichtigt zu werden.

Bei der Nutzung entfernter Kommunikation müssen sich die beteiligten Kommunikationspartner nicht auf unterschiedlichen Geräten befinden. Somit können auch lokale Kontextsenken auf den lokalen Context Service zugreifen. Beim lokalen Zugriff sind über entfernte Kommunikation theoretisch auch alle Formen der Interprozesskommunikation* nutzbar. Sind diese Kommunikationsformen jedoch nicht durch Java unterstützt, so wären sie für den Java-basierten Consumer nur unter Einsatz

*engl. Inter Process Communication (IPC)

des JNI anwendbar. Folglich reduziert sich die Menge der nutzbaren Kommunikationsmechanismen auf diejenigen, für welche eine Java-basierte Implementierung/Unterstützung existiert.

Variante C – OSGi Bundle

Kontextsenken die ohnehin als OSGi Bundles implementiert sind, können problemlos auf die Consumerimplementierung zugreifen – und umgekehrt. Dabei spielt es keine Rolle, ob sich beide im selben Bundle befinden, oder in unterschiedlichen Bundles. Jedoch bietet sich diese Kopplung – wie eingangs beschrieben – nur für den Fall an, dass die Kontextsenke eine OSGi-basierte Anwendung ist.

Variante D – Foreign Application Access

Für alle Anwendungen, welche zwar in Java implementiert sind, jedoch keine OSGi Bundles sind, bietet die OSGi Spezifikation die Möglichkeit des „Foreign Application Access“ an. Dabei handelt es sich um eine Schnittstelle mit der Nicht-OSGi-Software auf die Funktionen der OSGi Service Platform und die Dienste ihrer Bundles zugreifen kann. Dies schließt auch eventuell vorhandene Bundles von Consumern ein. Welche Schnittstellen zur Verfügung stehen, und wie diese zu benutzen sind, ist in der OSGi Spezifikation [OSG07b] beschrieben.

6.4.2. Durchführung des Kontextzugriffs

Zur Bedienung des Informationsbedarfs der Kontextsenke nutzt ein Consumer die unterschiedlichen Zugriffsformen – synchron und asynchron – die durch den Context Service und die Provider angeboten werden. Um dabei die benötigten Kontextinformationen zu spezifizieren, werden erneut Context Patterns eingesetzt.

Anfragen – Synchroner Zugriff

Der synchrone Zugriff einer Kontextsenke auf die Profile aller Nutzer ist in Quelltext 6.3 auf der nächsten Seite dargestellt. Die Zeilen 3 bis 5 enthalten lediglich den Zugriff auf die OSGi Service Platform, um eine Referenz auf die Zugriffskomponente zu erhalten, deren Dienst unter der Schnittstelle `de.tud.mocawi.access.QueryAndSubscribe` registriert ist. Die gewünschte Kontextinformation wird in Zeile 6 durch ein Context Pattern angegeben, welches lediglich die Typinformationen (Meta- und Domänentyp) spezifiziert. Beim Aufruf von `requestContext(ContextPattern)` mit dem Pattern werden die gewünschten Kontextinformationen zurück geliefert. In Abhängigkeit von der Verwendbarkeit geeigneter Provider und dem Inhalt des Patterns enthält die Ergebnismenge zwischen Null und beliebig vielen Zugriffsobjekten (Kap. 6.2.2) – je eines für jede gefundene Kontextinformation. Anschließend können

die Informationen verarbeitet werden (Zeilen 9 bis 13) indem die Methoden der Zugriffsobjekte verwendet werden. Werden die Kontextinformationen während dieser Methodenaufrufe nicht-verfügbar, so löst der Context Service eine `UnavailableContextException` aus, welche geeignet behandelt werden muss (Zeilen 14 bis 17).

Quelltext 6.3: Synchroner Kontextzugriff durch einen Consumer

```
public class ConsumerActivator implements BundleActivator, ContextSubscriber {
  public void start(BundleContext context) throws Exception {
    ServiceReference[] accessRefs = context.getServiceReferences(QueryAndSubscribe.class.getName(), null);
    if (accessRefs != null && accessRefs.length() == 1) {
      QueryAndSubscribe access = (QueryAndSubscribe) context.getService(accessRefs[0]);
      ContextPattern userPattern = new ContextPattern("EN", "user", null));
      Set users = access.request(userPattern);
      Iterator userIt = users.iterator();
      while (userIt.hasNext()) {
        Entity user = (Entity) userIt.next();
        try {
          // retrieve detailed information from user entity
          ...
        } catch (UnavailableContextException uce) {
          // handle missing context (retry?!)
          ...
        }
      }
    }
  }

  public void stop(BundleContext context) throws Exception {
  }
}
```

Das präsentierte Beispiel setzt natürlich voraus, dass ein Kontextmodell definiert wurde, welches der Beschreibung von Providern und Consumern zugrunde liegt. In diesem Modell ist auch die Entität „user" mit ihren Attributen und Assoziationen definiert. Jedoch existiert keinerlei Unterstützung für dieses Typsystem. Es ist folglich

Aufgabe der Entwickler von Providern und Consumern, nur die definierten Typen mit der festgelegten Semantik zu verwenden. Jede abweichende Verwendung kann dazu führen, dass die kontextbezogene Anwendung überhaupt keine Informationen erhält oder nicht die gewünschte Adaptivität zeigt.

Abonnements – Asynchroner Zugriff

In Quelltext 6.4 ist ein Subscriber auszugsweise dargestellt, welcher über einen längeren Zeitraum an Kontext interessiert ist. Wie es bei der Integration von Kontextsenken in den Varianten A, B und C erforderlich ist, wurde er als eigenständiges Bundle realisiert. Zur Vereinfachung ist in diesem Beispiel der Activator auch gleichzeitig der Subscriber. Deshalb implementiert die Beispielklasse `ConsumerActivator` die beiden Schnittstellen `BundleActivator` und `ContextSubscriber`.

Auch hier muss bei der Initialisierung – in der `startConsumer()`-Methode – eine Referenz zur Zugriffskomponente gefunden werden (Zeilen 6 bis 8). Die benötigte Kontextinformation – z. B. die Raumtemperatur des Zimmers „INF3100“ – wird wie zuvor durch Context Pattern beschrieben (Zeile 9) und bei der Registrierung des Abonnements (Zeile 10) an den Context Service übergeben. Als Ergebnis der Registrierung wird ein eindeutiges `SubscriptionTicket` erzeugt, welches das Abonnement, d. h. die Kombination aus Abonnent und abonnierter Kontextinformation, identifiziert. Mit diesem Ticket kann der Subscriber einerseits das Abonnement beenden (Zeile 23), indem er die Methode `unsubscribeFromContext(SubscriptionTicket)` der Anfrageschnittstelle aufruft. Andererseits kann ein Subscriber, welcher unterschiedliche Kontextinformationen abonniert hat, die per `notify(ContextUnavailableEvent, SubscriptionTicket)` eingehenden Benachrichtigungen den entsprechenden Abonnements zuordnen (Zeile 15) und ggf. die Kontextinformationen an die einzelnen Komponenten des Subscribers weiterleiten.

Quelltext 6.4: Asynchroner Kontextzugriff durch einen Subscriber

```
1  public class ConsumerActivator implements BundleActivator, ↘
   →ContextSubscriber {
     private QueryAndSubscribe access = null;
     private SubscriptionTicket myTicket = null;

5    public void start(BundleContext context) throws Exception↘
     → {
       ServiceReference[] accessRefs = context.↘
       →getServiceReferences(QueryAndSubscribe.class.getName()↘
       →, null);
       if (accessRefs != null && accessRefs.length() == 1) {
         this.access = (QueryAndSubscribe) context.getService(↘
```

```
          →accessRefs[0]);
          ContextPattern tempPattern = new ContextPattern("AT",↘
          → "room.temperature", "room.id='INF3100'"));
10        this.myTicket = access.subscribeToContext(tempPattern↘
          →, this);
      }
   }

   public void notify(ContextUnavailableEvent event, ↘
   →SubscriptionTicket ticket) {
15    if (event instanceof AttributeValueEvent && myTicket.↘
      →equals(ticket)) {
          Object newTemp = event.getNewValue();
          // notify context sink (for update of room ↘
          →temperature display)
          ...
      }
20 }

   public void stop(BundleContext context) throws Exception ↘
   →{
      this.access.unsubscribeFromContext(this.myTicket);
   }
25 }
```

Die Signatur der Methode `notify(ContextEvent, SubscriptionTicket)` lässt erkennen, dass einem Subscriber bei jeder Benachrichtigung nur ein Ereignis übergeben wird. Es erfolgt keine Sammlung von Ereignissen. Des weiteren erhält ein Subscriber nur dann – direkt nach der Registrierung des Abonnements – eine initiale Benachrichtigung, wenn entsprechende Provider verfügbar sind.

Hinweis 14. *Solange ein Subscriber keine initiale Benachrichtigung für ein Abonnement erhalten hat, ist dieses als „schwebend" (engl. pending) zu betrachten.*

Hinweis 15. *Sind nach erfolgter initialer Benachrichtigung zu einem beliebigen Zeitpunkt keine Provider – weder lokal noch entfernt – mehr verfügbar, um ein Abonnement zu bedienen, so übergibt der Manager beim Aufruf von `notify(ContextEvent, SubscriptionTicket)` kein `ContextUnavailableEvent`, sondern den Wert `null`. Der Subscriber muss in der Lage sein, mit einem solchen Argument umzugehen und es entsprechend zu bewerten.*

6.5. Verwendung von Filtern und Konvertern

Um bei der Providerselektion die Filterung anhand domänenspezifischer Merkmale zu unterstützen, müssen die Entwickler zuerst gültige Merkmalstypen definieren. Teil dieser Definition ist die Semantik des Merkmals, ein eindeutiger Typbezeichner, sowie eine Grammatik für gültige Ausdrücke bzw. Werte. Darauf aufbauend sollten einerseits die Merkmale in den Provided Pattern der entwickelten Provider deklariert werden (jeweils inkl. Ausdruck), und andererseits passende Filter entwickelt werden, welche die Merkmalsausdrücke auswerten können.

Jeder Filter wird wie ein Provider in Form eines OSGi-Bundles zur Verfügung gestellt. Um vom Kontextdienst gefunden zu werden, muss die Implementierung des `Filter`-Interfaces (Abb. 6.11 auf der nächsten Seite) bei der OSGi-Plattform registriert werden. Über die Methode `filtersFeatureTypes` muss der Filter bekanntgeben, anhand welcher Merkmalstypen er Context Pattern filtern kann. Zur Durchführung der Filterung ruft der `ContextBroker` die Methode `filter` auf, und übergibt dabei ein Consumed und ein Provided Pattern. Der Filter muss daraufhin die Ausdrücke der anzuwendenden Merkmalstypen miteinander vergleichen. Besitzen sie eine Schnittmenge, so muss der Methodenrückgabewert `true` sein, andernfalls `false`.

Hinweis 16. *Die Menge der Merkmalstypen, die durch einen Filter beim Vergleich zweier Pattern evaluiert werden müssen, ist die Schnittmenge der Merkmalstypen der beiden Pattern und des Filters.*

Für domänenübergreifende Interoperabilität können Konverter bereitgestellt werden, welche die Ausdrücke eines Merkmalstyps in Ausdrücke eines anderen Merkmalstyps umwandeln können. Analog zu den Filtern werden die Konverter ebenfalls als OSGi-Bundles realisiert. Das Bundle eines Konverters muss die Implementierung der `Converter`-Schnittstelle bei der OSGi-Plattform registrieren, und über die Methoden `inputFeatureType`, `outputFeatureType`, `isBidirectional` und `effectsFeatureTypes` die Selbstbeschreibung des Konverters an den `ContextBroker` liefern. Ruft der Broker die `convert`-Methode mit einem beliebigen Context Pattern auf, dann muss der Konverter den Ausdruck des Eingabemerkmalstyps in den des Ausgabemerkmalstyps umwandeln, sofern im Pattern der Eingabemerkmalstyp definiert ist. Ist das nicht der Fall, wird das Context Pattern unverändert ausgegeben. Nach erfolgter Konvertierung enthält das Pattern die Definition des Ausgabemerkmalstyps (inkl. Wert) anstatt des Eingabemerkmalstyps, sowie die veränderten Ausdrücke der zusätzlich beeinflussten Merkmalstypen.

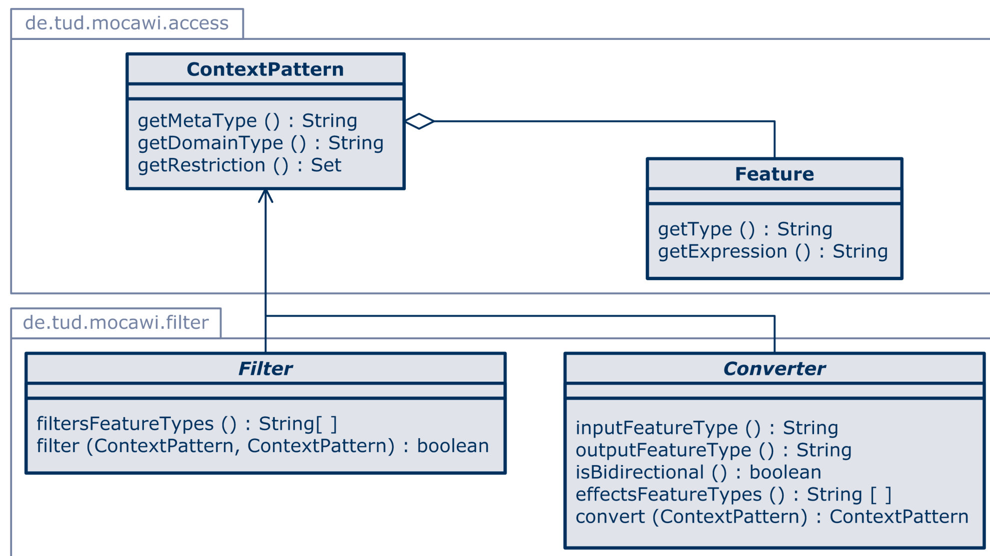

Abbildung 6.11.: Schnittstellen und Klassen für die Filterung von Context Patterns bei der Providerselektion

6.6. Evaluierender Einsatz des Context Service

Um die Verwendbarkeit des entwickelten Context Service Prototyps zu zeigen, wurde er als Teil der Laufzeitumgebung im Rahmen eines Modell-getriebenen Softwareentwicklungsprozesses für multimodale, adaptive Software eingesetzt [HHS08]. Dabei handelt es sich um ein System, welches die Techniker eines Chemieherstellers bei der Betreuung von Produktionsanlagen unterstützen soll. Im Folgenden soll an wesentlichen Aspekten des Entwicklungsprozesses die Behandlung von Kontextinformationen und der Einsatz des Kontextdienstes gezeigt werden.

6.6.1. Metamodelle

Die Basis der Methodik des Entwicklungsprozesses ist eine Menge von Objektorientierten Modellen. Die ihnen zugrunde liegenden Metamodelle wurden dementsprechend mit Hilfe der Meta Object Facility (MOF)* – einer Art Super- oder Meta-Metamodell – definiert. Zu den definierten Metamodellen gehören:

- *Konzept-*

*Auch die Unified Modelling Language (UML) ist ein Metamodell, welches mit Hilfe der MOF definiert wurde.

- *Prozess-*
- *Benutzungsschnittstellen-*
- *Geschäftslogik-* und
- *Kontextmetamodell*

Das Konzeptmetamodell gestattet die Modellierung aller für die Software notwendigen Informationen, ihrer Zusammenhänge und Datenstrukturen. Es bildet das Bindeglied zwischen den übrigen Metamodellen. Weil eine einheitliche Modellierung von Anwendungsdaten und Kontextinformationen angestrebt wurde, fiel die Wahl auf die OWL [TAA+03], deren Dialekt OWL – Description Logics (OWL-DL) prädikatenlogische Ausdrucksstärke bietet, und gleichzeitig das Kriterium der Entscheidbarkeit sichert. OWL hat bereits im Bereich der Wissensmodellierung breite Anwendung gefunden und wird auch von mehreren Forschergruppen zur Modellierung von Kontextinformationen eingesetzt. Um OWL – und andere Modellierungsansätze der Wissensmodellierung – in die Objekt-orientierte Welt zu integrieren, entwickelt die Object Management Group (OMG) das so genannte Ontoglogy Definition Metamodel (ODM) [OMG07], welches als Kern des Konzeptmetamodells ausgewählt wurde.

Das Prozessmetamodell erlaubt die Beschreibung von Prozessen mittels Daten- und Kontrollflüssen, sowohl kausal als auch temporal. Es werden zwei Prozessklassen definiert: Interaktionsprozesse und Systemprozesse. Erstere beschreiben die Interaktion mit den Nutzern des Systems, während letztere die Einbindung der Geschäftslogik repräsentieren. Somit bilden Prozesse das kontrollierende Bindeglied zwischen Benutzungsschnittstelle und Geschäftslogik.

Zur Beschreibung komplexer multimodaler Benutzungsschnittstellen existiert das Benutzungsschnittstellenmetamodell. Es gestattet die Entwicklung abstrakter, modalitätsunabhängiger Benutzungsschnittstellen. Diese werden im Laufe des Entwicklungsprozesses immer konkreter und modalitätsspezifischer.

Die Modellierung der Geschäftslogik – Kernaspekt vieler anderer Entwicklungsprozesse – wurde zugunsten der Fokussierung auf multimodale Benutzungsschnittstellen stark reduziert. So bietet das Geschäftslogikmetamodell lediglich die Mittel, um Schnittstellen zu den Bestandteilen der eigentlichen Geschäftslogik zu definieren und diese aufzurufen. Zur Entwicklung der darin gekapselten Funktionalität werden klassische Softwareentwicklungsprozesse vorausgesetzt.

Das Kontextmetamodell ermöglicht die Modellierung der Kontextnutzung und -bereitstellung. Die Nutzung von Kontextinformationen erfolgt über eine Erweiterung des Prozessmetamodells, welche es erlaubt, dass Prozesse analog zu Anwendungsdaten auch Kontextinformationen ein- und ausgeben können. Der Fluss von Kontextinformationen kann genauso definiert werden, wie der von Anwendungsdaten.

Ohnehin werden die Informationen/Daten beider Klassen mit Hilfe des Konzeptmetamodells einheitlich modelliert. Die Unterstützung der Kontextbereitstellung besteht in Modellierungskonzepten für die Charakteristika der Provider. Dies bezieht sich primär auf die von ihnen angebotenen Kontextinformationen, sowie – bei High-Level Providern – die von ihnen benötigten Kontextinformationen. Die Modellierung der Provider beschränkt sich auf die Klassenebene, d. h. es können Providertypen definiert werden, deren Instanzen sich identisch verhalten – die gleiche Informationen liefern und ggf. benötigen. Eine Unterstützung für die Modellierung von einzelnen Providerinstanzen gibt es nicht.

6.6.2. Entwicklungsmethodik

Das Vorgehen bei der Entwicklung adaptiver, multimodaler Anwendungen gliedert sich in vier Phasen:

1. die Anforderungsanalyse,
2. das abstrakte Systemdesign,
3. das detaillierte Systemdesign und
4. die Implementierung

All diese Phasen lassen sich in anerkannte Softwareentwurfsprozesse integrieren. Die Phasen 1 bis 3 und die in ihnen definierten Modelle werden durch Model-to-Model-Transformationen in einander überführt. Diese Transformationen basieren auf dem Standard MOF Query/View/Transformation (MOF QVT) [Obj08]. Beim Durchlauf der einzelnen Phasen werden die Modelle immer weiter verfeinert, sodass in Phase 4 die Umwandlung in ausführbare Software mittels Model-to-Code-Transformation erfolgen kann. Weil die Ausführungsumgebung Java-basiert ist, wurde das JET*-Framework der Eclipse-Entwicklungsumgebung eingesetzt. Mit seiner Hilfe kann u. a. Javaquelltext generiert werden.

In Phase 1 wird ein Anforderungsmodell definiert, welches in ein initiales Prozessmodell transformiert wird. Dieses Prozessmodell wird in Phase 2 verfeinert, wobei gleichzeitig auch das Konzeptmodell entsteht, welches die Datendefinitionen beherbergt, die für die Datenflüsse benötigt werden. Beim Übergang zu Phase 3 wird das Prozessmodell in das Benutzungsschnittstellenmodell und das Geschäftslogikmodell transformiert, und gleichzeitig auch das Konzeptmodell in das Kontextmodell. Bei diesen Transformationen wird bereits eine Trennung zwischen Kontextinformationen und Anwendungsdaten vorgenommen. Für alle im Prozessmodell eingebundenen Anwendungsdaten werden Aufrufe der Geschäftslogik generiert. Dem gegenüber

*Java Emitter Templates (JET)

werden für die Kontextinformationen entsprechende Beschreibungen von Low-Level Providern erzeugt. Diese Beschreibungen werden in Phase 3 präzisiert. Dazu gehört ggf. auch die Umwandlung einer Low-Level-Provider-Beschreibung in die eines High-Level Providers durch Binden der benötigten Kontextinformationen. Der Übergang zur letzten Phase erfolgt durch Quelltextgenerierung. Dabei wird die Objektorientierte Beschreibung der Benutzungsschnittstelle in XML-basierte Dokumente überführt. Aus dem Prozessmodell werden die Klassen der so genannten Task Process Engine erzeugt, welche zur Laufzeit die Ablaufsteuerung der generierten Software übernimmt (siehe Kap. 6.6.3). Nicht zuletzt werden auch die Schnittstellen der modellierten Provider generiert, wobei das Ontologie-basierte Datenmodell in ein Modell transformiert wird, dem das Metamodell des Context Service zugrunde liegt.

6.6.3. Laufzeitumgebung für multimodale, adaptive Software

Für die Ausführung multimodaler, adaptiver Software ist eine Laufzeitumgebung erforderlich, die einerseits die Nutzerinteraktion über eine multimodale Benutzungsschnittstelle ermöglicht, und andererseits die kontextbezogene Adaption der Interaktion und der Software gestattet. Zur Unterstützung des letzten Punktes wird der Context Service eingesetzt, während für den ersten Punkt die *Multimodal Services Component (MSC)** existiert. Die weiteren Bestandteile des Gesamtsystems sind die schon erwähnte Task Process Engine (TPE) und die generierte Software (siehe Abb. 6.12).

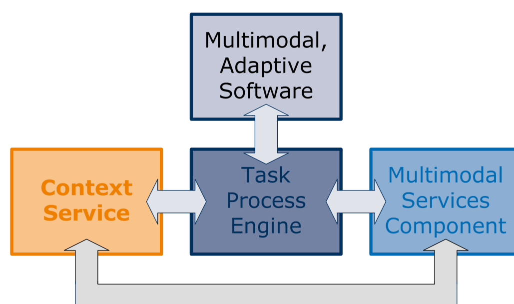

Abbildung 6.12.: Komponenten der Laufzeitumgebung für multimodale, adaptive Software

Die Task Process Engine greift – transparent für die multimodale, adaptive Software – auf Kontextinformationen zu, und leitet sie entweder an die Software oder auch die MSC weiter. Letztere kann jedoch auch direkt auf den Context Service zugreifen,

*Für Details zur MSC sei der interessierte Leser auf [HK07] und [Hüb08] verwiesen.

um die Kontextinformationen zu erhalten, welche in der Benutzungsschnittstellenbeschreibung referenziert sind.

6.6.4. Evaluationsanwendung

Zur Evaluation des Gesamtsystems wurde eine Anwendung modelliert, welche den Technikern in einem großen Chemiewerk die Betreuung der dort vorhandenen Anlagen erleichtert. Die Consumer werden durch einzelne Teilanwendungen gebildet (Abb. 6.13 auf der nächsten Seite):

1. Das *Organisationswerkzeug* (C_{MAN}) zeigt dem Techniker seine eigene Position, sowie die Lage der zu wartenden Anlagen und die notwendigen Arbeitsschritte (mit Anlage, Priorität und Beschreibung) an.
2. Im *Überwachungswerkzeug* (C_{MON}) werden dem Techniker die aktuellen Sensordaten der gewarteten Anlage präsentiert.
3. Mit dem *Verlaufswerkzeug* (C_{LOG}) kann der Techniker frühere, gespeicherte Sensordaten und ihren Verlauf überprüfen.

Die Adaptivität des Systems besteht in der Anpassung der Benutzungsschnittstelle des Organisationswerkzeuges. Zur Anzeige der zu wartenden Anlagen und der Position des Technikers überwacht das System letztere dauerhaft. Stehen dem System diese Positionsinformationen nicht zur Verfügung, unterbleibt die Darstellung der Übersichtskarte. Die Informationen werden ggf. vom GPS-Empfänger des PDA des Technikers geliefert. Des weiteren zeigt das Organisationswerkzeug eine Warnmeldung an, wenn sich der Techniker in der Nähe einer Anlage befindet, in der explosive Substanzen produziert oder verarbeitet werden. Bei Anzeige der Sensordaten im Überwachungswerkzeug werden die Kontextinformationen synchron von verschiedenen Sensoren der Anlagen abgefragt. Zu jedem dieser Sensoren ist der Verlauf ihrer Informationen während der letzten 15 Minuten über das Verlaufswerkzeug zugreifbar.

Für die Bereitstellung der notwendigen Informationen für die Evaluationsanwendung wurden mehrere Provider implementiert, welche entweder reale Kontextquellen kapseln oder virtuelle Kontextquellen emulieren. Letzteres war erforderlich, weil entsprechende physische Kontextquellen nicht zur Verfügung standen. Die Position des Technikers wurde durch einen Low-Level Provider (LLP_{GPS}) geliefert, welcher den GPS-Empfänger des PDA kapselte. Basis der Implementierung dieses GPS-Providers war eine frei verfügbare Java-basierte Bibliothek. Zur Indikation der Warnmeldung musste ein High-Level Provider (HLP_{WARN}) entwickelt werden der auf die Kontextinformationen des GPS-Providers und eines weiteren Low-Level Providers (LLP_{PLANT}) zugreift, welcher u. a. Anlagenpositionen und -beschreibungen liefert. Dieser Anlagenprovider liest aus einer Datei die notwendigen Informationen aus.

Eine reale Informationsquelle würde voraussichtlich auf eine zentrale Datenbank mit den notwendigen Informationen zugreifen. Für die Realisierung der Low-Level Provider, welche den Anlagenzustand liefern, mussten zusätzlich die von ihnen gekapselten Anlagensensoren emuliert werden – sie liefern zufällige Werte für Füllstände (LLP_{FILL}) und Temperaturen (LLP_{TEMP}). Passend zu diesen Low-Level Providertypen existieren jeweils High-Level Providertypen (HLP_{FILL}, HLP_{TEMP}), welche die Kontextinformationen der Low-Level Provider über einen Zeitraum von 15 Minuten protokollieren.

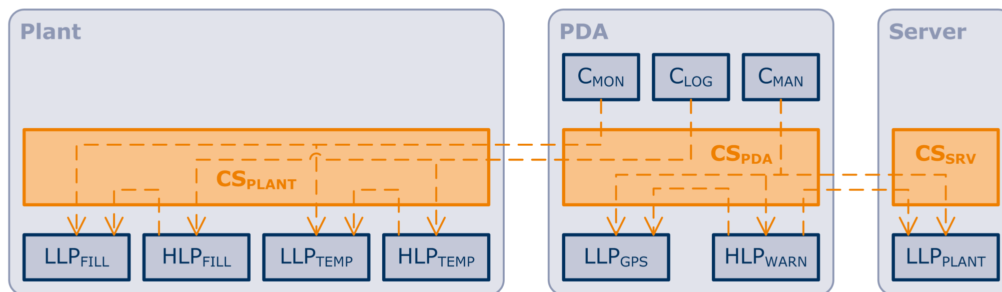

Abbildung 6.13.: Konfiguration der Evaluationsanwendung, sowie verteilte Kooperation ihrer Komponenten

6.6.5. Bewertung des Einsatzes des Context Service

Der Context Service konnte in einem modellgetriebenenen Softwareentwicklungsprozess eingesetzt werden. Dabei war es möglich, ein standardisiertes Metamodell für die Modellierung von Kontextinformationen zu verwenden, welches im Laufe des Entwicklungsprozesses auf das spezifische Metamodell des Context Service abgebildet wurde. Aufgrund der transparenten Einbindung des Context Services durch die generierte Process Engine war sogar eine einheitliche Nutzung von Anwendungsdaten und Kontextinformationen möglich (/A 1/). Der Context Service kam in der Konfiguration A, d. h. inklusive Kommunikationskomponente, zum Einsatz. Dabei war er in der Lage, die lokalen und entfernten Provider zu überwachen und zu verwalten und meldete ggf. die Nicht-Verfügbarkeit des Lokationsproviders (der den GPS-Empfänger kapselt) an die Consumer – hier die Process Engine und die MSC. Die MSC konnte sowohl momentan – bei Anzeige von Sensordaten – als auch dauerhaft – zur Verfolgung der Technikerposition – auf Kontextinformationen zugreifen.

6.7. Überprüfung der funktionalen Anforderungen

Um die Erfüllung der funktionalen Anforderungen (/A 1/–/A 3/, /A 5/ und /A 8/–/A 14/) zu überprüfen, wurde für jede Anforderung ein separater Funktionstest durchgeführt. Dabei wurde auf die Definition komplexer Testszenarien und -abläufe zugunsten einer verständlichen Darstellung verzichtet. Die Reihenfolge der überprüften Anforderungen orientiert sich an der Ordnung in Abbildung 2.3 auf Seite 27.

/A 1/ – Einheitliche Repräsentation und Zugriff von Kontext- und Anwendungsinformationen

Anforderung: „Softwaresysteme bzw. Kontextsenken sollen gleichermaßen einheitlich auf Kontext- und Anwendungsinformationen zugreifen können. Dies beinhaltet neben einheitlichen Zugriffsfunktionen auch die einheitliche Repräsentation der Informationen ungeachtet ihrer Art und ihres Ursprungs.“ (S. 9)

Test: Im Rahmen der Evaluationsanwendung (Kap. 6.6.4) wurden mehrere Low- und High-Level Provider implementiert. Diese kapselten einerseits typische Kontextquellen, andererseits Informationsquellen, welche vorwiegend als „klassische“ Anwendungsdatenquellen zu bezeichnen sind – z. B. Datenbanken. Auf diese Provider wurde durch mehrere Consumer zugegriffen.

Ergebnis: Es war möglich, Provider zu entwickeln, welche die Daten der Informationsquellen extrahierten und sie auf Basis des einheitlichen Metamodells, d. h. in Form von Entitäten, Attributen und Assoziationen, repräsentierten. Die Consumer konnten dabei über die zentrale `QueryAndSubscribe`-Schnittstelle einheitlich auf alle Informationen zugreifen, wobei der Context Service ad hoc die benötigten Provider abfragte.

Bewertung: *bestanden* – mit starken Einschränkungen

Der Zugriff auf die Provider ist zu grob granular, weil als Ergebnis von `request()` bzw. im Rahmen der `notify()`-Benachrichtigungen die gesamte Menge der von einem Provider lieferbaren bzw. momentan veränderten Kontextinformationen geliefert wird. Liefert eine Informationsquelle nur wenige Informationen, wie dies bei vielen Kontextquellen der Fall ist, dann ist beim Informationszugriff nur wenig Nacharbeitung erforderlich (z. B. Filterung, Selektion). Für Informationsquellen, welche jedoch einen großen Datenbestand liefern können, ist diese Form des Zugriffs unpraktikabel, weil die dadurch notwendige Nachverarbeitung (a) einen erheblichen Mehraufwand für den Context Service oder den Consumer bedeutet und (b) die

Leistungsfähigkeit der Informationsquelle (z. B. mächtige Anfragesprachen und effiziente Anfragebearbeitung) ignoriert. Folglich ist eine spezifischere Behandlung unterschiedlicher Quellentypen wünschenswert.

Während Mechanismen wie Caching und Prefetching zur Verbesserung der Verfügbarkeit von Kontextinformationen aufgrund deren Volatilität/Dynamik wenig sinnvoll sind, ist eine derartige Unterstützung für den Zugriff auf Anwendungsdaten nahezu unerlässlich, weil andernfalls die übergeordneten Anwendungen nicht mehr nutzbar sind. Diese Funktionalität ist prädestiniert für die transparente Auslagerung in einen Dienst der Middleware [Poh05].

/A 12/ – Asynchroner Zugriff auf Kontextinformationen

Anforderung: „Zur dauerhaften Überwachung ihrer Umgebung müssen Kontextsenken asynchron auf Kontextinformationen zugreifen können, d. h. sie müssen durch Benachrichtigungen über den aktuellen Zustand der Kontextvariablen unterrichtet werden.“ (S. 12)

Test: Für diesen Test wurde ein Low-Level Provider *LLP* implementiert, welcher zwei einfach strukturierte Kontextinformationen – hier Raumtemperatur und Helligkeit – in Form von Attributen bereitstellte. Diesen Attributen wurden in zufälligen Intervallen zufällige Werte zugewiesen. Zusätzlich wurden zwei Consumer C_1 und C_2 implementiert, die über die `subscribe()`-Methode der `QueryAndSubscribe`-Schnittstelle des Kontextdienstes versuchten, jeweils eine der angebotenen Kontextinformationen zu abonnieren. Es wurde untersucht, (a) ob C_1 und C_2 Benachrichtigungen über den veränderten Wert der Attribute erhalten, (b) ob jeder Consumer nur die von ihm abonnierten Benachrichtigungen erhält und (c) ob der Aufruf der `unsubscribe`-Methode die Benachrichtigungen beendet.

Ergebnis: Sowohl C_1 als auch C_2 erhielten in unregelmäßigen Abständen Benachrichtigungen über den aktuellen Wert der jeweils von ihnen abonnierten Kontextinformation. Dies war neben dem Meta- und Domänentyp der gelieferten Kontextinformationen auch an dem `SubscriptionTicket` zu erkennen. Es wurde als Teil jeder Benachrichtigung übermittelt, und erlaubte die Trennung bzw. Identifizierung der beiden Abonnements.

Bewertung: *bestanden*

/A 13/ – Synchroner Zugriff auf Kontextinformationen

Anforderung: „Um die aktuellen Charakteristika ihrer Umgebung ermitteln zu können, müssen Kontextsenken synchron auf ihre Kontextinformationen zugreifen kön-

nen. Nur wenn diese Informationen zeitnah zur Anfrage ermittelt werden, kann eine Basis für die konsistente, kontextbezogene Darstellung von Informationen oder Ausführung von Aktionen ermöglicht werden.“ (S. 13)

Test: Der Provider LLP aus dem Test von /A 12/ wurde für diesen Test wiederverwendet. Analog wurden zwei Consumer C_1 und C_2 eingesetzt, die jedoch über die `request()`-Methode der `QueryAndSubscribe`-Schnittstelle jeweils eine der von LLP angebotenen Kontextinformationen abfragten. Dabei wurde getestet, ob jeder Consumer ausschließlich die angefragten Kontextinformationen erhält.

Ergebnis: Erwartungsgemäß erhielt jeder Consumer nur die von ihm angefragten Kontextinformationen.

Bewertung: *bestanden*

/A 2/ – Generische Filterung von Kontextquellen und -informationen

Anforderung: „Die Ermittlung der Relevanz von Informationen ist durch eine generische Filterung der Kontextinformationen anhand verschiedener Kriterien zu unterstützen. Gleiches gilt auch für die Selektion entsprechender Kontextquellen.“ (S. 10)

Test: Basierend auf dem Beispiel aus Kapitel 5.4.2 wurden drei Tests mit unterschiedlicher Komplexität umgesetzt. In Variante A enthielt das Pattern des Consumers C keine Restriktion, während die Pattern der Provider LLP_{GPS} und LLP_{GK}, sowie die Beschreibungen des Filters F und des Konverters FC entsprechend des Beispiels aufgebaut waren. Variante B bestand aus C, F und LLP_{GK}, wobei das Pattern des Consumers die beschriebene Restriktion enthielt. In Variante C kamen alle Komponenten des Beispiels zum Einsatz. Zur Durchführung der Testvarianten stellte der Consumer eine synchrone Anfrage an den Context Service. Es wurde nur die Filterung von Providerbeschreibungen getestet, welche bei der Providerselektion zum Einsatz kommt.

Ergebnis: Als Resultat von Variante A wurden zwei Attribute an den Consumer ausgeliefert, weil dieser keine Restriktion spezifiziert hatte. In Variante B erhielt der Consumer keinerlei Kontextinformationen, weil der Filter zwar den Merkmalstyp des Consumed Pattern unterstützte, aber das Pattern des Providers diesen Merkmalstyp nicht definierte. Das Resultat von Variante C war mit dem von Variante A identisch.

Bewertung: *bestanden*

Die Filterung von Kontextquellen und -informationen wird vom System auf generische Art und Weise ermöglicht. Es liegt jedoch im Aufgabenbereich der Entwickler zusätzlich zu Consumern und Providern auch domänenspezifische Filter und domänenübergreifende Konverter bereitzustellen. Zur Untersuchung der Praktikabilität der Filterung von Kontextinformationen ist zuerst das System vollständig auf die Ausführung auf mobilen Geräten umzustellen, um den entsprechenden Ressourcenbedarf in einem praxisnahen Szenario zu ermitteln.

/A 3/ – Qualitative Beschreibung von Kontextinformationen und -quellen, sowie Signalisierung der Verfügbarkeit

Anforderung: „Kontext sollte dermaßen beschrieben werden, dass Kontextsenken in der Lage sind, die Qualität von Kontextinformationen zu bewerten und ihre Nutzung daran anzupassen, d. h. auch, diese ggf. zu unterlassen.“ (S. 12)

Test: Zur Durchführung des Tests wurden ein Provider, der eine Kontextvariable (Temperatur als `AttributeValue`) lieferte, und ein Consumer implementiert, der auf diese Kontextinformation zugriff. Der Provider ergänzte die von ihm gelieferten Kontextinformationen – konkret die Attribute – mit qualitativen Merkmalen. Der Consumer hingegen wurde derart implementiert, dass er bei der Abfrage bzw. beim Abonnement keine qualitativen Merkmale im Context Pattern spezifizierte, sondern die Merkmale der Restriktion der Kontextinformationen auf dem Bildschirm darstellte (analog Variante A des vorhergehenden Tests). In einem zweiten Teil des Tests wurde die Verfügbarkeit des Providers moduliert, indem er sich während der Bearbeitung der Anfrage/des Abonnements selbständig beendete.

Ergebnis: Alle Merkmale, welche den Kontextinformationen hinzugefügt wurden, konnten vom Consumer ausgelesen werden. Diese Verarbeitung war jedoch rein generisch – den Beschreibungsdaten wurde keine Semantik hinterlegt, welche zu einer konkreten Auswertung genutzt werden konnte.

Beendete sich der Provider während einer Anfrage, so löste der Context Service eine `UnavailableContextException` aus, und leitete sie an den Consumer weiter, weil kein alternativer Provider zur Verfügung stand. Analog wurde ein `ContextUnavailableEvent` ausgelöst, wenn sich der Provider während eines bestehenden Abonnements beendete. Nach anschließendem, erneutem Start des Providers wurde die Verfügbarkeit der Kontextinformation im Rahmen des Abonnements durch Übermittlung eines `AttributeEvent` mit dem aktuellen Wert des Attributs signalisiert.

Bewertung: *bestanden*

/A 8/ – Isolation des Fehlverhaltens von Kontextquellen

Anforderung: „Auswirkungen des Verhaltens (speziell des Fehlverhaltens) von Kontextquellen auf andere Kontextquellen, den Kontextdienst und die Kontextsenken sind nach Möglichkeit zu verhindern.“ (S. 22)

Selektiert der Context Service zur Bedienung von Anfragen und Abonnements einen Provider und greift er – stellvertretend für seine Consumer – auf dessen Kontextinformationen zu, so kann das dadurch ausgelöste Verhalten des Providers problematisch für die Funktionsfähigkeit des Context Service, sowie der übrigen Provider und der Consumer sein. Des weiteren muss sich ein Provider an die definierten Schnittstellen und seine Beschreibung halten, d. h. tatsächlich die Informationen liefern, welche er als lieferbar angibt. Weil jegliches Fehlverhalten eines Provider auszuschließen ist, wurden alle von einem Provider potentiell zu implementierenden Methoden geprüft.

Test: Die Prüfung umfasste einerseits das zeitliche Aufrufverhalten und den Inhalt der ggf. gelieferten Aufrufergebnisse. Dazu wurde ein erster High-Level Provider HLP_1 derart implementiert, dass bei Aufruf seiner Verwaltungsmethoden – `getId()`, `provides()`, `consumes()`, `isActivatable()`, `activate()` und `passivate()` – der aufrufende Thread für unbestimmte Zeit verzögert wurde. Weiterhin wurde ein zweiter High-Level Provider HLP_2 implementiert, dessen Verwaltungsmethoden zulässige Informationen lieferten, der aber bei Aufruf seiner Zugriffsmethoden – `request()`, `subscribe()` und `unsubscribe()` – die aufrufenden Threads unbestimmte Zeit verzögerte.

Andererseits wurde die Verarbeitung der von einem Provider gelieferten Kontextinformationen getestet. Zu diesem Zweck wurden zwei Low-Level Provider implementiert, welche (1) keine konformen Datenobjekte (LLP_1: `java.lang.String` statt `ContextElementDO`) bzw. (2) andere als die von ihnen spezifizierten Kontextinformationen lieferten (LLP_2: Temperatur statt Helligkeit).

Zur Beobachtung wurde der Context Service zusammen mit Consumern und Providern in einer Entwicklungsumgebung gestartet, sowie geeignete Haltepunkte im Context Service beim Aufruf der Provider-Methoden definiert.

Ergebnis: Bei den Aufrufen der Methoden von Provider HLP_1 wurde beobachtet, dass der aufrufende Thread* zunächst einen speziellen Timer-Thread startete. Anschließend übergab der aufrufende Thread seine Aufrufparameter an einen Access-Thread, startete diesen und pausierte sich selbst. Weil HLP_1 den Aufruf des Access-Threads für unbestimmte Zeit pausierte, reaktivierte der Timer-Thread den aufrufenden Thread nach einer festen Frist (1 Sekunde). Dieser stellte beim Zugriff

*Der aufrufende Thread kommt hier von der Überwachung der Provider und ist an OSGi-Mechanismen gekoppelt.

auf das erwartete Methodenergebnis des Access-Threads dessen Fehlverhalten fest, d. h. das Ergebnis lag nicht vor. Anschließend markierte der aufrufende Thread den Provider als fehlerhaft und beendete seine Ausführung.

Bewertung: *bestanden*

/A 14/ – Isolation des Fehlverhaltens von Kontextsenken

Anforderung: „Bei der Auslieferung der Kontextinformationen innerhalb der Kontextsenken auftretende Fehler sind durch den Kontextdienst zu behandeln, sodass ihre Auswirkungen auf die betreffende Kontextsenke isoliert bleiben.“ (S. 25)

Beim Zugriff einer Kontextsenke ist die Verwendung der `QueryAndSubscribe`-Schnittstelle durch den Consumer als problematisch anzusehen, weil dieser konform zur definierten Schnittstelle erfolgen muss. Ein großer Teil der dabei potentiell auftretenden Probleme ist durch die strenge Typbindung von Java bereits a priori ausgeschlossen. Fehlerhafter Einsatz der bereitgestellten Methoden ist jedoch nicht ausgeschlossen. Weiterhin ist ein gewisses Wohlverhalten der Implementierung des Consumers erforderlich, um die Arbeitsweise des Context Service und der übrigen Komponenten des Systems nicht zu beeinträchtigen.

Test: Weil eine Kontextsenke auf zwei Arten – synchron und asynchron – auf Kontextinformationen zugreifen kann, sind zwei Formen des Tests erforderlich. Für die Tests wurde ein speziell präparierter Consumer implementiert, welcher unerwünschtes Verhalten zeigte.

Der erste Teil bestand im synchronen Zugriff auf Kontextinformationen. Weil für den synchronen Zugriff lediglich die `request()`-Methode zur Verfügung steht, ist das zugehörige Protokoll der Schnittstelle auch entsprechend einfach. Folglich wurde der Consumer derart entwickelt, dass er eine konkrete Kontextinformation – spezifiziert durch ein `ContextPattern` – über die `request()`-Methode abfragte. Für die Beobachtung des ersten Teils wurde das System in einer Entwicklungsumgebung gestartet. Dabei wurde der aufrufende Java-Thread des Consumers im Schrittbetrieb abgearbeitet, während alle anderen Threads und Prozesse der VM unabhängig davon ausgeführt werden durften.

Beim zweiten Teil führte der Consumer – analog zum ersten Teil – einen asynchronen Zugriff über die `subscribe()`-Methode durch und registrierte sich dabei entsprechend. Als Teil dieser Registrierung wurde im Consumer die `notify()`-Methode derart implementiert, dass aufrufende Threads für unbestimmte Zeit zum Warten gezwungen wurden. Der zweite Teil wurde analog zum ersten in einer Entwicklungsumgebung gestartet. Für die Auslösung des Schrittbetriebs wurden zwei Haltepunkte definiert. (1) bei der Durchführung des Abonnements – Aufruf der `subscribe()`-Methode durch den Consumer – und (2) bei der Erzeugung eines Kontextereignisses

durch den selektierten Provider – bei dessen Aufruf der `notify()`-Methode des Context Service.

Ergebnis: Der erste Teil zeigte, dass der Consumer-Thread im Zuge seiner Ausführung (für die Selektion geeigneter Provider) auch Kernfunktionen des Context Service aufrief, was durch gegenseitigen Ausschluss (Mutex) zu entsprechenden kurzzeitigen Blockierungen der übrigen parallelen Threads führte. Der anschließende Zugriff des Consumer-Threads auf die Kontextinformationen des Providers über dessen `request()`-Methode führte zum Blockieren der parallelen Threads des Providers, welche auf gemeinsame Objekte zugegriffen haben. Abschließend kehrte der Consumer-Thread aus dem Context Service zurück und beendete seine Funktion.

Die Durchführung des zweiten Teils zeigte bezüglich des ersten Haltepunktes – analog zum ersten Teil – die erwarteten Blockierungen paralleler Threads beim Zugriff auf gemeinsame Objekte des Context Service. Nach der Durchführung des Abonnements beendete der Consumer-Thread seine Funktion. Aufgrund durch einen Provider emulierter Kontextereignisse wurde der zweite Haltepunkt ausgelöst und zeigte ebenfalls die erwarteten Blockierungen bei der Verarbeitung des Ereignisses durch den Context Service. Diese Verarbeitung wurde durch das Erzeugen eines `Notifier`-Threads beendet. Der anschließende Aufruf der `notify()`-Methode des Consumers führte lediglich zum Blockieren des `Notifier`-Threads – die übrigen Threads des Context Service wurden weiterhin ungehindert ausgeführt.

Bewertung: *bestanden*

Das Blockieren von parallelen Threads im Context Service, sowie von genutzten Providern ist architektonisch bedingt, und unvermeidbar, um ein konsistentes Verhalten des Context Service zu erreichen. Es besteht kein Einfluss des Consumers auf die Ausführung des Aufrufs von `request()` oder `subscribe()`, sodass beabsichtigte oder unbeabsichtigte Seiteneffekte ausgeschlossen sind.

Das vermeidbare Blockieren des Context Service und des selektierten Providers bei der Benachrichtigung des Consumers/Subscribers wurde erfolgreich demonstriert. Ein un-/beabsichtigtes Verzögern von benachrichtigenden Threads hat keinen Effekt auf die Funktionsfähigkeit des Context Service.

Aufgrund fehlender Kontrolle über die Speicherverwaltung und den Scheduler der VM kann der Context Service Seiteneffekte von Consumern, welche übermäßig Systemressourcen konsumieren, nicht vermeiden. Um diese überhaupt zu erkennen, wäre eine Überwachung der Consumer erforderlich. Die damit einhergehenden Voraussetzungen würden den Entwurf von Consumern und somit die Integration von Kontextsenken weiter erschweren. Alternativ wäre die Unterstützung von Echtzeitanforderungen durch die Java Ausführungsumgebung erforderlich.

/A 5/ – Überwachung von Kontextquellen

Anforderung: „In einer Umgebung verfügbare Kontextquellen müssen jederzeit gefunden und überwacht werden, um Kontextsenken ein Maximum an Kontextquellen bereitstellen zu können. Das gilt besonders, wenn bisher genutzte Quellen nicht mehr verfügbar/erreichbar sind, und somit alternative Quellen für dieselben oder ähnliche Informationen benötigt werden.“ (S. 14)

Test: Weil sich Anforderung /A 5/ aus /A 9/ und /A 11/ zusammen setzt, ist für sie kein separater Test erforderlich.

Ergebnis: siehe Ergebnisse der Tests von /A 9/ und /A 11/

Bewertung: *bestanden*

/A 9/ – Überwachung lokaler Kontextquellen

Anforderung: „Die Menge der Knoten-lokalen Kontextquellen ist dauerhaft zu überwachen, um zu jedem Zeitpunkt deren Verfügbarkeit feststellen zu können.“ (S. 22)

Test: In Vorbereitung des Tests wurden drei Provider LLP, HLP_1 und HLP_2 implementiert. Die Informationsabhängigkeiten zwischen ihnen wurden so spezifiziert, dass HLP_1 ausschließlich von LLP abhängig war. HLP_2 dagegen war sowohl von LLP, als auch von HLP_1 abhängig. Des weiteren wurde ein Consumer C implementiert, welcher die lieferbaren Kontextinformationen der Provider synchron abfragte, sodass deren Verwendbarkeit implizit festgestellt werden konnte. Die Steuerung der Registrierung der Provider erfolgte durch ein Testprogramm, welches in Form eines Bundles sowohl die Dienste der Provider beim OSGi-Framework registrierte und de-registrierte, als auch die Prüfung der Providerverwendbarkeit durch Aufruf des Consumers auslöste. Es wurden sequentiell alle acht potentiell möglichen Kombinationen der drei Provider geprüft.

Ergebnis: War keiner der Provider registriert, so lieferte die Prüfung erwartungsgemäß keine Kontextinformationen. Bei der Registrierung eines einzelnen Providers war lediglich die Prüfung erfolgreich, bei welcher ausschließlich Provider LLP registriert war. Beim Einzelbetrieb von Provider HLP_1 bzw. HLP_2 wurden diese korrekt als nicht-verwendbar erkannt. Die gleichzeitige Registrierung von zwei Providern führte in der Kombination (LLP, HLP_1) zur Verwendbarkeit beider Provider und in der Kombination (LLP, HLP_2) zur Verwendbarkeit von LLP. In der Kombination (HLP_1, HLP_2) erhielt der Consumer keinerlei Kontextinformationen, weil kein

Provider verwendbar war. Bei der Registrierung aller drei Provider wurden diese als verwendbar erkannt, und ihre Kontextinformationen ausgeliefert.

Bewertung: *bestanden*
Es wurde lediglich die Überwachung der Context Provider getestet – nicht die der Kontextquellen. Letztere ist nur dann sinnvoll und möglich, wenn Provider existieren durch welche die Kontextquellen für den Context Service nutzbar sind. Somit ist der Test der Provider notwendig und hinreichend.

/A 11/ – Überwachung entfernter Knoten und ihrer Kontextquellen

Anforderung: „Die Erreichbarkeit entfernter Knoten, sowie die Verfügbarkeit ihrer lokalen Quellen ist ebenfalls zu überwachen.“ (S. 23)

Test: Die Provider des vorangegangenen Tests wurden in diesem Test wiederverwendet. Sie wurden jedoch auf zwei Rechner verteilt. Auf dem lokalen Testsystem wurden LLP und HLP_2 betrieben, während HLP_1 auf dem entfernten System installiert wurde. Die De-/Registrierung der Provider wurde manuell über ein Graphical User Interface (GUI) des OSGi-Frameworks kontrolliert. Gleiches galt für den Consumer C der versuchte, die Kontextinformationen aller drei Provider abzufragen. Es wurden erneut alle Kombinationen der Provider getestet. Zusätzlich wurde die Überwachung des entfernten Knotens durch Steuerung der entfernten Kontextdienstinstanz geprüft.

Ergebnis: Die Ergebnisse der Tests zur Überwachung der entfernten Kontextquellen waren identisch mit denen der Überwachung lokaler Kontextquellen.

Bewertung: *bestanden*
Auf die Ermittlung der Verzögerungen bei der Aktualisierung der Providerbeschreibungen wurde verzichtet, weil die Realisierung eines entsprechenden Testszenarios unvertretbar hohen Aufwand zur Folge gehabt hätte. Dieser Aufwand begründet sich einerseits im Problem der Koordination der verteilten Testabläufe und der dafür notwendigen global einheitlichen Systemzeit. Andererseits ist ein reales Netzwerk mit nennenswerter Ausdehnung und somit nennenswerter Nachrichtenumlaufzeit erforderlich. Die Emulation von Netzwerklatenzen würde lediglich zur Messung dieser emulierten Latenzen führen.

/A 10/ – Automatische Selektion alternativer Quellen

Anforderung: „Ist eine Quelle nicht mehr verfügbar, so sind alternative Quellen zu bestimmen und ggf. zu verwenden. Dies hat automatisch und für die Senke transpa-

rent zu erfolgen. Nur im Falle fehlender Alternativen sind eventuelle Kontextsenken zu benachrichtigen.“ (S. 22)

Test: Für diesen Test wurden zwei Low-Level Provider LLP_1 und LLP_2 implementiert, welche die gleiche Kontextinformation – Raumtemperatur – liefern. Beide emulierten einen schwankenden Temperaturverlauf durch zufällige Temperaturwerte die in zufälligen Intervallen verändert wurden. Ein Consumer C wurde so implementiert, dass er die Temperaturinformationen abonnierte. Zur Feststellung des liefernden Providers ermittelte der Consumer aus den Schlüsseln der gelieferten Objekte den Bezeichner des Providers und gab eine Meldung aus wenn er sich änderte bzw. wenn er nicht ermittelt werden konnte. Analog zum Test von Anforderung /A 9/ wurden die Provider durch eine Testanwendung in Form eines OSGi-Bundles gesteuert. Die Testanwendung registrierte und de-registrierte die Provider so, dass abwechselnd gar kein Provider, ein Provider oder beide Provider registriert – und entsprechend verwendbar – waren. Im Rahmen des Tests wurden folgende Übergänge zwischen den Kombinationen der Provider untersucht: $(LLP_1) \rightarrow (LLP_1, LLP_2) \rightarrow (LLP_2) \rightarrow (\emptyset) \rightarrow (LLP_1) \rightarrow (\emptyset) \rightarrow (LLP_2)$

Ergebnis: Bei den Übergängen wurden folgende Veränderungen der genutzten Provider beobachtet: $LLP_1 \rightarrow LLP_1 \rightarrow LLP_2 \rightarrow \emptyset \rightarrow LLP_1 \rightarrow \emptyset \rightarrow LLP_2$. Diese Beobachtung stimmt mit den Erwartungen ein. Beim ersten Übergang wurde keine alternativer Provider selektiert, weil der Bedarf bereits durch LLP_1 abgedeckt wurde. Bei den übrigen Übergängen jedoch musste ggf. der einzige, verwendbare Provider selektiert werden.

Bewertung: *bestanden*

6.8. Diskussion der nicht-funktionalen Anforderungen

Für die Evaluation der nicht-funktionalen Anforderungen /A 6/, /A 7/ und /A 15/ bis /A 17/ ist eine qualitative Beurteilung des Kontextdienstes und der gewählten Konzepte erforderlich. Eine Ausnahme bildet hier Anforderung /A 4/. Analog zu den Funktionstests werden die nicht-funktionalen Anforderungen anschließend einzeln behandelt. Die Reihenfolge orientiert sich ebenfalls an der Ordnung in den Abbildungen 2.3 und 3.2 auf den Seiten 27 und 58.

/A 4/ – Verhinderung veralteter Kontextinformationen

Anforderung: “Der Zugriff auf Kontext muss zeitnah erfolgen, sodass die bereitgestellten Kontextinformationen auch noch aktuell sind und die zeitnahe Anpassung

daran möglich ist.“ (S. 13)

Test: Diese Anforderung wurde durch einen Funktionstest überprüft, weil die Messung der Latenz beim Kontextzugriff auf ausgewählte Kontextinformationen einzelner Provider in einem lokalen Netzwerk keine allgemein gültige Aussagekraft besitzt (siehe Bewertung).

Teil diese Tests war ein Low-Level Provider *LLP* der eine Entität vom Typ „Raum“, sowie deren zwei Attribute „Temperatur“ und „Helligkeit“, bereitstellte. Die beiden Attribute wurden in regelmäßigen Intervallen auf zufällige Werte gesetzt, sodass die Dynamik der Kontextinformationen sichergestellt war. Um das anforderungskonforme Verhalten des Context Service prüfen zu können, musste das Verhalten von *LLP* konform zu den definierten Schnittstellen sein (Kap. 6.3). Das hieß einerseits, dass beim synchronen Zugriff über die `request()`-Methode die aktuellen Werte der Attribute bestimmt wurden, und die Entität mit ihren Attributen zurück geliefert wurde. Anderseits mussten Wertänderungen der Attribute an den Kontextdienst signalisiert werden, sofern sich dieser für die Kontextinformationen des Providers registriert hatte – durch `subscribe()` bzw. `unsubscribe()`.

Ein speziell implementierter Consumer *C* griff auf die Kontextinformationen zu. Im ersten Teil des Tests wurden dazu regelmäßig* die Attribute direkt abgefragt, und deren Werte beobachtet. Beim zweiten Teil wurde regelmäßig die Entität abgefragt, anschließend wurde navigierend, d. h. über ihre `getAttributes()`-Methode, auf die Attribute zugegriffen und endlich deren Werte ermittelt. Der dritte und abschließende Test bestand in der Durchführung von Abonnements für die beiden Attribute und Beobachtung ihrer Werte. Die Auswertung des Tests erfolgte sowohl durch Beobachtung der abgefragten/abonnierten Informationen, als auch durch Überprüfung der Funktionsweise des Context Service im Schrittbetrieb innerhalb einer Entwicklungsumgebung.

Ergebnis: Beim ersten Test waren regelmäßig wechselnde Attributwerte zu beobachten. Die Untersuchung im Schrittbetrieb zeigte, dass selbst bei wiederholten Aufrufen die Informationen stets durch zeitnahe Abfrage des Providers ermittelt wurden. Die Beobachtungen des zweiten Tests waren analog zum ersten. Im Schrittbetrieb wurde zusätzlich erkennbar, dass zur Ermittlung der Attribute der Entität ebenfalls zeitnah auf den Provider zugegriffen wurde. Auch der dritte Test bestätigte die Erwartungen indem regelmäßig Änderungen der abonnierten Attribute signalisiert wurden.

Bewertung: *bestanden* – mit theoretischen Einschränkungen

*Mit demselben Intervall wie die emulierte Änderung der Attributwerte.

Die Definition von Zeitschranken für die Gültigkeit und somit die Relevanz der benötigten Kontextinformationen erfolgt durch jeden Consumer separat, d. h. Anwendungs- bzw. Domänen-spezifisch, bei Ausführung der Anfrage bzw. Durchführung des Abonnements. Somit ist die Messung der Latenz beim Kontextzugriff spezifisch für jeden Consumer durchzuführen, und kann nicht allgemein gültig beantwortet werden.

Grundsätzlich ist von der Existenz praktischer Untergrenzen für die Latenz beim Zugriff auf lokale und entfernte Provider auszugehen. Diese Grenzen werden durch Faktoren wie CPU-Leistung, Netzwerktopologie und -latenz, etc. beeinflusst. Weil speziell bei der Nachrichtenumlaufzeit Verzögerungen im Bereich von Milli- bis hin zu Zehntelsekunden auftreten können, würden Consumer-seitige Anforderungen, welche weit unter diesen Werten liegen, die Bedienung der Consumer verhindern. Folglich ist die Definition von moderaten zeitlichen Anforderungen notwendig. Dies bedeutet auch, dass sich der Einsatz des Kontextdienstes für Anwendungen mit Echtzeitanforderungen nicht empfiehlt.

Um bei der Bedienung von Anfragen und Abonnements der Consumer die Aktualität der gelieferten Informationen zu erzwingen, wird zur Ausführungszeit durch das dynamische Binden der Provider und den zeitnahen Zugriff auf ihre Daten ein signifikanter Zeit- und Rechenaufwand verursacht. Die Reduzierung des Zeitaufwandes wäre jedoch nur durch Zwischenspeicherung (engl. caching) bereits abgefragter Kontextinformationen möglich. Dazu sind einerseits Informationen über die theoretischen Möglichkeiten zur Zwischenspeicherung erforderlich – nicht jede Kontextinformation ist dafür geeignet, d. h. wenig volatil. Andererseits wird der reduzierte Zeitaufwand durch einen erhöhten Speicheraufwand (für den Zwischenspeicher) eingetauscht. Die Informationen über die potentielle Möglichkeit zur Zwischenspeicherung können entweder durch Erweiterung des Metamodells während der Entwicklung der Anwendungen und Provider, oder mittels Beobachtung der Volatilität der Kontextinformationen durch den Kontextdienst selbst hinzugefügt werden. Für letzteres wäre jedoch eine entsprechende Erweiterung des Kontextdienstes erforderlich.

/A 6/ – Einheitliche Verwaltung und Zugriff auf alle Kontextquellen

Anforderung: "Die große Heterogenität der Kontextquellen ist vor den Kontextsenken zu verbergen, sodass der einheitliche Zugriff auf unterschiedlichste Kontextquellen möglich ist.“ (S. 22)

Diskussion: Der einheitliche Zugriff auf Kontext- bzw. Informationsquellen unterschiedlicher Art (z. B. Sensor, Datenbank) wurde bereits durch die Realisierung und den Betrieb der Evaluationsanwendung exemplarisch gezeigt. Er geht einher mit dem einheitlichen Zugriff auf Kontext- und Anwendungsinformationen. Dem entsprechend wurde bei der Diskussion von /A 1/ richtiger weise betont, dass die Ver-

einheitlichung der unterschiedlichen Klassen von Informationsquellen zu den zwei Klassen Low- und High-Level Provider im praktischen Einsatz nicht ausreichend ist. Auf Seiten des Kontextdienstes muss über das aktuelle Maß hinaus Aufwand betrieben werden, um die Heterogenität der Informationsquellen auszunutzen, und sie trotzdem vor den Senken transparent zu verbergen.

Bewertung: *bestanden* – mit Einschränkungen (siehe Evaluation von /A 1/)

/A 7/ – Minimaler Aufwand und Wechselwirkung mit Kontextquellen

Anforderung: "Die Integration der Kontextquellen in das Gesamtsystem hat derart zu erfolgen, dass die dafür notwendigen Maßnahmen geringen Aufwand mit sich bringen, und nur geringe Wechselwirkungen mit den Kontextquellen zur Folge haben." (S. 22)

Diskussion: Die grundsätzliche Konzeption der Realisierung von Context Providern (Kap. 6.3) vermeidet durch Definition der Schnittstellen in Form von Interfaces (nicht abstrakten Klassen) typische Probleme bei der Integration von Java-basierten Anwendungen – speziell fehlende Unterstützung für Mehrfachvererbung. Weiterhin stellen die Schnittstellen des Kontextdienstes die notwendige Funktionalität zur Verfügung, um aus Kontextquellen extrahierte Informationen Metamodell-konform zu repräsentieren. Darüber hinaus vereinfacht die Bereitstellung des Provider Frameworks (Kap. 5.2.1) die Implementierung von Providern dergestalt, dass die verantwortlichen Entwickler primär mit dem Problem der Extraktion von Informationen aus den Quellen beschäftigt sind.

Bewertung: *bestanden*
Die gebotene Unterstützung für die Entwicklung der Provider kann trotz allem die Heterogenität der Kontextquellen nicht vor den Entwicklern der Provider verbergen. Eine Reduktion des Integrationsaufwandes ist lediglich durch Entwicklung Domänen-spezifischer Software-Bibliotheken zu erwarten, welche wesentliche Bausteine für den Zugriff auf die Domänen-spezifischen Kontextquellen bereitstellen.

/A 16/ – Schonung der Ressourcen mobiler und integrierter Geräte

Anforderung: "Den begrenzten Ressourcen der mobilen und spezialisierten Geräte ist Rechnung zu tragen. Dies gilt speziell für deren meist schmalbandige und langsame Kommunikationsschnittstellen." (S. 46)

Diskussion: Bei der Realisierung wurden die Voraussetzungen zum Einsatz des Context Service auf mobilen und integrierten Geräten durch die Auswahl einer OSGi- und Java-basierten Laufzeitumgebung geschaffen. Darüber hinaus gestattet die modulare Architektur (Kap. 6.2) den angepassten Einsatz des Context Service in verschiedenen Szenarien und auf den darin anzutreffenden Geräten. Im Rahmen der Evaluationsanwendung wurde das System auf einem PDA und einem Notebook betrieben. Die dabei verwendete Version mit zentraler, proaktiver Verteilung der Providerbeschreibungen benötigt (in Konfiguration A; Kap. 6.2.1) weniger als ein Megabyte persistenten Speicherplatz. Dies ist eine Größenordnung welche aktuell auf der Mehrheit der mobilen Geräte problemlos verfügbar sein sollte.

Problematisch ist der schon angesprochene Zeit- und Rechenaufwand während des Betriebs des Context Service und der Ausführung der Kontextbezogenen Anwendungen. So ist der Aufwand zur Analyse von gestarteten Providern (Kap. 5.4.4) aufgrund der Komplexität des Markierungsalgorithmus ($\mathcal{O}(n^2)$) überproportional von der Anzahl bereitgestellter Kontexttypen abhängig. Weil für das Verhältnis vorhandener Provider zu ihren Kontexttypen schlimmstenfalls von einem linearen Zusammenhang ausgegangen werden muss, wirkt sich die Komplexität des Algorithmus in vollem Umfang aus, und führt zu Verzögerungen bei der Einbindung der Provider in die Verwaltung des Context Service. Dieser Umstand kann im praktischen Einsatz abgeschwächt auftreten, weil das Starten und Stoppen von Providern nicht kontinuierlich statt findet, sondern meist schubweise – z. B. wenn ein Gerät ein- bzw. ausgeschaltet wird, oder Funkkontakt herstellt bzw. verliert.

Mangels geeigneter anwendungsbasierter Benchmarks und eines größeren Testszenarios ist eine quantitative Bewertung der Auswirkungen der Kommunkationsprotokolle auf die Nutzung der Netzwerkschnittstellen der mobilen Geräte nicht möglich. Das stark begrenzte Szenario der Evaluationsanwendung war a priori nicht in der Lage, entsprechende Schwachstellen aufzuzeigen. Jedoch offenbarte die theoretische Betrachtung, dass die proaktive Verteilung der unzähligen Aktualisierungen durch den Server und deren Empfang durch die Clients die Netzwerkschnittstellen aller Knoten überlasten. Dieses Problem kann nur durch eine Paradigmenwechsel bei der verteilten Providerverwaltung gelöst werden (Kap. 5.5.5). Die Lösung besteht weiterhin in der Abkehr vom starren C/S-Paradigma hin zu einer hybriden Peer-to-Peer-Architektur. Die Betrachtung des Kommunikationsaufwandes des P2P-Systems offenbarte, neben des großen Potentials zur Verbesserung der Skalierbarkeit, die gleichzeitig wachsende Belastung für das Gesamtsystem.

Bewertung: *nicht evaluierbar*

Vor der praktischen Evaluation der Peer-to-Peer-basierten Providerverwaltung sind weitere grundlegende Probleme zu lösen. Dazu gehört die Spezifizierung und Realisierung von repräsentativen anwendungsbasierten Benchmarks, welche eine um-

fassende Beurteilung von Kontextdiensten erlauben.

/A 17/ – Skalierbare Verwaltung großer Mengen von Kontextquellen und Geräten

Anforderung: "Das System muss eine hohe Zahl von Informationsknoten ($> 10^5$) und -quellen ($> 10^7$) verwalten können." (S. 54)

Diskussion: Die vorausgegangene Betrachtung von /A 16/ hat bereits gezeigt, dass eine C/S-basierte Lösung nicht skalierbar ist. Skalierbarkeit ist nur durch den Einsatz einer Peer-to-Peer-basierten Verwaltung zu erreichen (Kap. 5.6). Der zu erwartende Aufwand für die Verwaltung der Peers sinkt aufgrund der Effizienz des strukturierten Überlagerungsnetzes der Super Peers um mehrere Größenordnungen. Für die Verwaltung der Provider eine ähnliche Verbesserung zu erwarten. Die Peer-to-Peer-basierte Verwaltung verspricht eine gleichmäßige Verteilung des Verwaltungsaufwandes auf die Super Peers und somit die Erhöhung der Skalierbarkeit des Gesamtsystems. Grundvoraussetzung für den erfolgreichen Einsatz ist die Existenz eines hinreichend großen Prozentsatzes von Knoten, welche die Rolle der Super Peers übernehmen können. Diese Knoten sollten leistungsfähig und stationär sein.

Bewertung: *nicht evaluierbar*
Weil praktische Untersuchungen noch nicht möglich sind, kann diese Anforderung nicht abschließend evaluiert werden.

/A 15/ – Effiziente Quellennutzung und -verwaltung

Anforderung: "Die Verwaltung der Informationsquellen muss derart effizient sein, dass der daraus entstehende Aufwand sehr gering ist – selbst wenn keine oder nur wenige der verwalteten Quellen genutzt werden. Idealerweise ist der Verwaltungsaufwand unabhängig von der Menge der verwalteten Quellen." (S. 43)

Diskussion: Auf konzeptioneller Ebene wird der Aufwand für die lokale Nutzung von Providern bzw. ihren Kontextquellen dadurch minimiert, dass diese Bedarfsabhängig vom Kontextdienst aktiviert werden. Dies setzt natürlich voraus, dass der Provider eine solche Form der Aktivierung unterstützt.

In den vorausgegangen Diskussionen der Anforderungen /A 16/ und /A 17/ hat sich eine mögliche Reduzierung des Verwaltungsaufwandes gezeigt, jedoch auch die Probleme bei dessen Evaluation. Grundsätzlich ist die Verwaltung der Quellen bei proaktiver Verteilung – wie durch die Edge Peers vorgenommen – linear von der Menge der Quellen abhängig, wenngleich die Peer-to-Peer-basierte Verwaltung den Aufwand gleichmäßig auf alle Super Peers verteilt und somit eine signifikante Reduzie-

rung für den einzelnen Peer bewirkt. Eine weitere Reduzierung wäre nur durch vollständige Abkehr von der proaktiven Verteilung der Providerbeschreibungen durch die Edge Peers möglich. Dies führt voraussichtlich zu einer nennenswerten Vergrößerung der Latenz beim Binden geeigneter Provider und somit auch beim Kontextzugriff. Ein solches Verhalten steht allerdings im Gegensatz zu Anforderung /A 4/ und ist somit nicht zu favorisieren.

Bewertung: *teilweise bestanden*
Die Evaluation der verteilten Quellenverwaltung ist analog zu den Anforderungen /A 16/ und /A 17/ nicht möglich.

7. Zusammenfassung und Ausblick

7.1. Zusammenfassung

Um die Ausführung ubiquitärer Software zu unterstützen, wurde im Rahmen dieser Arbeit untersucht, welche Anforderungen durch die Charakteristika der notwendigen Kontextinformationen entstehen. Kontext beschreibt Situationen von Systemen und deren Nutzern. Er entsteht durch die Beobachtung der Umgebungen in denen die Situationen bestehen. Aufgrund dieser Beobachtung i. V. m. der inhärenten Dynamik der Umgebungen ist Kontext dynamisch. Die Kontextquellen, welche sich in der Umgebung befinden und so deren Beobachtung ermöglichen, sind von der Dynamik ebenfalls betroffen, und deshalb für potenzielle Kontextsenken dynamisch verfügbar.

Nur diejenigen Informationen, die relevant für die Situation eines Nutzers sind, sind Kontext. Weil er jedoch erst aus der Situation heraus entsteht, d. h. diese widerspiegelt, kann die Relevanz konkreter Informationen nicht a priori festgestellt werden. Aufgrund der Existenz messbarer Relevanzfaktoren wie Ort und Zeit ist die Feststellung der Relevanz jedoch ad hoc möglich.

Für die Gewinnung von Kontext aus einer Umgebung werden die in ihr verfügbaren Kontextquellen genutzt. Dabei werden Quellen und deren Informationen, die ohnehin Teil einer Umgebung sind, opportunistisch genutzt. Dementsprechend schwankt die Qualität der erfassten Informationen. Um diese Schwankungen zu kompensieren, müssen ubiquitäre Anwendungen die Qualität der Informationen feststellen können. Die Abhängigkeit ubiquitärer Anwendungen von Kontextinformationen

ist nicht existenziell. Statt dessen handelt es sich um Kontextbewusstheit bzw. Kontextbezug – analog des Englischen Begriffs der Context-Awareness.

Aufgrund der vielfältigen Aspekte bei der Beschreibung von Umgebungen kann Kontext ebenso vielfältigen Ursprungs sein. In der vorliegenden Arbeit liegt der Fokus der Betrachtungen auf gemessenem und abgeleitetem Kontext, welcher in erster Linie von Sensoren und Ableitungsschemata stammt.

Die Betrachtung eines komplexen Anwendungsszenarios zeigte, dass die Heterogenität nicht nur die Kontextquellen und -informationen betrifft, sondern auch die Geräte bzw. Computer einer Umgebung. Im Gegensatz zu klassischen Szenarien kann in ubiquitären Szenarien die Menge der Kontextquellen die Menge der Kontextsenken um mehrere Größenordnungen überwiegen. Unabhängig von diesem Verhältnis umfasst das Szenario eine große Anzahl von Geräten, Quellen und Senken, und stellt somit hohe Anforderungen an die Skalierbarkeit des zugrunde liegenden Systems.

Basierend auf diesen Analysen wurden folgende Thesen aufgestellt:

- Zur Verbesserung des Kontextzugriffs ist ein Wechsel des Gegenstandes der Verwaltungsaktivitäten sinnvoll – weg von der Verwaltung von Kontextinformationen, hin zu einer Verwaltung ihrer Kontextquellen. Die Behandlung der dynamischen Verfügbarkeit von Kontextquellen durch transparente, lose Kopplung an die Kontextsenken erhöht dabei die Robustheit der Kontextsenken gegenüber der schwankenden Verfügbarkeit von Kontextinformationen.
- Die vorhandenen Mengen von Kontextquellen und vernetzten Geräten machen eine Abkehr vom Client/Server-Paradigma erforderlich, um durch Skalierbarkeit die Funktionsfähigkeit ubiquitärer Software mit globaler Verteilung zu sichern.

Das Ergebnis der Arbeit ist die Entwicklung und Realisierung eines verteilten Kontextdienstes. Seine Kernfunktionalität ist die verteilte Verwaltung heterogener, dynamisch verfügbarer Kontextquellen zum Ziel des verteilungstransparenten Informationszugriffs durch Kontextsenken. Die Lösung der Problematik besteht aus folgenden Facetten:

1. Definition eines Modells generischer Low- und High-Level Providerkomponenten zur einheitlichen Kapselung heterogener Kontextquellen,
2. Reduzierung des Implementierungsaufwandes für Provider durch Spezifikation eines Frameworks zur generischen Emulation von Zugriffssemantik,
3. Transparente Kopplung von Kontextdienstinstanzen zur zentralen Verwaltung der Kontextquellen mittels Client/Server-Kommunikation,
4. Behandlung der Heterogenität von Endgeräten und ihrer schwankenden Verfügbarkeit bei gleichzeitiger Erreichung von Skalierbarkeit durch Einsatz von KBR in einem hybriden Peer-to-Peer-Netzwerk.

Die prototypische Umsetzung des Gesamtkonzepts, sowie die praktischen Untersuchungen und theoretischen Analysen erbringen den Nachweis der Eignung der entwickelten Konzepte zur Erfüllung der identifizierten Anforderungen.

7.2. Ausblick

Die vorliegende Arbeit zeigt die Komplexität und Vielschichtigkeit der zu lösenden Probleme bei der Entwicklung von Ubiquitous Computing Middleware. Im Rahmen der Arbeit wurde aufgrund der Gefahr proprietärer, domänenspezifischer Lösungsansätze absichtlich darauf verzichtet, weitere Metamodelle zur Repräsentation von Kontextinformationen zu entwickeln. Jedoch gibt es zahlreiche Aspekte beim Kontextzugriff, deren Modellierung bisher nicht oder nur ansatzweise untersucht ist.

Eine Weiterentwicklung der Modelle zur Beschreibung von Informationsquellen hinsichtlich quantifizierbarer Klassifikationsmerkmale wie Ausdrucksstärke und Flexibilität der Schnittstelle, Ortsgebundenheit, etc. eröffnet i. V. m. entsprechender Unterstützung durch Middleware und Frameworks die Nutzung der spezifischen Funktionen und Ressourcen heterogener Kontextquellen anstatt deren generische Kapselung.

Bei der Entwicklung von ubiquitärer Software ist es problematisch, die Vielfalt der erforderlichen Kontextinformationen einzugrenzen, und die darauf basierende Adaptivität zu spezifizieren. Entwickler und Systembetreiber benötigen Informationen, welche das tatsächliche Verhalten ubiquitärer Software wiedergeben, und Rückschlüsse über den Erfolg von Adaptionsmaßnahmen erlauben. Dazu muss Ubiquitous Computing Middleware generische Unterstützung für Adaptivität bieten und in der Lage sein, darauf basierende Software zu beobachten.

Um den Zugriff der Super Peers auf Providerbeschreibungen flexibler zu gestalten, sind Ansätze zu untersuchen, welche die Abbildung einer schwankenden Menge von Merkmalen mit Werten wechselnden Formats auf den begrenzten Schlüsselraum es KBR erlauben.

Für eine nachvollziehbare Evaluation existierender und zukünftiger Systeme wäre ein anwendungsnaher Vergleichstest wünschenswert, welcher charakteristische Szenarien ubiquitärer Software umsetzt. Um diese zu identifizieren, ist zuerst eine detaillierte Analyse und Klassifikation vorhandener ubiquitärer Systeme erforderlich. Die vorliegende Arbeit zeigt jedoch schon die Schwierigkeiten bei der notwendigen Begriffsbestimmung.

A. XML Schemata für Nachrichteninhalte

Für das Verständnis der folgenden Schemabeschreibungen sind Kenntnisse in der Definition und dem Aufbau von XML Schemata wünschenswert, aber nicht zwingend erforderlich.

A.1. Nachrichten für proaktive Verteilung

Dieses Kapitel enthält den Quelltext der XSD für den Inhalt der Nachrichten zur proaktiven Verteilung von Providerbeschreibungen (Kap. 5.5). Die Zeilen 13 bis 23 zeigen die Wurzelelemente der Nachrichteninhalte, welche durch die anschließenden Typdefinitionen weiter strukturiert werden. Die Semantik der Nachrichteninhalte wurde bereits zuvor erläutert.

Quelltext A.1: XSD der Nachrichteninhalte zur proaktiven Verteilung von Providerbeschreibungen

```
1 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ↘
→xmlns="http://www.inf.tu-dresden.de/mocawi" ↘
→targetNamespace="http://www.inf.tu-dresden.de/mocawi">
  <xs:annotation>
```

```
    <xs:documentation xml:lang="en">
      Schema for node reachability, provider management,
      and context information access messages as well as
      all information exchanged between distributed context
      service instances with proactive exchange of provider
      descriptions.
      Copyright 2008 Thomas Hamann
    </xs:documentation>
  </xs:annotation>
  <xs:element name="signon" type="mocawi.node.signon.type↘
→"/>
  <xs:element name="signoff" type="mocawi.node.signoff.type↘
→"/>
  <xs:element name="ping" type="mocawi.node.ping.type"/>
  <xs:element name="pong" type="mocawi.node.pong.type"/>
  <xs:element name="updPro" type="mocawi.provider.update.↘
→type"/>
  <xs:element name="reqCtxt" type="mocawi.context.request.↘
→type"/>
  <xs:element name="respCtxt" type="mocawi.context.respond.↘
→type"/>
  <xs:element name="subCtxt" type="mocawi.context.subscribe↘
→.type"/>
  <xs:element name="confCtxt" type="mocawi.context.confirm.↘
→type"/>
  <xs:element name="updCtxt" type="mocawi.context.update.↘
→type"/>
  <xs:element name="unsubCtxt" type="mocawi.context.↘
→unsubscribe.type"/>
  <xs:complexType name="mocawi.node.signon.type">
    <xs:sequence>
      <xs:element name="node" type="mocawi.node.id.type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.node.signoff.type">
    <xs:sequence>
      <xs:element name="node" type="mocawi.node.id.type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.node.ping.type">
    <xs:sequence>
```

```
      <xs:element name="seq" type="xs:positiveInteger"/>
      <xs:element name="to" type="mocawi.node.id.type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.node.pong.type">
    <xs:sequence>
      <xs:element name="seq" type="xs:positiveInteger"/>
      <xs:element name="from" type="mocawi.node.id.type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.node.id.type">
    <xs:choice>
      <xs:element name="jxtaPeerId" type="mocawi.node.id.↘
      →jxta.type"/>
      <xs:element name="ipAddress" type="mocawi.node.id.ip.↘
      →type"/>
    </xs:choice>
  </xs:complexType>
  <xs:simpleType name="mocawi.node.id.jxta.type">
    <xs:restriction base="xs:anyURI">
      <xs:pattern value="([uU][rR][nN]:[jJ][xX][tT][aA]:)↘
      →.+\-.+"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="mocawi.node.id.ip.type">
    <xs:choice>
      <xs:element name="ipV4Address" type="mocawi.node.id.↘
      →ipv4.type"/>
      <xs:element name="ipV6Address" type="mocawi.node.id.↘
      →ipv6.type"/>
    </xs:choice>
  </xs:complexType>
  <xs:simpleType name="mocawi.node.id.ipv4.type">
    <xs:restriction base="xs:string">
      <xs:pattern value↘
      →="((0|1[0-9]{0,2}|2([0-4][0-9]?|5[0-5]?|[6-9])↘
      →?|[3-9][0-9]?)\.)↘
      →{3}(0|1[0-9]{0,2}|2([0-4][0-9]?|5[0-5]?|[6-9])↘
      →?|[3-9][0-9]?)"/>
    </xs:restriction>
  </xs:simpleType>
```

```
   <xs:simpleType name="mocawi.node.id.ipv6.type">
     <xs:restriction base="xs:string">
70     <xs:pattern value="(([0-9]|[a-f]|[A-F]){1,4}:)↘
       →{7}([0-9]|[a-f]|[A-F]){1,4}"/>
     </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="mocawi.provider.update.type">
     <xs:sequence>
75     <xs:element name="usableProviders" type="mocawi.↘
       →provider.hosted.set.type"/>
       <xs:element name="unusableProviders" type="mocawi.↘
       →provider.id.hosted.set.type"/>
       <xs:element name="seq" type="xs:positiveInteger"/>
     </xs:sequence>
   </xs:complexType>
80 <xs:complexType name="mocawi.provider.hosted.set.type">
     <xs:sequence maxOccurs="unbounded">
       <xs:element name="hostedProviders" type="mocawi.↘
       →provider.hosted.type"/>
     </xs:sequence>
   </xs:complexType>
85 <xs:complexType name="mocawi.provider.hosted.type">
     <xs:sequence>
       <xs:element name="providerHost" type="mocawi.node.id.↘
       →type"/>
       <xs:element name="providers" type="mocawi.provider.↘
       →set.type"/>
     </xs:sequence>
90 </xs:complexType>
   <xs:complexType name="mocawi.provider.set.type">
     <xs:sequence maxOccurs="unbounded">
       <xs:element name="provider" type="mocawi.provider.↘
       →type"/>
     </xs:sequence>
95 </xs:complexType>
   <xs:complexType name="mocawi.provider.type">
     <xs:sequence>
       <xs:element name="providerId" type="mocawi.provider.↘
       →id.type"/>
       <xs:element name="provides" type="mocawi.context.↘
       →pattern.set.type"/>
```

```
100         <xs:element name="consumes" type="mocawi.context.↘
            →pattern.set.type" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
        <xs:complexType name="mocawi.provider.id.type">
          <xs:sequence>
105         <xs:element name="sourceId" type="xs:string"/>
            <xs:element name="localHost" type="mocawi.node.id.↘
            →type" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
        <xs:complexType name="mocawi.context.pattern.set.type">
110       <xs:sequence maxOccurs="unbounded">
            <xs:element name="contextPattern" type="mocawi.↘
            →context.pattern.type"/>
          </xs:sequence>
        </xs:complexType>
        <xs:complexType name="mocawi.context.pattern.type">
115       <xs:sequence>
            <xs:choice>
              <xs:element name="entityType" type="mocawi.context.↘
              →entity.type.type"/>
              <xs:element name="attributeType" type="mocawi.↘
              →context.attribute.type.type"/>
              <xs:element name="associationType" type="mocawi.↘
              →context.association.type.type"/>
120         </xs:choice>
            <xs:element name="restriction" type="xs:string" ↘
            →minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
        <xs:simpleType name="mocawi.context.entity.type.type">
125       <xs:restriction base="xs:string"/>
        </xs:simpleType>
        <xs:complexType name="mocawi.context.attribute.type.type↘
        →">
          <xs:sequence>
            <xs:element name="parentType" type="mocawi.context.↘
            →parent.type.type"/>
130         <xs:element name="attributeName" type="xs:string"/>
          </xs:sequence>
```

```
  </xs:complexType>
  <xs:complexType name="mocawi.context.parent.type.type">
    <xs:choice>
135     <xs:element name="entityType" type="mocawi.context.↘
      →entity.type.type"/>
      <xs:element name="attributeType" type="mocawi.context↘
      →.attribute.type.type"/>
    </xs:choice>
  </xs:complexType>
  <xs:simpleType name="mocawi.context.association.type.type↘
  →">
140   <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:complexType name="mocawi.provider.id.hosted.set.type↘
  →">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="hostedProviderIds" type="mocawi.↘
      →provider.id.hosted.type"/>
145   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.provider.id.hosted.type">
    <xs:sequence>
      <xs:element name="providerHost" type="mocawi.node.id.↘
      →type"/>
150     <xs:element name="providerIds" type="mocawi.provider.↘
      →id.set.type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.provider.id.set.type">
    <xs:sequence maxOccurs="unbounded">
155     <xs:element name="providerId" type="mocawi.provider.↘
      →id.type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.context.request.type">
    <xs:sequence>
160     <xs:element name="request" type="mocawi.context.↘
      →pattern.type"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.context.respond.type">
```

```
    <xs:sequence>
165     <xs:element name="requested" type="mocawi.context.↘
        →pattern.type"/>
        <xs:element name="current" type="mocawi.context.↘
        →element.set.type"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="mocawi.context.element.set.type">
170   <xs:sequence maxOccurs="unbounded">
        <xs:element name="contextElement" type="mocawi.↘
        →context.element.type"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="mocawi.context.element.type">
175   <xs:choice>
        <xs:element name="entity" type="mocawi.context.entity↘
        →.type"/>
        <xs:element name="attribute" type="mocawi.context.↘
        →attribute.type"/>
        <xs:element name="association" type="mocawi.context.↘
        →association.type"/>
      </xs:choice>
180 </xs:complexType>
    <xs:complexType name="mocawi.context.entity.type">
      <xs:sequence>
        <xs:element name="entityKey" type="mocawi.context.↘
        →entity.key.type"/>
        <xs:element name="attributes" type="mocawi.context.↘
        →attribute.set.type" minOccurs="0"/>
185     <xs:element name="associations" type="mocawi.context.↘
        →association.set.type" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="mocawi.context.entity.key.type">
      <xs:sequence>
190     <xs:element name="entityType" type="mocawi.context.↘
        →entity.type.type"/>
        <xs:element name="entityId" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="mocawi.context.attribute.type">
```

```
195     <xs:sequence>
          <xs:element name="attributeKey" type="mocawi.context.↘
          →attribute.key.type"/>
          <xs:choice>
            <xs:sequence>
              <xs:element name="value" type="xs:string"/>
200           <xs:element name="javaClass" type="mocawi.context↘
              →.attribute.javaclass.type"/>
              <xs:element name="unit" type="xs:string"/>
            </xs:sequence>
            <xs:element name="attributes" type="mocawi.context.↘
            →attribute.set.type" minOccurs="0"/>
          </xs:choice>
205     </xs:sequence>
      </xs:complexType>
      <xs:complexType name="mocawi.context.attribute.key.type">
        <xs:sequence>
          <xs:element name="parentKey" type="mocawi.context.↘
          →parent.key.type"/>
210       <xs:element name="attributeName" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="mocawi.context.parent.key.type">
        <xs:choice>
215       <xs:element name="entityKey" type="mocawi.context.↘
          →entity.key.type"/>
          <xs:element name="attributeKey" type="mocawi.context.↘
          →attribute.key.type"/>
        </xs:choice>
      </xs:complexType>
      <xs:simpleType name="mocawi.context.attribute.javaclass.↘
      →type">
220     <xs:restriction base="xs:string">
          <xs:pattern value="([a-z][a-z0-9]*\.)*([A-Z][a-z0↘
          →-9]*)+"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:complexType name="mocawi.context.attribute.set.type">
225     <xs:sequence maxOccurs="unbounded">
          <xs:element name="attribute" type="mocawi.context.↘
          →attribute.type"/>
```

```
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.context.association.type">
230   <xs:sequence>
      <xs:element name="associationKey" type="mocawi.↘
      →context.association.key.type"/>
      <xs:element name="sourceEntity" type="mocawi.context.↘
      →entity.key.type"/>
      <xs:element name="targetEntity" type="mocawi.context.↘
      →entity.key.type"/>
    </xs:sequence>
235 </xs:complexType>
  <xs:complexType name="mocawi.context.association.key.type↘
  →">
    <xs:sequence>
      <xs:element name="associationType" type="mocawi.↘
      →context.association.type.type"/>
      <xs:element name="associationId" type="xs:string"/>
240   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.context.association.set.type↘
  →">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="association" type="mocawi.context.↘
      →association.type"/>
245   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.context.subscribe.type">
    <xs:sequence>
      <xs:element name="subscribe" type="mocawi.context.↘
      →pattern.type"/>
250   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="mocawi.context.confirm.type">
    <xs:sequence>
      <xs:element name="subscribed" type="mocawi.context.↘
      →pattern.type"/>
255     <xs:element name="ticket" type="mocawi.context.ticket↘
      →.type"/>
    </xs:sequence>
  </xs:complexType>
```

```
    <xs:complexType name="mocawi.context.ticket.type">
      <xs:sequence>
260     <xs:element name="host" type="mocawi.node.id.type"/>
        <xs:element name="seqNo" type="xs:positiveInteger"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="mocawi.context.update.type">
265   <xs:sequence>
        <xs:element name="ticket" type="mocawi.context.ticket↘
        →.type"/>
        <xs:element name="events" type="mocawi.context.event.↘
        →set.type"/>
      </xs:sequence>
    </xs:complexType>
270 <xs:complexType name="mocawi.context.event.set.type">
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="contextEvent" type="mocawi.context.↘
        →event.type"/>
      </xs:sequence>
    </xs:complexType>
275 <xs:complexType name="mocawi.context.event.type">
      <xs:choice>
        <xs:element name="entityEvent" type="mocawi.context.↘
        →event.entity.type"/>
        <xs:element name="attributeEvent" type="mocawi.↘
        →context.event.attribute.type"/>
        <xs:element name="associationEvent" type="mocawi.↘
        →context.event.association.type"/>
280   </xs:choice>
    </xs:complexType>
    <xs:complexType name="mocawi.context.event.entity.type">
      <xs:sequence>
        <xs:element name="entityKey" type="mocawi.context.↘
        →entity.key.type"/>
285     <xs:element name="created" type="xs:boolean"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="mocawi.context.event.attribute.type↘
    →">
      <xs:sequence>
```

```
290        <xs:element name="attributeKey" type="mocawi.context.↘
           →attribute.key.type"/>
           <xs:element name="value" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="mocawi.context.event.association.↘
      →type">
295     <xs:sequence>
           <xs:element name="associationKey" type="mocawi.↘
           →context.association.key.type"/>
           <xs:element name="created" type="xs:boolean"/>
        </xs:sequence>
      </xs:complexType>
300   <xs:complexType name="mocawi.context.unsubscribe.type">
        <xs:sequence>
           <xs:element name="ticket" type="mocawi.context.ticket↘
           →.type"/>
        </xs:sequence>
      </xs:complexType>
305 </xs:schema>
```

A.2. Nachrichten für hybriden Austausch

Die Zeilen 11 bis 15 von Quelltext A.2 zeigen die Wurzelelemente der fünf zusätzlichen Nachrichtentypen des hybriden Austauschs von Providerbeschreibungen (Kap. 5.5.5).

Quelltext A.2: XSD der Nachrichteninhalte zum hybriden Austausch von Providerbeschreibungen

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ↘
  →xmlns="http://www.inf.tu-dresden.de/mocawi" ↘
  →targetNamespace="http://www.inf.tu-dresden.de/mocawi">
    <xs:include schemaLocation="lst-messages-proaktiv.xsd"/>
    <xs:annotation>
5     <xs:documentation xml:lang="en">
         Schema extension with additional provider management
         messages for reactive exchange provider descriptions.
         Copyright 2008 Thomas Hamann
      </xs:documentation>
```

```
10    </xs:annotation>
      <xs:element name="reqPro" type="mocawi.provider.request.↘
      →type"/>
      <xs:element name="respPro" type="mocawi.provider.respond.↘
      →type"/>
      <xs:element name="subPro" type="mocawi.provider.subscribe↘
      →.type"/>
      <xs:element name="unsubPro" type="mocawi.provider.↘
      →unsubscribe.type"/>
15    <xs:element name="ackPro" type="mocawi.provider.↘
      →acknowledge.type"/>
      <xs:complexType name="mocawi.provider.request.type">
        <xs:sequence>
          <xs:element name="request" type="mocawi.context.↘
          →pattern.type"/>
        </xs:sequence>
20    </xs:complexType>
      <xs:complexType name="mocawi.provider.respond.type">
        <xs:sequence>
          <xs:element name="usableProviders" type="mocawi.↘
          →provider.hosted.set.type"/>
        </xs:sequence>
25    </xs:complexType>
      <xs:complexType name="mocawi.provider.subscribe.type">
        <xs:sequence>
          <xs:element name="subscribe" type="mocawi.context.↘
          →pattern.type"/>
        </xs:sequence>
30    </xs:complexType>
      <xs:complexType name="mocawi.provider.unsubscribe.type">
        <xs:sequence>
          <xs:element name="unsubscribe" type="mocawi.context.↘
          →pattern.type"/>
        </xs:sequence>
35    </xs:complexType>
      <xs:complexType name="mocawi.provider.acknowledge.type">
        <xs:sequence>
          <xs:element name="seq" type="xs:positiveInteger"/>
        </xs:sequence>
40    </xs:complexType>
    </xs:schema>
```

Literatur

[AMCRC04] Al-Muhtadi, Jalal, Shiva Chetan, Anand Ranganathan und Roy H. Campbell: *Super Spaces: A Middleware for Large-Scale Pervasive Computing Environments.* In: *PerWare2004: Workshop on Middleware Support for Pervasive Computing at the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, März 2004.

[Bar05] Bardram, Jakob E.: *The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications.* In: Gellersen, Hans-Werner, Roy Want und Albrecht Schmidt (Herausgeber): *Pervasive*, Band 3468 der Reihe *Lecture Notes in Computer Science*, Seiten 98–115, Munich, Germany, Mai 2005. Springer.

[BCPR03] Bellifemine, Fabio, Giovanni Caire, Agostino Poggi und Giovanni Rimassa: *JADE – A White Paper.* exp, 3(3):6–19, September 2003. Telecom Italia Lab.

[BD05] Bradley, N. A. und M. D. Dunlop: *Towards a Multidisciplinary Model of Context to Support Context-Aware Computing.* Journal of Human-Computer Interaction, 20(4):403–446, 2005.

[BHH04] Buchholz, Sven, Thomas Hamann und Gerald Hübsch: *Comprehensive Structured Context Profiles (CSCP): Design and Experiences.* In: *2nd IEEE Conference on Pervasive Computing and Communications Workshops (PerCom 2004 Workshops)*, Seiten 43–47, März 2004.

[Bib90] Bibliographisches Institut & F. A. Brockhaus AG (Herausgeber): *Brockhaus-Enzyklopädie*, Band Bd. 12. Kir - Lag. Mannheim: F.A. Brockhaus, 1990.

[BM07] Bardram, Jakob E. und Martin Mogensen: *DOLCLAN - Middleware Support for Peer-to-Peer Distributed Shared Objects.* In: *Distributed Applications and Interoperable Systems (DAIS'07)*, Seiten 119–132, 2007.

[BN04] Becker, Christian und Daniela Nicklas: *Where do spatial context-models end and where do ontologies start? A proposal of a combined approach.* In: Indulska, Jadwiga und David De Roure (Herausgeber): *Proceedings of the First International Workshop on Advanced Context Modelling, Reasoning and Management in conjunction with UbiComp 2004*, Seiten 48–53. University of Southhampton, September 2004.

[Bun06] Bundeszentrale für politische Bildung: *Datenreport 2006 - Zahlen und Fakten über die Bundesrepublik Deutschland*, Band 544. Statistisches Bundesamt, 2006. ISBN 3-89331-675-2.

[CAMCM04] Chetan, Shiva, Jalal Al-Muhtadi, Roy H. Campbell und M. Dennis Mickunas: *A Middleware for Enabling Personal Ubiquitous Spaces.* In: *UbiSys '04: Workshop on System Support for Ubiquitous Computing at the Sixth Annual Conference on Ubiquitous Computing (UbiComp 2004)*, 2004.

[CAMCM05] Chetan, Shiva, Jalal Al-Muhtadi, Roy Campbell und M. Dennis Mickunas: *Mobile Gaia: a middleware for ad-hoc pervasive computing.* In: *Second IEEE Consumer Communications and Networking Conference (CCNC)*, Seiten 223–228, Januar 2005.

[CFJ03a] Chen, Harry, Timothy W. Finin und Anupam Joshi: *An Intelligent Broker for Context-Aware Systems.* In: *Adjunct Proceedings of UbiComp2003*, 2003.

[CFJ03b] Chen, Harry, Timothy W. Finin und Anupam Joshi: *An ontology for context-aware pervasive computing environments.* The Knowledge Engineering Review, 18(3):197–207, 2003.

[CFJ03c] Chen, Harry, Timothy W. Finin und Anupam Joshi: *Semantic Web in a Pervasive Context-Aware Architecture.* Artificial Intelligence in Mobile Systems, 2003.

[CFJ03d] Chen, Harry, Timothy W. Finin und Anupam Joshi: *Using OWL in a Pervasive Computing Broker.* In: *Ontologies in Agent Systems 2003, Proceedings of the Workshop on Ontologies in Agent Systems (OAS 2003) at the 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Seiten 9–16, Melbourne, Australia, Juli 2003.

[CFJ04a] Chen, Harry, Timothy W. Finin und Anupam Joshi: *A Context Broker for Building Smart Meeting Rooms.* In: *Proceedings of the Knowledge Representation and Ontology for Autonomous Systems Symposium*, 2004.

[CFJ04b] Chen, Harry, Timothy W. Finin und Anupam Joshi: *Semantic Web in the Context Broker Architecture.* In: *PerCom*, Seiten 277–286, 2004.

[CH88] Croft, W. Bruce und Gavin D. J. Harper: *Document retrieval systems*, Kapitel Using probabilistic models of document retrieval without relevance information, Seiten 161–171. Taylor Graham Publishing, London, UK, UK, 1988.

[Che76] Chen, Peter Pin-Shan: *The entity-relationship model – toward a unified view of data.* ACM Trans. Database Syst., 1(1):9–36, 1976.

[CHRC01] Cerqueira, Renato, Christopher K. Hess, Manuel Román und Roy H. Campbell: *Gaia: A Development Infrastructure for Active Spaces.* In: *Workshop on Application Models an Programming Tools for Ubiquitous Computing*, September 2001.

[CK00] Chen, Guanling und David Kotz: *A Survey of Context-Aware Mobile Computing Research.* Technischer Bericht, Dartmouth College, 2000.

[CRS07] Conan, Denis, Romain Rouvoy und Lionel Seinturier: *Scalable Processing of Context Information with COSMOS.* In: *Distributed Applications and Interoperable Systems (DAIS'07)*, 2007.

[CvHH+01] Conolly, Dan, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter L. Patel-Schneider und Lynn Andrea Stein: *DAML+OIL (March 2001) Reference Description.* W3C Note, World Wide Web Consortium, Dezember 2001.

[Dar05] Darlagiannis, Vasilios: *Peer-to-Peer Systems and Applications*, Kapitel Hybrid Peer-to-Peer Systems, Seiten 353–366. Band 3485 der Reihe Steinmetz, Ralf und Klaus Wehrle [SW05], 2005.

[Dar06] Dargie, Waltenegus: *A Distributed Architecture for Computing Context in Mobile Devices.* Doktorarbeit, Technische Universität Dresden, 2006.

[Dey00] Dey, Anind K.: *Providing architectural support for building context-aware applications.* Doktorarbeit, Georgia Institute of Technology, 2000.

[Dey01] Dey, Anind K.: *Understanding and Using Context.* Personal and Ubiquitous Computing, 5(1):4–7, 2001.

[DH06] Dargie, Waltenegus und Thomas Hamann: *A Distributed Architecture for Reasoning about a Higher-Level Context.* In: *2nd IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob2006)*, Seiten 268–275, Juni 2006.

[Dou04] Dourish, Paul: *What we talk about when we talk about context.* Personal and Ubiquitous Computing, 8(1):19–30, Februar 2004.

[DZD+03] Dabek, Frank, Ben Y. Zhao, Peter Druschel, John Kubiatowicz und Ion Stoica: *Towards a Common API for Structured Peer-to-Peer Overlays.* In: Kaashoek, M. Frans und Ion Stoica (Herausgeber): *IPTPS*, Band 2735 der Reihe *Lecture Notes in Computer Science*, Seiten 33–44. Springer, 2003.

[Eic07] Eichhorn, Jörg: *Analyse logisch-basierter Reasoner zur Ableitung von Kontextinformationen in mobilen Szenarien.* Großer Beleg, Technische Universität Dresden, 2007.

[ET03] Eidson, John C. und John Tengdin: *IEEE-1588 standard for a precision clock synchronization protocol for networked measurement and control systems and applications to the power industry.* In: *Proceedings of DistribuTECH 2003*, 2003.

[Gos06] GOSLAR, KEVIN: *Ein Integrations- und Darstellungsmodell für verteilte und heterogene kontextbezogene Informationen.* Doktorarbeit, Technische Universität Dresden, 2006.

[GPY05] GU, TAO, HUNG KENG PUNG und JIAN KANG YAO: *Towards a flexible service discovery.* Journal of Network and Computer Applications, 28(3):233–248, 2005.

[GPZ04a] GU, TAO, HUNG KENG PUNG und DA QING ZHANG: *A middleware for building context-aware mobile services.* In: *IEEE 59th Vehicular Technology Conference (VTC 2004-Spring)*, Seiten 2656–2660, Mai 2004.

[GPZ04b] GU, TAO, HUNG KENG PUNG und DA QING ZHANG: *Toward an OSGi-Based Infrastructure for Context-Aware Applications.* IEEE Pervasive Computing, 3(4):66–74, 2004.

[GPZ05] GU, TAO, HUNG KENG PUNG und DA QING ZHANG: *A service-oriented middleware for building context-aware services.* Journal of Network and Computer Applications, 28(1):1–18, 2005.

[HHN06] HAMANN, THOMAS, GERALD HÜBSCH und RENÉ NEUMERKEL: *Specification of the EMODE Runtime Environment.* Technischer Bericht, Technische Universität Darmstadt, 2006.

[HHS08] HAMANN, THOMAS, GERALD HÜBSCH und THOMAS SPRINGER: *A Model-Driven Approach for Developing Adaptive Software Systems.* In: MEIER, RENÉ und TERZIS SOTIRIOS (Herausgeber): *Distributed Applications and Interoperable Systems (DAIS 2008)*, Band 5053 der Reihe *Lecture Notes in Computer Science*, Seiten 196–209. Springer Verlag, 2008.

[HI04] HENRICKSEN, KAREN und JADWIGA INDULSKA: *Modelling and Using Imperfect Context Information.* Technischer Bericht, School of Information Technology and Electrical Engineering, The University of Queensland, 2004.

[HK07] HÜBSCH, GERALD und KAY KADNER: *Multimodal Information Access Across Multiple Devices.* In: *Proceedings of the Mobility Conference 2007 – The International Conference on Mobile Technology, Applications and Systems*, 2007.

[HKL+99] HOHL, FRITZ, UWE KUBACH, ALEXANDER LEONHARDI, KURT ROTHERMEL und MARKUS SCHWEHM: *Nexus – An Open Global Infra-*

structure for Spatial-Aware Applications. Technischer Bericht 1999/02, Universität Stuttgart, 1999.

[Hor95] Hornby, A. S.: *Oxford Advanced Learner's Dictionary of Current English.* Oxford University Press, 5th edition Auflage, 1995.

[Hor02] Horrocks, Ian: *DAML+OIL: A Reason-able Web Ontology Language.* In: *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*, Seiten 2–13, London, UK, 2002. Springer-Verlag.

[HR06] Henricksen, Karen und Ricky Robinson: *A survey of middleware for sensor networks: state-of-the-art and future directions.* In: *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, Seiten 60–65, New York, NY, USA, 2006. ACM.

[HSP+03] Hofer, Thomas, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann und Werner Retschitzegger: *Context-Awareness on Mobile Devices - the Hydrogen Approach.* In: *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, Washington, DC, USA, 2003. IEEE Computer Society.

[Hüb08] Hübsch, Gerald: *Systemunterstützung für interaktorbasierte multimodale Benutzungsschnittstellen.* Doktorarbeit, Technische Universität Dresden, 2008.

[Jia88] Jiang, Bin: *Deadlock detection is really cheap.* SIGMOD Rec., 17(2):2–13, 1988.

[JQSP07] Jiang, Nanyan, Andres Quiroz, Cristina Schmidt und Manish Parashar: *Meteor: A Middleware Infrastructure for Content-based Decoupled Interactions in Pervasive Grid Environments.* Concurrency and Computation: Practice and Experience, 2007.

[JSP06] Jiang, Nanyan, Cristina Schmidt und Manish Parashar: *A Decentralized Content-based Aggregation Service for Pervasive Environments.* In: *Proceedings of the IEEE International Conference on Pervasive Services (ICPS 2006)*, Seiten 203 – 212. IEEE Computer Society Press, Juni 2006.

[KC04] Klyne, Graham und Jeremy J. Carroll: *Resource Description Framework (RDF) : Concepts and Abstract Syntax.* W3C Recommendation, World Wide Web Consortium, Februar 2004.

[KCL00] Kumar, Sanjeev, Philip R. Cohen und Hector J. Levesque: *The Adaptive Agent Architecture: Achieving Fault-Tolerance Using Persistent Broker Teams.* In: *ICMAS '00: Proceedings of the Fourth International Conference on MultiAgent Systems (ICMAS-2000)*, Seite 159, Washington, DC, USA, 2000. IEEE Computer Society.

[KLL+97] Karger, David, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine und Daniel Lewin: *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web.* In: *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, Seiten 654–663, New York, NY, USA, 1997. ACM Press. Chairman-F. Tom Leighton and Chairman-Peter Shor.

[KW05] Karl, Holger und Andreas Willig: *Protocols and Architectures for Wireless Sensor Networks.* John Wiley & Sons, Juni 2005.

[Lia99] Liang, Sheng: *The Java Native Interface – Programmer's Guide and Specification.* Addison-Wesley, Juni 1999.

[LMS05] Leach, Paul J., Michael Mealling und Rich Salz: *A Universally Unique IDentifier (UUID) URN Namespace.* RFC 4122, The Internet Society – Network Working Group, Juli 2005.

[LQS05] Leclercq, Matthieu, Vivien Quéma und Jean-Bernard Stefani: *DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware.* IEEE Distributed Systems Online, 6:9, September 2005.

[Mat08] Matossian, Vincent: *jxta-meteor – Projekt-Webseite.* https://jxta-meteor.dev.java.net/, zuletzt besucht am 7. September 2008.

[MCP+02] Mainwaring, Alan, David Culler, Joseph Polastre, Robert Szewczyk und John Anderson: *Wireless sensor networks for habitat monitoring.* In: *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, Seiten 88–97, New York, NY, USA, 2002. ACM Press.

[Obj08] Object Management Group, Inc.: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* Technischer Bericht formal/2008-04-03, OMG, November 2008.

[OMG07] Object Management Group, Inc.: *Ontology Definition Metamodel.* OMG Adopted Specification ptc/2007-09-09, OMG, November 2007.

[OSG07a] OSGi Alliance: *OSGi Service Platform Core Specification – Release 4, Version 4.1.* Technischer Bericht, OSGi Alliance, April 2007.

[OSG07b] OSGi Alliance: *OSGi Service Platform Service Compendium – Release 4, Version 4.1.* Technischer Bericht, OSGi Alliance, April 2007.

[oSN02] Standards, National Institute of und Technology (NIST): *Federal Information Processing Standards Publication 180-2 : Secure Hash Standard*, August 2002.

[Pla08] PlanetLab Consortium: *PlanetLab – Projekt-Webseite.* http://www.planet-lab.org/, zuletzt besucht am 7. September 2008.

[Poh05] Pohl, Christoph: *Adaptive Caching of Distributed Components.* Doktorarbeit, Technische Universität Dresden, 2005.

[PSHH04] Patel-Schneider, Peter F., Patrick Hayes und Ian Horrocks: *OWL Web Ontology Language Semantics and Abstract Syntax.* W3C Recommendation, World Wide Web Consortium, Februar 2004.

[RC00] Román, Manuel und Roy H. Campbell: *Gaia: enabling active spaces.* In: *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, Seiten 229–234, New York, NY, USA, 2000. ACM.

[RD01] Rowstron, Antony I. T. und Peter Druschel: *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems.* In: Guerraoui, Rachid (Herausgeber): *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Seiten 329–350, London, UK, 2001. Springer-Verlag.

[RdF06] Riva, Oriana und Cristiano di Flora: *Contory: A Smart Phone Middleware Supporting Multiple Context Provisioning Strategies.* In: *ICDCSW '06: Proceedings of the 26th IEEE International Conference-Workshops on Distributed Computing Systems*, Seite 68, Washington, DC, USA, 2006. IEEE Computer Society.

[RFH+01] Ratnasamy, Sylvia, Paul Francis, Mark Handley, Richard Karp und Scott Schenker: *A scalable content-addressable network.* In: *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Seiten 161–172, New York, NY, USA, 2001. ACM Press.

[RHC+02] Román, Manuel, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell und Klara Nahrstedt: *A middleware infrastructure for active spaces.* IEEE Pervasive Computing, 1(4):74–83, Oktober-Dezember 2002.

[RHR+01] Román, Manuel, Christopher K. Hess, Anand Ranganathan, Pradeep Madhavarapu, Bhaskar Borthakur, Prashant Viswanathan, Renato Cerqueira, Roy H. Campbell und M. Dennis Mickunas: *GaiaOS: An Infrastructure for Active Spaces.* Technischer Bericht UIUCDCS-R-2001-2224, UILU-ENG-2001-1731, University of Illinois at Urbana-Champaign, Champaign, IL, USA, Mai 2001.

[SAW94] Schilit, Bill, Norman Adams und Roy Want: *Context-Aware Computing Applications.* In: *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Seiten 85–90, Santa Cruz, CA, USA, Dezember 1994. IEEE Computer Society Press.

[SDA99] Salber, Daniel, Anind K. Dey und Gregory D. Abowd: *The Context Toolkit: Aiding the Development of Context-Enabled Applications.* In: *Proceedings of the SIG CHI Conference on Human Factors in Computing Systems*, Seiten 434–441, 1999.

[SG01] Schmidt, Albrecht und Hans-Werner Gellersen: *Modell, Architektur und Plattform für Informationssysteme mit Kontextbezug.* Inform., Forsch. Entwickl., 16(4):213–224, 2001.

[SKSY06] Springer, Thomas, Kay Kadner, Frank Steuer und Ming Yin: *Middleware Support for Context-Awareness in 4G Environments.* In: *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, Seiten 203–211, Washington, DC, USA, 2006. IEEE Computer Society.

[SMK+01] Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek und Hari Balakrishnan: *Chord: A scalable peer-to-peer lookup service for internet applications.* In: *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Seiten 149–160, New York, NY, USA, 2001. ACM Press.

[Spr04] Springer, Thomas: *Ein komponentenbasiertes Meta-Model kontextabhängiger Adaptionsgraphen für mobile und ubiquitäre Anwendungen - Dissertation.* Doktorarbeit, Technische Universität Dresden, Dresden, Oktober 2004.

[Sto07] STOCK, WOLFGANG G.: *Information Retrieval. Suchen und Finden von Informationen.* Oldenbourg, München, Wien, 2007.

[Sun04] SUN MICROSYSTEMS: *The Real-Time Java Platform.* White paper JSR-001, Sun Microsystems, Inc., Juni 2004.

[Sun07] SUN MICROSYSTEMS: *Connected Limited Device Configuration.* Specification 1.1.1, Sun Microsystems, Inc., November 2007.

[SW05] STEINMETZ, RALF und KLAUS WEHRLE: *Peer-to-Peer Systems an Applications*, Band 3485 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag Berlin Heidelberg, 2005.

[TAA+03] TRAVERSAT, BERNARD, AHKIL ARORA, MOHAMED ABDELAZIZ, MIKE DUIGOU, CARL HAYWOOD, JEAN-CHRISTOPHE HUGLY, ERIC POUYOUL und BILL YEAGER: *Project JXTA 2.0 Super-Peer Virtual Network.* Project documentation, Mai 2003.

[VDW08] VDW, VEREIN DEUTSCHER WERKZEUGMASCHINENFABRIKEN E.V.: *Die deutsche Werkzeugmaschinenindustrie im Jahr 2007.* Branchenreport, März 2008.

[WB06a] WECK, MANFRED und CHRISTIAN BRECHER: *Werkzeugmaschinen 3 – Mechatronische Systeme, Vorschubantriebe, Prozessdiagnose.* Springer-Verlag, 2006. ISBN 978-3-540-22506-5.

[WB06b] WECK, MANFRED und CHRISTIAN BRECHER: *Werkzeugmaschinen 4 – Automatisierung von Maschnen und Anlagen.* Springer-Verlag, 2006. ISBN 978-3-540-22507-5.

[WDC+04] WANG, XIAOHANG, JIN SONG DONG, CHUNG YAU CHIN, SANKA RAVIPRIYA HETTIARACHCHI und DA QING ZHANG: *Semantic Space: An Infrastructure for Smart Spaces.* IEEE Pervasive Computing, 3(3):32–39, 2004.

[Wei91] WEISER, MARK: *The Computer for the Twenty-First Century.* Scientific American, 265:94–104, 1991.

[WGR05] WEHRLE, KLAUS, STEFAN GÖTZ und SIMON RIECHE: *Distributed Hash Tables.* In: STEINMETZ, RALF und KLAUS WEHRLE [SW05], Seiten 79–93.

[YKZ03] YANG, GUIZHEN, MICHAEL KIFER und CHANG ZHAO: *Flora-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web.* In: MEERSMAN, ROBERT, ZAHIR TARI

und DOUGLAS C. SCHMIDT (Herausgeber): *CoopIS/DOA/ODBASE*, Band 2888 der Reihe *Lecture Notes in Computer Science*, Seiten 671–688. Springer, 2003.

[Zad87] ZADEH, LOTFI A.: *Fuzzy sets and applications : selected papers.* Wiley, 1987. ISBN 0-471-85710-6.

[Zim80] ZIMMERMANN, HUBERT: *OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection.* IEEE Transactions on Communications, 28(4):425–432, April 1980.

Abkürzungen

ACL	Access Control List 76
API	Application Programming Interface 70
AR	Associative Rendezvous 86
B	Byte 42
CAD	Computer-Aided Design 38
CAM	Computer-Aided Manufacturing 38
CAN	Content-Addressable Network 135
CAST	Group Anycast and Multicast 135
CDC	Connected Device Configuration 153
CLDC	Connected-Limited Device Configuration 153
CMMS	Computerized Maintenance Management System 35
CoBrA	Context Broker Architecture 68
CORBA	Common Object Request Broker Architecture 71
COSMOS	Context Entities Composition and Sharing 88
C/S	Client/Server 59
DAML-OIL	DARPA Agent Markup Language – Ontology Inference Layer 66

DARPA Defense Advanced Research Projects Agency
DHT Distributed Hash Table 85
DNS Domain Name System 117
DOLR Decentralized Object Location and Routing 135
DPI Distributed Provider Index 136
DREAM Dynamic Reflective Asynchronous Middleware 88
FP Foundation Profile 154
GB Gigabyte 55
GPS Global Positioning System 128
GUI Graphical User Interface 206
IETF Internet Engineering Task Force 184
IP Internet Protocol 116
IPC Inter Process Communication 186
IT Informationstechnik 6
I2R Institute for Infocomm Research, Singapur 63
JADE Java Agent Development Framework 68
Java ME Java Micro Edition 153
Java SE Java Standard Edition 154
JCAF Java Context-Awareness Framework 76
JET Java Emitter Templates 194
JNI Java Native Interface 153
JRE Java Runtime Environment 155
JXTA Juxtapose 86
kB Kilobyte 42
KBR Key-Based Routing 135
KVM Kilobyte Virtual Machine 153
LAN Local Area Network 45
MAN Metropolitan Area Network 46
MB Megabyte 37
MDSD Model-Driven Software Development 84
MOCAWI Mobile Context-Awareness Infrastructure 158

MOF	Meta Object Facility	192
MOF QVT	MOF Query/View/Transformation	194
MPACC	Model-Presentation-Adapter-Controller-Coordinator	71
MSC	Multimodal Services Component	195
MVC	Model-View-Controller	71
NAT	Network Address Translation	184
ODM	Ontoglogy Definition Metamodel	193
OMG	Object Management Group	193
OSGi	Open Services Gateway Initiative	153
OWL	Web Ontology Language	64
OWL-DL	OWL – Description Logics	193
P2P	Peer-to-Peer	73
PB	Personal Basis Profile	154
PDA	Personal Digital Assistant	45
PDSO	Peer-to-Peer Distributed Shared Objects	87
PP	Personal Profile	154
RDF	Resource Description Framework	8
RDQL	Resource Description Query Language	68
RFC	Request for Comments	184
RFID	Radio Frequency Identification	90
RTT	Round Trip Time	119
SFC	Space Filling Curve	85
SLS	Service Locating Service	64
SOCAM	Service-Oriented Context-Aware Middleware	63
TB	Terabyte	55
TPE	Task Process Engine	195
ULCO	Upper Level Context Ontology	64
UML	Unified Modelling Language	192
UOB	Unified Object Bus	71
UPnP	Universal Plug and Play	156
UUID	Universally Unique Identifier	184

VM	Virtual Machine 153
WAN	Wide Area Network 41
WPAN	Wireless Personal Area Network 72
WSN	Wireless Sensor Network 20
XML	Extensible Markup Language 118
XSD	XML Schema Definition 118