Flexible Kommunikation in effizient entwickelten adaptiven vernetzten Dienste- und Gerätesystemen

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r
I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Oliver Dohndorf

Dortmund

2016

Tag der mündlichen Prüfung: 27.04.2016
Dekan: Prof. Dr.-Ing. Gernot A. Fink
Gutachter: Prof. Dr. Heiko Krumm, Prof. Dr. Dirk Timmermann

Kurzfassung

Die fortschreitende Miniaturisierung der IT-Landschaft und die zunehmende Mobilität durch die Ausbreitung von Funkstandards schufen Voraussetzungen um im Sinne des Internet der Dinge Geräte des Alltags miteinander zu vernetzen und so IT-basierte Systeme zu erschaffen, die unterstützend in Situationen des menschlichen Lebens eingreifen. Diese Umgebungen werden im allgemeinen als ambiente Systeme bezeichnet. Für die Integration von Geräten und Diensten unterschiedlicher Hersteller und Anwendungsgebiete werden Domänen übergreifende Frameworks benötigt, die dem Nutzer unabhängig von der Hardware das komfortable und effiziente Entwickeln ambienter Systeme ermöglicht.

Die vorliegende Arbeit beschreibt dafür die wichtigsten Anforderungen und stellt einige existierende Frameworks vor. Für den Ansatz der *OSGi Remote Services* wird die vom Autor realisierte Middleware Comoros vorgestellt, die den Standard mit dem *Devices Profile for Web Services* kombiniert. Dadurch entsteht eine standardkonforme Lösung, welche die Dynamik der OSGi-Plattform mit der Webservice basierten Kommunikation für Kleinstgeräte kombiniert.

Von dieser Lösung ausgehend wurde Comoros um Bereiche erweitert, die für die Entwicklung verteilter ambienter Systeme notwendig sind. Neben einem dynamischen und komfortablen Ansatz für das Daten-Marshaling umfasst die Comoros-Erweiterung auch eine Event-basierte Kommunikation und eine komfortable Integration von Altsystemen. Weiterhin wird die Hersteller unabhängige Integration von Geräten in die Service-Plattform beschrieben, die für den Einsatz im IoT-Umfeld eine besondere Bedeutung hat. Um auf wechselnde Anforderungen dynamisch reagieren zu können setzt Comoros zudem etablierte Management-Standards um und kann so an die jeweils gültige Anforderung adaptiert werden.

Um die Realisierung der definierten Anforderungen von Comoros zu belegen wurde eine umfangreiche Evaluierung durchgeführt. Der Fokus dieser Evaluierung liegt dabei auf der Vermessung der Effizienz und Leistungsfähigkeit der Middleware, Eigenschaften, die bei einem Einsatz in Ressourcen beschränkten Umgebungen von besonderem Interesse sind. Zusätzlich wurde auch der Entwicklungskomfort vermessen, der Indikator für eine hohe Benutzerakzeptanz ist.

Inhaltsverzeichnis

1 Einleitung

Die technologische Evolution führte zu einem immer stärker werdenden Trend zur Miniaturisierung von Mikroelektronik. Die IT-Landschaft der 60er und 70er Jahre war maßgeblich geprägt von stationären Großrechnern, gefolgt von der Einführung der PCs in den 80er und 90er Jahren. Heutzutage nehmen Smartphones, gepaart mit der Ausbreitung neuer Funkstandards wie WLAN, Bluetooth, UMTS und LTE, eine wichtige Rolle ein, mit denen der Wunsch nach ständiger Erreichbarkeit und mobiler Kommunikation erfüllt werden kann. Mit Hilfe dieser Microcomputer ist es nun auch möglich unterstützend in Situation des menschlichen Lebens einzugreifen. Ambiente Systeme nehmen hier eine immer wichtigere Rolle ein. Diese sind vernetzte, eingebettete Systeme, die über eine intelligente Sensorik und Aktorik verfügen und weitgehend unabhängig arbeiten. Gegenstände des Alltags werden in diesen Systemen durch die Erweiterung von Kommunikationsfähigkeit und Intelligenz zu so genannten *SmartObjects* [Kor10], die Informationen verarbeiten, speichern und untereinander interagieren. So entstehen intelligente Umgebungen, die den Alltag der Menschen in verschiedenen Bereichen erleichtern können.

Die wichtigsten Domänen betreffen hierbei die Medizintechnik, die Gebäudeautomatisierung und den Automobilsektor. Die Möglichkeiten für Parkplätze, die uns die aktuelle Belegung anzeigen, Transfusionsbeutel, welche die Temperatur überwachen und Fehler melden, Autositze, die sich automatisch auf die Position verschiedener Fahrer einstellen oder Blumentöpfe, die Feuchtigkeit messen und zum Gießen auffordern werden erschaffen. Die Anwendungen sind vielfältig und nahezu unbeschränkt. Diese Umgebung, auch häufig durch das *Internet der Dinge* [Atz10] kategorisiert, bietet eine komplexe und flexible Landschaft mit vielfältigen technischen Herausforderungen.

Die herausragendste Herausforderung betrifft dabei die Kommunikation der Geräte untereinander. Wie der Name verlautbaren lässt, setzt das Internet der Dinge auf die Verwendung von Internet-Technologien, z. B. in der Kommunikation auf das IP-Protokoll,

als gemeinsame Basis [Ray12]. Diese Basis befindet sich im OSI-Referenzmodell [Day83] auf der Vermittlungsschicht und bietet Geräteherstellern auf höheren Schichten, gerade auf der Anwendungsschicht, somit die Möglichkeit spezifische Protokolle zu entwickeln und einzusetzen. Für Unternehmen kann dies ein wichtiges Instrument zur Kundenbindung sein. Gerade in der Gebäudeautomatisierung kann eine Vielzahl von unterschiedlichen Geräten ein Alleinstellungsmerkmal sein und einen Hersteller für Kunden attraktiv machen. Durch ein Hersteller-spezifisches, proprietäres Kommunikationsprotokoll sind nun nur Geräte dieses Herstellers untereinander interoperabel, es findet somit eine Abschottung gegenüber Fremdherstellern statt. Dadurch entstehen Insellösungen, die wiederum für Kunden unattraktiv sind und zugleich der Vision des Internets der Dinge und des *Ubiquitous Computing* [Wei91] widersprechen. Die Vermeidung solcher Insellösungen wird allgemein als Herausforderung angesehen [Edw01].

Im Bereich ambienter Systeme spielen zudem Sensornetzwerke eine wichtige Rolle zur Analyse der Umgebung. Sensoren setzen aber zumeist nicht auf Internet-Technologien zur Kommunikation sondern auf Funk-Standards wie ZigBee oder Bluetooth auf. In dieser Umgebung kann eine Homogenität der Kommunikation erst auf Anwendungsschicht erreicht werden. Die zuvor beschriebenen Probleme von Insellösungen werden dadurch also verschärft.

Eine Überwindung dieser Beschränkungen und die damit verbundene Überlappung von unterschiedlichen Domänen bieten Herstellern aber auch Chancen zur Erweiterung des Geschäftsmodells. Ambiente Systeme können mit einer homogenen Kommunikation und mit standardisierten Interfaces der Geräte und Sensoren einfach erstellt werden. Empfehlungen für den Alltag könnten so einfach durch Informationen aus unterschiedlichen Domänen generiert werden. Das Fahrrad meldet, dass diese Woche erst 20 km mit dem Rad gefahren wurden, zu wenig für die Gesundheitsvorsorge. Wetterinformationen sagen Regen für die nächsten Tage voraus, und die Parkplätze an der Arbeitsstelle sind schon alle belegt. So gibt es eine Empfehlung mit dem Rad zur Arbeit zu fahren. Diese neuen Möglichkeiten, kombiniert mit der Kostenersparnis durch eine standardisierte Kommunikation, kann innovativen Herstellern als Anreiz zur Überwindung von Insellösungen dienen.

1.1 Problemstellung und Zielsetzung

Die Dissertation beschreibt die Untersuchung, Umsetzung und Evaluierung der folgenden vierteiligen These:

> (1) Bei der Entwicklung von ambienten Anwendungen innerhalb verteilter Geräte- und Service-Systeme können Insellösungen durch den Einsatz einer *Serviceorientierten Architektur* [Erl05] auf Basis von akzeptierten und offenen Standards vermieden werden. (2) Verteilte ambiente Systeme unterliegen einer Reihe von speziellen Eigenschaften, wie z.B. einer hohen Dynamik und Effizienz, die bereits innerhalb der für die Kommunikation relevanten Middleware berücksichtigt werden müssen. (3) Damit diese, die Entwicklung verteilter Anwendung unterstützende Middleware, eine hohe Akzeptanz erhält, muss die Integration von bestehenden Altkomponenten gesichert sein, und

sowohl ein hoher Entwicklungskomfort als auch eine hohe Produktqualität umgesetzt werden. (4) Der entwickelte Ansatz Comoros setzt genau diese Anforderungen um und unterstützt den Benutzer bei der Entwicklung von verteilten Anwendungen in heterogenen Systemlandschaften auf Basis einer SOA.

Der erste Punkt umfasst die Festlegung der Technologien für die Entwicklung verteilter Anwendungen. Hier werden die Vorzüge einer komponentenorientierten Softwareentwicklung aufgegriffen und die *OSGi*-Service-Plattform [The12b] als Basis verwendet. Bei der Einbettung der ursprünglich lokalen OSGi-Plattformen in eine verteilte Umgebung wird das SOA-Konzept weiter angewendet und das *Devices Profile for Web Services (DPWS)* [Nix09b] als Kommunikationsprotokoll verwendet. Die Einhaltung des SOA-Paradigmas verhindert Inselbildungen und erlaubt neuen Geräten und Diensten somit die einfache Integration in bestehende Landschaften. Für eine homogene Kommunikationslandschaft sind allerdings nicht nur offene Protokolle und Schnittstellen notwendig, auch die versendeten Daten spielen eine wichtige Rolle. Die Heterogenität von Hardware und Betriebssystemen, sowie die Heterogenität von Programmiersprachen bedingen individuelle Konzepte zur Darstellung von Datentypen. Hersteller-spezifische, proprietäre Datenformate verschärfen das Problem. Um nicht an dieser Stelle wieder in die Inselwelten zurückzufallen, müssen offene, Plattform unabhängige externe Datenformate, wie die *Extensible Markup Language (XML)* [Bra08] für die Kommunikation Akzeptanz finden.

Im zweiten Punkt werden die speziellen Anforderungen ambienter Systeme betrachtet. In der medizinischen Domäne beispielsweise haben ambiente Systeme die Aufgabe Patienten und deren Umgebung zu überwachen und in notwendigen Situationen helfend einzugreifen. Dabei sind Sensoren und Aktoren möglichst unsichtbar am Körper und im Umfeld integriert. Um diese Anforderung zu realisieren, können meist keine leistungsstarken Rechner verwendet werden, sondern lediglich möglichst kleine, eingebettete Systeme. Folglich muss auch die Kommunikation zwischen den Knoten einer verteilten Anwendung möglichst effizient gestaltet werden. Das DPWS, die Umsetzung einer Webservice-basierten Kommunikation für Kleinstgeräte, dient als Basis für die effiziente Kommunikation. Weiterhin sind ambiente Systeme einer hohen Dynamik unterworfen, nicht nur verfügbare Services können kontinuierlich wechseln, auch Anforderungen an das System können sich zur Laufzeit ändern. Die Adaptierbarkeit der Middleware ist folglich unumgänglich.

Punkt Drei der These behandelt die Nutzer-Akzeptanz der entwickelten Lösung. Um eine Chance für den produktiven Einsatz zu haben, muss eine Software-Komponente eine hinreichende Produktqualität aufweisen, einen hohen Entwicklungskomfort bieten und zusätzlich eine einfache Integration in bestehende Umfelder realisieren.

Für die Umsetzung und den Nachweis der ersten drei Teile der These wurde ws4d.Comoros, eine Middleware für die Kommunikation in verteilten ambienten Systemen entworfen und implementiert. Abbildung 1.1 zeigt abstrakt eine durch Comoros aufgespannte verteilte Service- und Geräte-Infrastruktur mit den verwendeten Technologien. Über die Implementierung der OSGi-Remote-Service-Spezifikation werden die Grenzen einer einzelnen Java Virtual Machine mittels des DPWS als Kommunikationsprotokoll überwunden.

Auch native Geräte anderer Technologien können einfach in das System integriert werden.

Im Anschluss an diese Entwicklung wurde die Middleware ausführlich experimentell evaluiert.

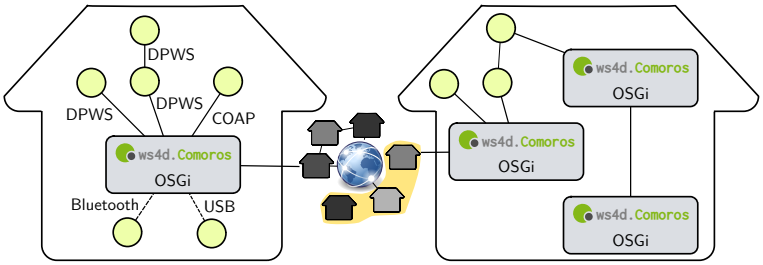


Abbildung 1.1: Verteilte, durch Comoros aufgespannte Infrastruktur

Bis zum Erscheinen dieser Arbeit existiert kein bekannter Ansatz, der die aufgestellten Anforderungen vollständig erfüllt. Zwar existieren Realisierungen für verteilte OSGi-basierte Systeme, diese sind jedoch nicht auf die speziellen Anforderungen ambienter Systeme zugeschnitten. Die mit dem Comoros-Ansatz neuartigen Merkmale können wie folgt zusammengefasst werden:

- Umsetzung einer verteilten auf OSGi basierten SOA unter Verwendung von DPWS als Kommunikationsprotokoll. Daraus entsteht ein System, in dem OSGi- und DPWS-Komponenten ohne zusätzlichen Implementierungsaufwand miteinander interagieren können. Verteilte Anwendungen können so auf einfache Weise die Vorteile beider Technologien anwenden.
- Vollständige Realisierung einer Event-basierten Kommunikation in diesem Umfeld.
- Berücksichtigung Ressourcen beschränkter Umgebungen bei der Umsetzung einer verteilten OSGi-basierten Kommunikation.
- Vermeidung von Insellösungen durch Standardkonformität des Entwurfs.
- Eine hohe Flexibilität erlaubt der Middleware auf wechselnde Anforderungen zu reagieren. Der Austausch von Komponenten, die Konfiguration von Service-Bindungen zur Laufzeit und die Erweiterung um zusätzliche Kommunikationsprotokolle ermöglichen die komplikationslose Umsetzung unterschiedlichster Anwendungsfällen ambienter Systeme. Wohl definierte Schnittstellen zur Überwachung und zur Konfiguration des Systems bieten dazu die technischen Voraussetzungen.
- Altkomponenten, die nicht explizit für ein verteiltes Umfeld entwickelt wurden können unverändert Teil der verteilten ambienten Anwendung werden, wodurch eine einfache Integration in ein bestehendes Umfeld gewährleistet wird.

1.2 Strukturierung der Arbeit

Zur Bearbeitung der aufgezeigten Problemstellung ist die Dissertation in insgesamt neun Kapitel unterteilt. Die Strukturierung ist in Abbildung 1.2 grafisch dargestellt, dabei sind eigene Arbeiten deutlich gekennzeichnet.

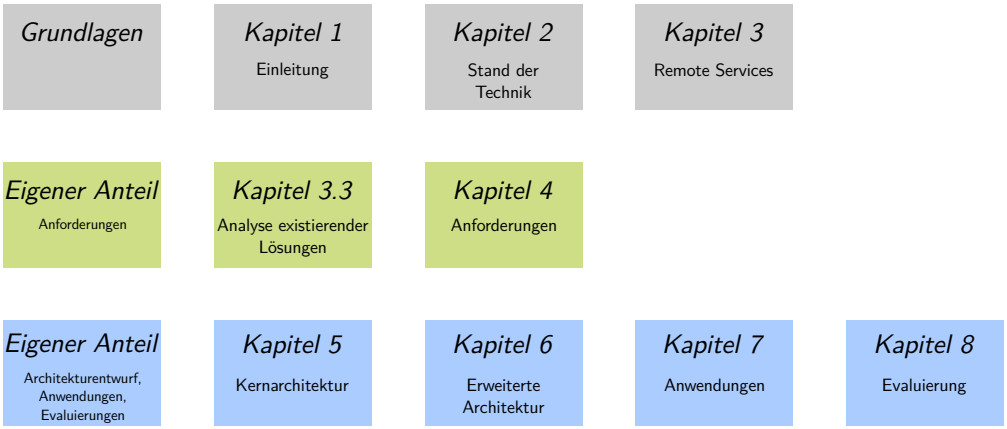


Abbildung 1.2: Struktur der Arbeit mit Darstellung des eigenen Anteils

Nach der Einleitung führt das Kapitel 2 den Leser in die Grundlagen und den Stand der Technik des betrachteten Problemumfeldes ein. Kapitel 3 widmet sich ganz dem Kerngebiet der Arbeit, den OSGi-Remote-Services. Hier werden Standards und existierende Lösungen vorgestellt.

Kapitel 3.3 leitet den eigenen Anteil der Arbeit ein. An dieser Stelle werden die existierenden Lösungen analysiert und offene Probleme dargestellt. Nach dieser Problemanalyse werden in Kapitel 4 die daraus entstehenden Anforderungen an die Comoros-Middleware präsentiert. Die Kapitel 5 und 6 enthalten den ausführlichen Architekturentwurf, wobei Kapitel 5 die Umsetzung der OSGi-Remote-Service-Spezifikation mittels des DPWS beschreibt und Kapitel 6 neue Aspekte einführt, die aus den Anforderungen entstanden sind aber grundsätzlich nicht durch die Spezifikation abgedeckt werden. Kapitel 7 präsentiert mögliche Anwendungsfelder, während Kapitel 8 die fertige Entwicklung hinsichtlich der aus der These abgeleiteten Fragestellungen evaluiert. Obwohl jedes Kapitel mit einem Zwischenfazit endet, werden in Kapitel 9 noch einmal alle wesentlichen Ergebnisse der Dissertation zusammengefasst und ein Ausblick in mögliche weitere Arbeiten gegeben.

2 Stand der Technik

Dieses Kapitel beschäftigt sich im wesentlichen mit den architektonischen Grundlagen, die für die weitere Betrachtung von verteilten Dienste- und Geräte-Systemen von Bedeutung sind. Die wichtigsten Themen werden dabei in einer kompakten Darstellung beleuchtet und bieten durch Quellenangaben Einstiegspunkte für tiefergehende Betrachtungen.

Das Kapitel behandelt dabei folgende Aspekte:

- In welchen Bereich der vielfältigen Informatik-Landschaft sind verteilte Dienste- und Geräte-Systeme einzuordnen?
- Welche Technologien werden aktuell in diesem Bereich eingesetzt?
- Wer sind die Treiber für technische Neuentwicklungen und Standardisierungen?
- Wie können komplexe verteilte Systeme verwaltet werden?

2.1 Technologien für verteilte Geräte- und Service-Systeme

Die Entwicklung verteilter ambienter Anwendungen beinhaltet viele Themengebiete und Herausforderungen. Abbildung 2.1 gibt einen Überblick über die abgedeckten Bereiche und illustriert deren Zusammenspiel. Die Kommunikation der physikalischen Geräte ist als unterste Schicht integraler Bestandteil ambienter Systeme. Komponenten-Plattformen und lokale SOAs umspannen einen weiteren Bereich, sowie die Integration von Sensoren und Geräten in dieses Umfeld. Service-orientierte Architekturen dienen anschließend als Basis für Domänen-spezifische Anwendungen. Bei der Entwicklung dieser Applikationen müssen Ressourcenbeschränkungen berücksichtigt und eine hohe Softwarequalität sichergestellt werden. Neben den Herausforderungen in der Kommunikation, die bereits in Kapitel 1 dargelegt wurden, müssen weitergehende Fragestellungen in der Verwaltung von ambienten Systemen untersucht werden, die sich über alle Schichten erstrecken. Ambiente Systeme

können eine große Komplexität aufweisen, so dass eine effiziente Verwaltung – der Austausch und die Aktualisierung von Komponenten zur Laufzeit – notwendig ist.

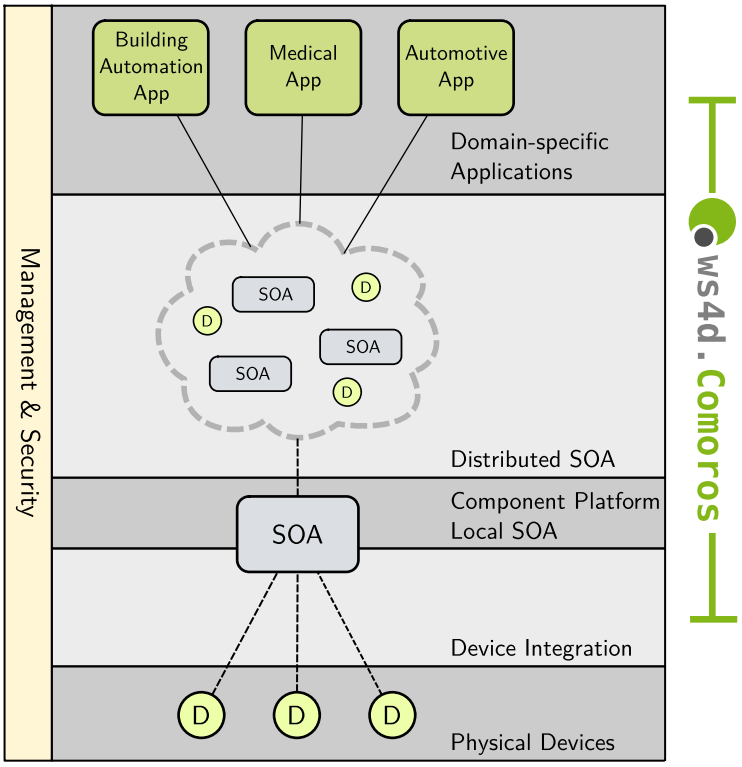


Abbildung 2.1: Überblick über den Bereich des Stands der Technik

Der Comoros Ansatz betrifft die in Abbildung 2.1 gekennzeichneten Bereiche. Die Grundlagen und der Stand der Technik dieser Bereiche wird in den nachfolgenden Abschnitten behandelt.

2.1.1 Standardisierungsgremien

Kompatibilität, Sicherstellung von Interoperabilität, ein einfacher Austausch von Informationen, die Vergleichbarkeit von Qualität und Leistung oder auch die Kostenersparnis, all das ist Motivation für die Entwicklung und die Einhaltung von Standards. Aus dieser Motivation heraus haben sich im Laufe der Zeit verschiedene Organisationen gegründet um in ihren jeweiligen Bereichen, Standards und Normen zu entwickeln und festzulegen.

Die wichtigsten Standardisierungsgremien im Kontext von Internettechnologien sind dabei die *International Organization for Standardization (ISO)*, das *Institute of Electrical and Electronics Engineers (IEEE)*, die *Internet Engineering Task Force (IETF)*, die *Organization for the Advancement of Structured Information Standards (OASIS)*, das *World Wide Web Consortium (W3C)* und die *Object Management Group (OMG)*.

1984 veröffentlichte die ISO mit dem OSI-Referenzmodells [Day83] einen der für die Kommunikation im Internet wichtigsten Standards. Anhand des OSI-Referenzmodell können auch die Zuständigkeitsbereiche der weiteren Organisationen wie in Abbildung 2.2 zu sehen gegliedert werden.

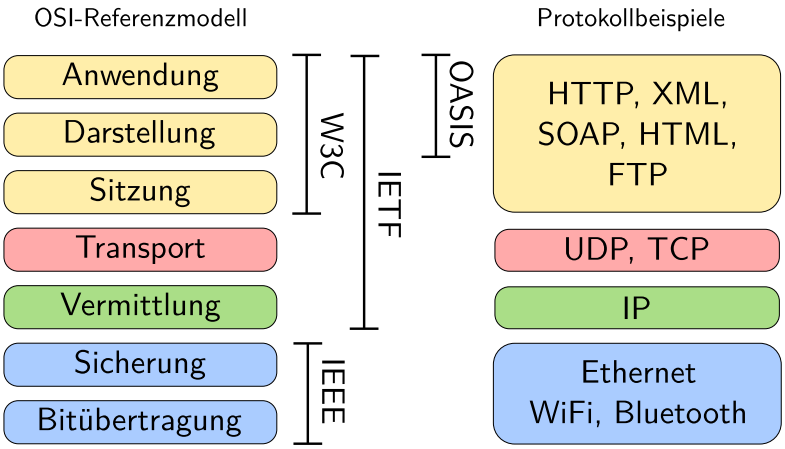


Abbildung 2.2: Überblick und Einordnung der verschiedenen Standardisierungsgremien

Die IEEE definiert dabei Lösungen vornehmlich auf den Netzzugriffsschichten. Entgegen des OSI-Referenzmodells unterteilt die IEEE die Sicherungsschicht in die zwei Unterschichten *Logical Link Control (LLC)* und *Media Access Control (MAC)*. Die bekanntesten von der IEEE definierten Protokolle auf diesen Schichten sind unter den Namen Ethernet, Bluetooth und Wireless Local Area Network (WLAN) bekannt.

Die 1986 gegründete IETF befasst sich mit Internetstandards auf den höher liegenden Schichten, insbesondere mit der Standardisierung der im Internet eingesetzten Kommunikationsprotokolle. Dazu gehören beispielsweise das Internet Protokoll (IP) auf der Vermittlungsschicht, auf der Transportschicht die Protokolle TCP und UDP, sowie HTTP auf der Anwendungsschicht. Parallel zur IETF organisiert die *Internet Research Task Force (IRTF)* Forschungen zur langfristigen Entwicklung des Internets.

Die inkonsistente Nutzung der Technologien HTTP, HTML und URI veranlasste Tim Berners-Lee dazu die Spezifikationen zu diesen Technologien von der IETF standardisieren zu lassen. Eine dazu gegründete Arbeitsgruppe konnte keine relevanten Ergebnisse erzielen. Um die Gefahr der Unterteilung des Webs abzuwehren und dessen universellen Charakter beizubehalten gründete Berners-Lee das W3C. Zusätzlich zu der Homogenisierung der Anwendungsschicht untersuchte das W3C das Problem der Datenrepräsentation und veröffentlichte die bekannten Lösungen Hypertext Markup Language (HTML), XML und die SOAP-basierten Webservices.

1993 gründete sich das Konsortium *SGML Open* und befasste sich fortan mit der Standard Generalized Markup Language (SGML). Mit der Standardisierung von XML durch die IETF und der immer zunehmenden Bedeutung dieser Protokollfamilie änderte sich der Fokus des Konsortiums und schließlich auch der Name zum heutigen OASIS. OASIS befasst sich heute vornehmlich mit e-Business und Webservice-Standards und veröffentlichte auch die für diese Arbeit relevante Lösung, das Devices Profile for Web Services (DPWS), einer Spezifikation von Webservice-Standards für eingebettete Systeme.

Außerhalb des OSI-Referenzmodells liegen die Arbeiten der OMG. Ursprünglich arbeitete das Gremium im Bereich der verteilten Objektsysteme, aus der das bekannte CORBA hervorgegangen ist. Heute liegt der Fokus auf Modellierungs-bezogenen Standar-

disierungen. Die bekanntesten Lösungen sind hierbei die Interface Definition Language (IDL), die Unified Modeling Language (UML) und die Object Management Architecture (OMA).

Als Mitglied von OASIS liefert die Arbeitsgruppe Rechnernetze und verteilte Systeme (RvS) der Technischen Universität Dortmund aktive Arbeiten in dem Arbeitsbereich, in dem das DPWS spezifiziert wird. Weiterhin ist die RvS-Gruppe Teil der WS4D-Initiative, einer Plattform zur Verbreitung von Forschungsergebnissen aus dem Bereich der Webservices für Geräte. Alle Implementierungen, die in dieser Arbeit entstanden sind, wurden im Rahmen von WS4D als Open-Source-Projekt veröffentlicht.

2.1.2 Kommunikation physikalischer Geräte und Sensoren

Vernetzte Geräte und Sensoren bilden Rechnernetze, die je nach Leistungsfähigkeit in unterschiedliche Klassen eingeteilt werden. Die Einteilung der relevanten Klassen ist in Abbildung 2.3 zu sehen. Der Oberbegriff der Sensornetze (engl. *Wireless Sensor Networks (WSN)*) kann wiederum in zwei Klassen unterteilt werden. Das *Body Area Network (BAN)* findet insbesondere im eHealth-Bereich Anwendung [Jov05]. Direkt am Körper getragene Sensoren, z.B. EKG-, Blutdruck- oder SpPO2-Sensoren, ermitteln notwendige Vitaldaten. Werden durch fest installierte Sensoren aus der unmittelbaren Umgebung - beispielsweise Licht- oder Temperatur-Sensoren – Daten erhoben, so spricht man von einem *Personal Area Network (PAN)*. Außerhalb der Klasse der Sensornetzwerke befindet sich das *Local Area Network (LAN)*.

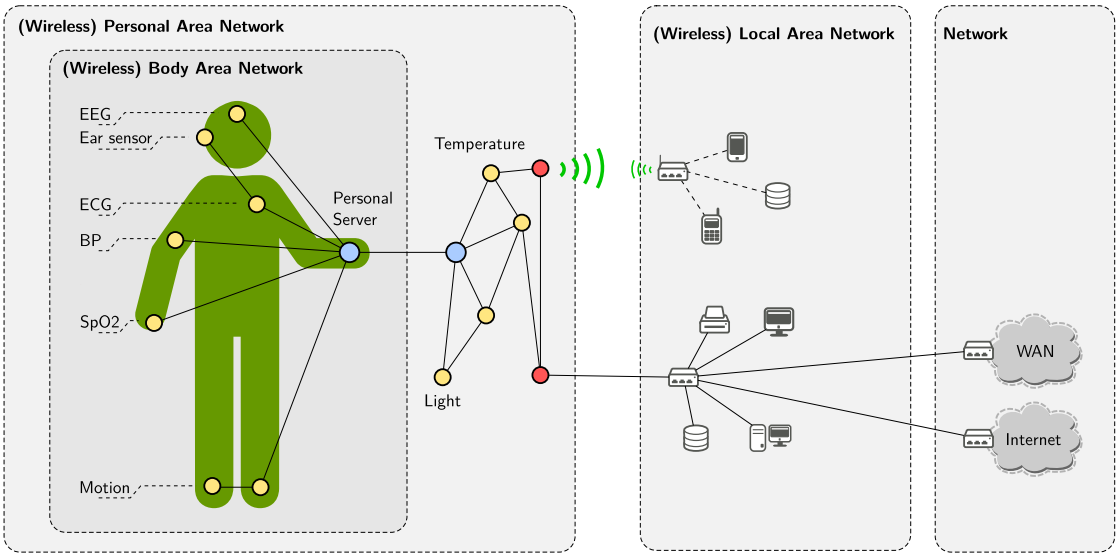


Abbildung 2.3: Überblick über verschiedene Netzwerkklassen

Innerhalb dieser Typen von Rechnernetzen gelten verschiedene Eigenschaften und Anforderungen, die sich auf die verwendeten Kommunikationsprotokolle auswirken. Die Unterteilung der Netzwerktypen betrifft ausschließlich die Bitübertragungs- und die Sicherungsschicht im OSI-Referenzmodell, die darüberliegenden Schichten bleiben unberührt.

Local Area Network

Ein Local Area Network wird typischerweise im Heim- und Unternehmensbereich eingesetzt. Die einzelnen Knoten sind weder in ihrer Größe noch in ihrer Leistungsfähigkeit begrenzt. So können hier beispielsweise leistungsfähige Personal Computer, Datenbank-Server, Drucker oder Sensoren eingesetzt werden. Die Verbindung der Knoten erfolgt über Kabel oder über Funktechnologien, man spricht dann von einem *Wireless Local Area Network (WLAN)*. Wurden früher Protokolle wie ARCNET oder Token Ring in LANs eingesetzt, so sind heute das Ethernet und, im kabellosen Betrieb, das WiFi-Protokoll etabliert.

LANs bilden auf den unteren Schichten eine Sterntopologie, deren Teilnehmer über einen Switch miteinander gekoppelt sind. Router verbinden die einzelnen Netzwerke untereinander. Dabei werden heutzutage die Eigenschaften von Switches, Routern und drahtlosen Access Points zumeist in einem Gerät vereint, so dass alle Netzwerkteilnehmer sich in einem Subnetz befinden, unabhängig davon, ob eine drahtlos, oder eine drahtgebundene Vernetzung vorliegt.

Personal Area Network

Da in einem LAN zumeist keine Energie sparenden Eigenschaften vorhanden sind, wurde für Sensornetzwerke ein neuer Netzwerktyp eingeführt. Personal Area Networks beschreiben ein Netz von Kleinstgeräten die in kurzen Entfernungen (0,2m - 50m) miteinander verbunden sind. Die Übertragungsraten sind deutlich geringer und bewegen sich zwischen 3 MBit/s und 10 MBit/s. Gerade die kabellose Vernetzung der Knoten, dann als *Wireless Personal Area Networks (WPAN)* bezeichnet werden, von der IEEE in verschiedenen Arbeitsgruppen untersucht.

- IEEE 802.15.1 beschäftigt sich mit dem Funkübertragungsprotokoll Bluetooth.
- IEEE 802.15.2 regelt die Zusammenarbeit mit IEEE 802.11 (WLAN).
- IEEE 802.15.3 untersucht höhere Übertragungsraten für PANs.
- IEEE 802.15.4 entwickelt ein energieeffizientes Übertragungsverfahren für geringe Übertragungsraten.

Knoten innerhalb eines PANs werden nicht notwendigerweise über eine Sterntopologie miteinander verbunden, Sensornetzwerke bilden häufig ein vermaschtes Netz.

Body Area Network

Das Body Area Network ist ein spezialisiertes PAN, in dem alle beteiligten Sensoren und Aktoren am Körper getragen werden. Das BAN wird innerhalb der Arbeitsgruppe IEEE 802.15.6 standardisiert. Die Reichweite dieses Netzwerktyp liegt bei 2 Metern, die Übertragungsraten sind mit denen eines PAN identisch, es werden sogar zum größten Teil die gleichen Kommunikationsprotokolle eingesetzt.

Die in [Che11a] beschriebene Architektur von BANs unterteilt die Kommunikation in drei Ebenen, die *intra-BAN*, die *inter-BAN* und die *extra-BAN*.

Die *intra-BAN* definiert die Kommunikation innerhalb des BAN. Teilnehmer in diesem Netzwerk sind Sensoren, leistungsfähigere Knoten, so genannte Central Processors, und der Personal Server (PS). Die verschiedenen Möglichkeiten der Vernetzung sind in

Abbildung 2.4 abgebildet. In zwei Varianten sammelt der Personal Server die Rohdaten der Sensoren und übermittelt die Daten über Access Points an verarbeitende Systeme. Je nach Leistungsfähigkeit des PS können Daten vorbearbeitet und anschließend übermittelt werden. Weitere Varianten sind die direkte Kommunikation zwischen Sensor-Knoten und Access Point und eine zweischichtige Kommunikation über den Central Processor.

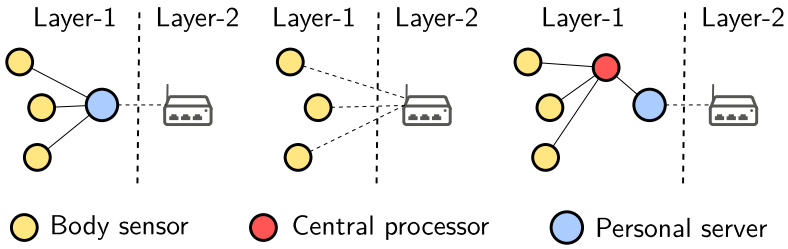


Abbildung 2.4: Varianten der Vernetzung innerhalb eines BAN

Die Kommunikation zwischen der intra-BAN-Schicht und einem oder mehreren Access Points wird als inter-BAN definiert. Das extra-BAN umfasst schließlich die weiteren Netzwerktypen.

Protokolle der Netzwerkklassen

Jede Netzwerkklasse stellt unterschiedliche Anforderungen an das Kommunikationsprotokoll hinsichtlich der Leistungsfähigkeit, des Energieverbrauchs und – bei kabellosen Verbindungen – hinsichtlich der Baugröße des Funkmoduls. Die wichtigsten Protokolle sind im folgenden aufgeführt.

Ethernet Der bekannteste Vertreter der Zugriffsschicht ist sicherlich die Ethernet-Protokollfamilie, die in der Arbeitsgruppe IEEE 802.3 standardisiert wird. Ethernet findet sich vornehmlich in Umgebungen, in denen keinerlei Ressourcenlimitierungen vorhanden sind und hohe Bandbreiten benötigt werden. In den Kabel gebundenen Übertragungstechnologien werden Übertragungsraten von 10 MBit/s bis 1000 MBit/s realisiert.

WiFi - IEEE 802.11 Die IEEE-Norm 802.11, auch Wireless LAN oder WiFi genannt, beschreibt eine Kommunikation in Funknetzwerken und ist die drahtlose Variante des Ethernets. Beide Technologien werden in vielen Infrastrukturen parallel eingesetzt. Derzeit können Übertragungsraten von bis zu 600 MBit/s erreicht werden. Die weite Verbreitung und hohe Akzeptanz der WiFi-Technologie führt sowohl zu einer stetigen Weiterentwicklung, es ist eine Erhöhung der Bandbreite auf bis zu 6933 MBit/s geplant, als auch zu einer Spezialisierung der Technologie. So existiert z.B. eine Variante speziell für die Kommunikation zwischen Fahrzeugen.

IEEE 802.15.4 Das bekannteste Übertragungsprotokoll für WPAN ist in der IEEE-Norm 802.15.4 spezifiziert. Die Hauptmerkmale dieses Protokolls liegen in der Energieeffizienz, den geringen Herstellungskosten und einer hohen Skalierbarkeit. Diese Skalierbarkeit wird durch die Einführung unterschiedlicher Knoten erreicht, die in unterschiedlichen

Topologien angeordnet werden können. Die *Full Function Devices (FFD)* sind Geräte mit einem vollen Funktionsumfang und erhöhter Leistungsfähigkeit, die mit allen anderen Knoten kommunizieren können. *Reduced Function Devices (RFD)* enthalten nur einen verminderten Funktionsumfang, so dass diese Knoten nur mit FFDs kommunizieren können. Allerdings sind diese Knoten energiesparender und können kostengünstiger konstruiert werden. Ein einzelner FFD übernimmt die Rolle des PAN-Koordinator, der das dazugehörige Netz von anderen 802.15.4-Netzen in Reichweite abgrenzt. Die Knoten können, wie in Abbildung 2.5 zu sehen, sternförmig oder in einer Peer-to-Peer-Struktur angeordnet werden.

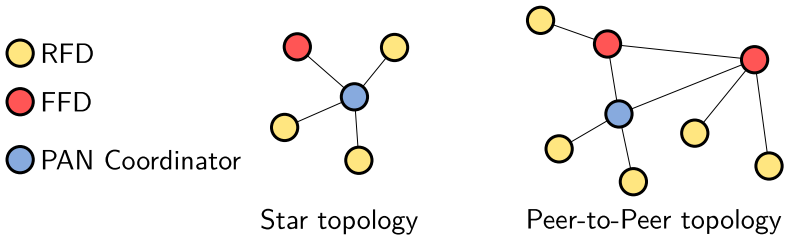


Abbildung 2.5: Topologien von IEEE 802.15.4-Netzen

Der Standard behandelt nur die beiden untersten Schichten des OSI-Referenzmodells. Das Routing und eine Anwendungsschnittstelle sind nicht geregelt.

ZigBee ZigBee ist ein Standard für Funknetze, der auf den Standard IEEE 802.15.4 aufsetzt und dessen Protokollstapel die Vermittlungs-, die Sicherheits- und die Anwendungsschicht ergänzt. Dadurch gibt es analog zu den FFDs, den RFDs und dem PAN-Koordinator ZigBee-Router, ZigBee-Endgerät und einen ZigBee-Koordinator. ZigBee wird in allen Klassen der Wireless Sensor Networks eingesetzt, also sowohl in WPANs als auch in WBANs.

Bluetooth Das weit verbreitete Bluetooth ist in verschiedene Klassen unterteilt, die sich hinsichtlich von Leistungsfähigkeit und Energieverbrauch unterscheiden. In Klasse 1 können Reichweiten von bis zu 100 Metern erreicht werden. Dabei bestehen Bluetooth-Netzwerke aus einer geringen Teilnehmeranzahl. Gerade einmal 8 aktive Teilnehmer bilden ein so genanntes Piconetz, in dem bis zu 255 weitere inaktive Teilnehmer geparkt werden können. Maximal 10 Piconetze können ein Scatternetz bilden. Die Kommunikation innerhalb der Netze ist nach dem Master/Slave-Prinzip aufgebaut. Ein Master pro Netz steuert die Kommunikation und vergibt Sendeslots an die Slaves. Der Master kann wiederum Slave in einem anderen Netz sein.

2.1.3 Komponentenorientierte Entwicklung und lokale Service-orientierte Architekturen

Die fokussierte Entwicklung von Softwaresystemen, das heißt eine Konzentration auf die Implementierung von Geschäftslogiken ohne die Neuentwicklung von Basisfunktionalitäten durch Wiederverwendung von bereits Entwickeltem, ist ein Paradigma, das in der Softwaretechnik schon lange erforscht wird. Die komponentenbasierte Softwareentwicklung hat diese Forschung geprägt und liefert einen Ansatz, in dem umfassende

Anwendungen in flexibler Weise aus einzelnen, von verschiedenen Seiten aus gelieferten Software-Komponenten zusammengesetzt werden.

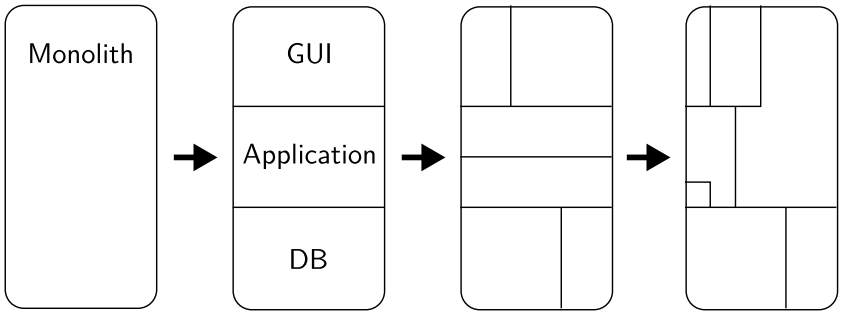


Abbildung 2.6: Entwicklung der Strukturierung von Softwaresystemen [Gru00]

Abbildung 2.6 veranschaulicht den Wandel der Softwareentwicklung im Laufe der Zeit. Prägten in den Anfängen monolithisch strukturierte Systeme die Software-Welt, wurden anschließend die Systeme immer feingranularer strukturiert. Begonnen mit der Einführung geschichteter Architekturen, setzte die horizontale Aufteilung, also die Unterteilung in logisch zusammengehörende Teile, den Trend fort, bis die Aufteilung in Komponenten erreicht wurde [Gru00].

Dabei ist der Ansatz der Komponentenstrukturierung zwar ein neuer, aber aus älteren Wurzeln gewachsener Ansatz. Die Idee, die Komplexität von Software durch Dekomposition in Module überschaubar werden zu lassen beschrieb David Parnas bereits 1972 [Par72]. In der Komponentenstrukturierung ist allerdings der Aspekt der kommerziellen herstellerübergreifenden Vermarktung von Komponenten verstärkt. Dieser neuartige Ansatz hat dabei schnell eine hohe Akzeptanz und Marktrelevanz erzielt [Szy02]. So wurden in den letzten Jahren umfassende Komponenten-Frameworks entwickelt, wie z.B. das OSGi-Framework und die Enterprise Java Beans.

Die Grundzüge der komponentenbasierten Softwareentwicklung lassen sich schnell erläutern. Die *Komponente* ist zentraler Bestandteil des Paradigmas, eine festgelegte Definition existiert für sie aber nicht. Dennoch herrscht über viele Eigenschaften einer Komponente Einigkeit. Softwarekomponenten sind Einheiten, die eine bestimmte Funktionalität realisieren und geschaffen wurden, um aus ihnen Software zusammenzusetzen (aus dem Lat. von 'componens' - das Zusammensetzen). Dazu kommunizieren die Komponenten untereinander über fest spezifizierte Schnittstellen, die Implementierung ist aber verborgen. Alle Abhängigkeiten zu ihrer Umgebung sind in einer Spezifikation ausdrücklich beschrieben. Diese Eigenschaft erlaubt es unterschiedlichsten Herstellern Komponenten unabhängig voneinander zu entwickeln und zu vertreiben, die dann – auch von dritten Parteien – zu Softwaresystemen zusammengesetzt werden können.

Die Größe einer Komponente ist nicht festgelegt und kann daher in einem breiten Spektrum liegen. So ist es sogar möglich Komponenten aus Kompositionen weiterer Komponenten zu erzeugen. Es sollte allerdings darauf geachtet werden, dass sich die Entwicklung einer Komponente aus betriebswirtschaftlichen Gesichtspunkten noch lohnt und dabei nicht das komponentenbasierte Paradigma ad absurdum führt. So darf die Komponente nicht zu klein sein und sich so auf einen zu kleinen und speziellen Anwender-

kreis beschränken und andererseits nicht so groß, dass die Komponente monolithischen Strukturen nahe kommt.

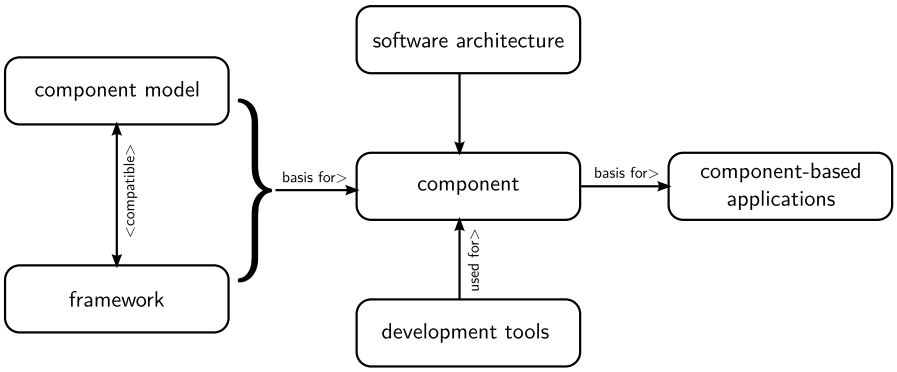


Abbildung 2.7: Das Zusammenspiel von Komponenten, Komponentenmodell und Frameworks [Gru00]

Eine Komponente alleine ist aber wertlos. Abbildung 2.7 zeigt das Zusammenspiel von Komponenten, Komponentenmodellen und Frameworks, die zusammen die komponentenbasierte Softwareentwicklung ausmachen. Das *Komponentenmodell* definiert die Anforderungen und Konventionen, die eine Komponente sowohl syntaktisch als auch semantisch erfüllen muss. Dazu gehört neben der Typisierung von Komponenten auch die Festlegung von Interaktionsmodellen. Diese Modelle beinhalten unter anderem die Beschreibung wie Komponenten untereinander und mit dem Framework interagieren. Weitere Regeln betreffen Topologie-Einschränkungen. Es wird beispielsweise definiert, welche Komponenten auf welche Weise untereinander kommunizieren dürfen und wie viele Clients eine Komponente gleichzeitig aufrufen dürfen. Eine weitere Klasse von Interaktionen behandelt den Lebenszyklus von Komponenten, das Ressourcen-Management und Fragen zur Persistenz.

Um die Komponentenmodelle durchzusetzen wird innerhalb eines jeden Komponentenstandards ein *Framework* zur Verfügung gestellt. Ein Framework wird definiert durch eine Menge von Klassen und deren Beziehungen untereinander. Die vorhandenen Klassen bieten Dienste an, die sich z. B. mit dem Lebenszyklus von Komponenten befassen oder mit dem Registrieren und Finden von Komponenten im System. Verschiedene Frameworks enthalten ganze Datenbankschichten oder Schichten zur einfachen Erstellung von grafischen Oberflächen. Um die Basisfunktionalität zu erweitern, ist es dem Benutzer möglich neue Klassen in das Framework einzuhängen. Zusätzlich verfügt ein Framework über eine Laufzeitumgebung, so dass die Komponenten direkt eingesetzt werden können.

Komponentenbasierte Ansätze verfolgen bei der Systembildung zumeist das Konzept der *Megaprogrammierung* [Boh92; Wie92], bei der nur noch Probleme der Koordination von Komponenten zu lösen sind. Es gibt also in der komponentenbasierten Softwareentwicklung Personen, die Komponenten entwickeln, und solche, die Anwendungen entwickeln. In der Regel kann diese strikte Unterteilung aber nicht eingehalten werden, da es für die Anwendungsentwicklung meistens notwendig ist zusätzliche individuelle Komponenten zu bauen oder Modifikationen an bestehenden Komponenten vorzunehmen. Um den

Anteil aber möglichst gering zu halten, sind Komponenten zumeist parametrisierbar, sie können also zur Laufzeit konfiguriert werden, und können so wesentlich flexibler eingesetzt werden.

Das Zusammensetzen der Komponenten zu einem Softwaresystem, also die Komposition, unterliegt ebenfalls Strukturen. Bachmann et al. [Bac00] definieren unterschiedliche Formen der Komposition, die sich grob in drei Gruppen unterteilen lassen. Kompositionen auf Applikationsebene (Komponente - Komponente), auf Systemebene (Komponente - Framework) und auf Ebene der Interoperation (Framework - Framework). Die wichtigsten Typen der Komposition sind dabei die folgenden:

- *Component Deployment:* Komponenten müssen innerhalb eines Frameworks installiert werden, bevor sie ausgeführt werden können. Dazu ist im Komponentenmodell ein Interface definiert, das die Komponenten implementieren müssen. Auf diese Weise kann das Framework die Dienste der Komponente verwalten.
- *Framework Deployment:* Frameworks können innerhalb anderer Frameworks verteilt werden.
- *Simple Composition:* Komponenten im gleichen Framework können miteinander verknüpft werden. Der Interaktionsmechanismus wird von Seiten des Frameworks zur Verfügung gestellt.

Da es keine einheitliche Definition für den Entwurf eines Komponentenmodells gibt, haben sich im Laufe der Zeit verschiedene Komponentenstandards in unterschiedlichen Anwendungsbereichen gebildet. Lediglich erwähnt werden sollen hier das *CORBA-Komponentenmodell* [Obj06] und das *Component Object Model (COM/DCOM)* [Tho97]. Es folgt eine kurze Beschreibung der Enterprise Java Beans und eine ausführliche Vorstellung der für diese Arbeit besonders relevanten OSGi-Service-Plattform.

Enterprise Java Beans

Dem Paradigma der komponentenbasierten Softwareentwicklung folgend ist die Idee der *Enterprise Java Beans (EJB)* im Rahmen der J2EE-Technologie Komponenten zur Verfügung zu stellen. Dadurch muss der Benutzer keinen infrastrukturellen Code selber schreiben und kann den Fokus auf die Geschäftslogik lenken. EJB sind für die Entwicklung und den Betrieb von Enterprise-Applikationen ausgelegt. Diese Art von Applikationen zeichnen sich durch eine große Anzahl von Benutzern, heterogene Systemlandschaften und hohes Lastaufkommen aus.

Die Funktionalität der EJB-Komponenten wird innerhalb der Java-EE-Applikationsserver realisiert, welche die zentrale Kommunikationseinheit in heterogenen Systemlandschaften bilden. Der Applikationsserver ist ein großer Container, der wiederum eine Menge kleiner Container beinhaltet. Hier ist insbesondere der EJB-Container zu erwähnen, der die Laufzeitumgebung für die EJB-Komponenten anbietet. Im Sinne eines Frameworks innerhalb des Paradigmas zur komponentenbasierten Entwicklung überwacht dieser Container die Ausführung der Komponenten und bietet verschiedene Infrastrukturdienste an. Diese umfassen die Persistierung, Transaktionen, Steuerung von konkurrierenden Zugriffen, die Durchsetzung von Zugriffsrechten, die Steuerung des Lebenszyklus und die

Instantiierung der Komponenten. Die einzelnen EJB-Komponenten sind typisiert, es wird zwischen Session-Beans, Message-Driven-Beans und Persistent-Entitys unterschieden. Abbildung 2.8 veranschaulicht die EJB-Architektur.

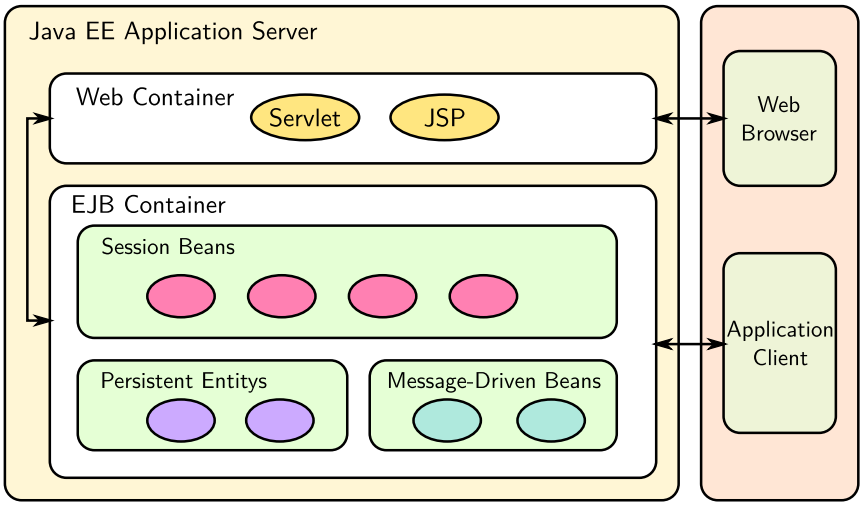


Abbildung 2.8: Java-EE-Server und Container

OSGi

Die OSGi-Spezifikation [The12b] folgt dem Paradigma der komponentenbasierten Softwareentwicklung und definiert ein dynamisches, serviceorientiertes Komponentenmodell für Java. Es handelt sich um eine Software-Plattform, die es ermöglicht voneinander unabhängige Softwarekomponenten (Bundles) und Dienste (Services) dynamisch zu integrieren. Zur Laufzeit können die Komponenten im Framework installiert, gestartet, gestoppt und deinstalliert werden, ohne dass die Plattform als Ganzes neu gestartet werden muss. Die einzelnen Komponenten kommunizieren über die Services miteinander. Auf diese Weise können komplexe Anwendungen über die Komposition der Komponenten realisiert werden.

Die Spezifikation der OSGi-Service-Plattform wird durch die *OSGi-Alliance* realisiert. Die OSGi-Alliance ist ein Non-Profit-Industriekonsortium, das 1999 gegründet wurde und aus mehr als 100 Mitgliedern aus unterschiedlichen Branchen besteht. Darunter befinden sich renommierte Unternehmen wie die Deutsche Telekom, IBM, Oracle oder SAP. Intern gliedert sich die Allianz in verschiedene Arbeitsgruppen, so genannte *expert groups*. Unter den Themenschwerpunkten *Residential, Enterprise, Mobile, Vehicle und Core Platform* werden die Spezifikationen der Standards vorangetrieben. Dabei profitieren die Unternehmen von der transparenten Zusammenarbeit und dem Wissensaustausch innerhalb der gesamten Allianz was die Branchen übergreifende Verbreitung von OSGi fördert.

Das erste Release der OSGi-Spezifikation wurde als Version R1 im Jahr 2000 veröffentlicht, die aktuelle Version stammt aus dem Jahr 2012 und ist die mittlerweile fünfte Hauptversion (R5). Implementierungen der Spezifikation existieren mittlerweile von mehreren unabhängigen kommerziellem Herstellern und auch als Open-Source-Projekte.

Das OSGi-Framework Das OSGi-Framework ist die Basiskomponente der OSGi-Service-Plattform und basiert auf dem in Abbildung 2.9 dargestellten Schichtenmodell. Jede Schicht gruppiert zusammengehörige Aufgaben:

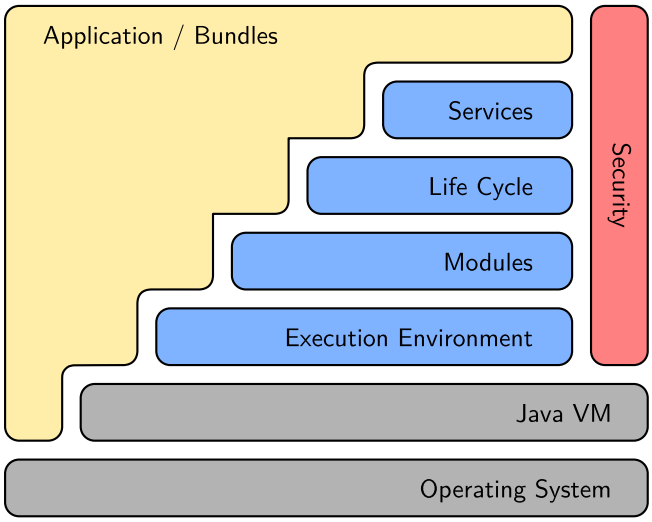


Abbildung 2.9: Logische Schichten im OSGi-Framework [The12b]

- *Execution Environment:* Das OSGi-Framework ist auf unterschiedlichen Plattformen lauffähig. Das gewählte Execution Environment definiert welche Klassen und Methoden innerhalb der Plattform zur Verfügung stehen müssen.
- *Module-Schicht:* Innerhalb der Module-Schicht werden die Bundles als Komponenten beziehungsweise grundlegende Modularisierungseinheiten definiert. Weiterhin werden Regeln bezüglich der Sichtbarkeit und des Bundle-übergreifenden Zugriffs auf Ressourcen anderer Bundles festgelegt.
- *Lifecycle-Schicht:* Jedes Bundle ist innerhalb der OSGi-Service-Plattform einem Lebenszyklus unterworfen. Die Zustände und Zustandsübergänge innerhalb dieses Zyklus werden in dieser Schicht definiert. Darauf aufbauend werden Funktionen zum Installieren, Starten, Stoppen, Aktualisieren und Deinstallieren von Bundles angeboten.
- *Service-Schicht:* Bundles kommunizieren über Services miteinander. Die Service-Schicht spezifiziert wie Services innerhalb der Plattform registriert, aufgefunden und genutzt werden können.
- *Security-Schicht:* Die Sicherheitsschicht basiert auf der *Java 2 Security Architecture* und erweitert vorhandene Mechanismen um OSGi-spezifische Sicherheitsanforderungen. Diese betreffen den Umgang mit signierten Bundles und das Einschränken von Ausführungsrechten und Zugriffsrechten auf Bundle-, Service-, und Package-Ebene.

Über den dargestellten Schichten definiert die Framework-Architektur noch fünf so genannte Framework-Services. Diese Services realisieren die Basisfunktionalität des

Frameworks und bieten Schnittstellen für die Implementierung von Management Agents und werden weniger in der Anwendungsentwicklung verwendet. Die Funktionen umfassen die Verwaltung der Abhängigkeiten und der Start-Level der einzelnen Bundles, sowie Funktionen bezüglich der Umsetzung von Sicherheitsaspekten.

Durch die Framework-Architektur ergeben sich einige Hauptmerkmale für die OSGi-Technologie, welche die Vorteile für die Benutzung ausmachen. Diese können zusammenfassend wie folgt benannt werden:

- *Modularisierung:* Die Modularisierung dient der Komplexitätsreduzierung bei der Entwicklung von Anwendungssystemen. Durch die Entwicklung von Komponenten die stets nur gegen Schnittstellen von Services arbeiten, werden diese Komponenten entkoppelt und können so autark implementiert werden.
- *Abhängigkeitsmanagement:* Damit ein Anwendungssystem aus einzelnen Modulen zusammengesetzt werden kann, stehen die Module in Bezug zueinander und greifen auf Ressourcen von jeweils anderen Modulen zu. Das Abhängigkeitsmanagement organisiert diese Abhängigkeiten auf Package- und Bundle-Ebene.
- *Versionierung:* Schnittstellen und Bundles können im OSGi-Framework versioniert werden. Auf diese Weise können unterschiedliche Versionen eines Bundles parallel im Framework betrieben werden. Das zuvor benannte Abhängigkeitsmanagement kann sich auch auf spezielle Versionen beziehen.
- *Hot Depoyment:* Unter Hot Deployment versteht man die Möglichkeit im laufenden Betrieb Bundles zu installieren, deinstallieren und zu aktualisieren. Damit ist OSGi die ideale Plattform zur Entwicklung von hochverfügbaren Anwendungen, da einzelne Komponenten ausgetauscht werden können ohne den Gesamtbetrieb zu unterbrechen.
- *Remote Management:* Zur entfernten Verwaltung des Frameworks, beispielsweise zur Durchsetzung des Hot Deployments, existieren fest definierte Schnittstellen.

Zur Erläuterung der Architektur und zum besseren Verständnis der aufgeführten Merkmale der OSGi-Service-Plattform werden im Folgenden die einzelnen Schichten der Framework-Architektur detaillierter erläutert.

Die Execution-Environments Das OSGi-Framework ist so spezifiziert, dass es auf den unterschiedlichsten Plattformen lauffähig ist. Dazu gehören unter anderem die Java Standards Edition (JSE) und die Java Mobile Edition (JME). Gerade da die OSGi-Plattform im Bereich der mobilen und eingebetteten Systeme starke Verwendung findet ist eine Beschränkung auf die JSE ausgeschlossen.

Um die OSGi-Spezifikation unabhängig von der Java-Laufzeitumgebung (JRE) durchführen zu können, hat die OSGi-Alliance so genannte Execution Environments definiert. Diese legen eine Menge von Klassen, Interfaces und Methoden fest, die in der Plattform enthalten sein müssen. Ist nun eine Plattform auf ein bestimmtes Execution Environment festgelegt, kann dieses immer noch von verschiedenen JRE erfüllt werden. Die OSGi-Alliance spezifiziert zwei Execution Environments:

- *OSGi/Minimum-1.2:* Die hier definierten Klassen sind grundlegend für die Ausführung des OSGi-Frameworks und der Basis-Services.
- *CDC-1.1/Foundation-1.1:* Abgeleitet vom JME-Foundation-Profil definiert dieses Environment ein Superset der Minimalumgebung und bietet somit eine erweiterte Ausführungsumgebung.

Die Execution Environments werden auf Ebene der Bundles angegeben. So kann der Bundle-Hersteller spezifizieren, welche minimale Umgebung für die Ausführung des Bundles benötigt wird. Bei der Installation des Bundles im Framework kann zudem überprüft werden, ob das Bundle zu der durch das Framework angebotenen Umgebung passt.

Die Module-Schicht Innerhalb der Module-Schicht wird das Modulkonzept der OSGi-Service-Plattform definiert. Im Sinne der komponentenbasierten Softwareentwicklung werden *Bundles* als abgeschlossene Einheiten, die im Framework installiert werden können, eingeführt. Technisch sind Bundles JAR-Archive, in denen die Funktionalität des Bundles über eine Sammlung von Klassen realisiert wird. Zusätzlich enthalten diese Archive alle benötigten Ressourcen und eine Beschreibung der Metadaten.

Die Metadaten werden in der so genannten Manifest-Datei angegeben. Hier wird eine Menge von Attributen angegeben, die das Bundle beschreiben. Die Angabe eines Namens und einer Version zur eindeutigen Identifizierung eines Bundles gehört ebenso dazu wie Angaben über den Hersteller. Die Abhängigkeiten zwischen den Bundles werden vom Framework explizit verwaltet und werden ebenfalls in der Manifest-Datei spezifiziert. Grundsätzlich sind alle Klassen eines Bundles für andere Bundles nicht sichtbar und müssen für eine Benutzung freigeschaltet werden. Dies geschieht über Einträge in den Attributen Import-Packages und Export-Packages, so dass indirekt Abhängigkeiten zu anderen Bundles aufgebaut werden. Eine direkte Abhängigkeit kann über das Attribut Require-Bundle aufgebaut werden, dies wird von der OSGi-Allianz aber nicht empfohlen.

Das zugrunde liegende Komponentenmodell schreibt dem Bundle-Entwickler die Implementierung eines speziellen Activator-Interfaces vor. Über dieses Interface wird zur Laufzeit der Komponente ein so genannter Bundle-Context übergeben, der die Schnittstelle zum Framework darstellt. Hierüber können Services registriert und gesucht werden sowie Events abonniert und versendet werden.

Die Lifecycle-Schicht Die in der Module-Schicht definierten Komponenten nehmen während ihrer Lebenszeit unterschiedliche Zustände ein. Diese sind in der Lifecycle-Schicht spezifiziert. Die Übergänge zwischen den einzelnen Zuständen sind in Abbildung 2.10 veranschaulicht.

Wird ein Bundle neu im Framework installiert, so nimmt es zunächst den Zustand *INSTALLED* ein und bekommt vom Framework eine systemweit eindeutige BundleID zugewiesen, die im Framework aufsteigend an alle Bundles verteilt wird. Anschließend versucht das Framework alle spezifizierten Abhängigkeiten des Bundles aufzulösen. Im Erfolgsfall wechselt das Bundle in den Zustand *RESOLVED*.

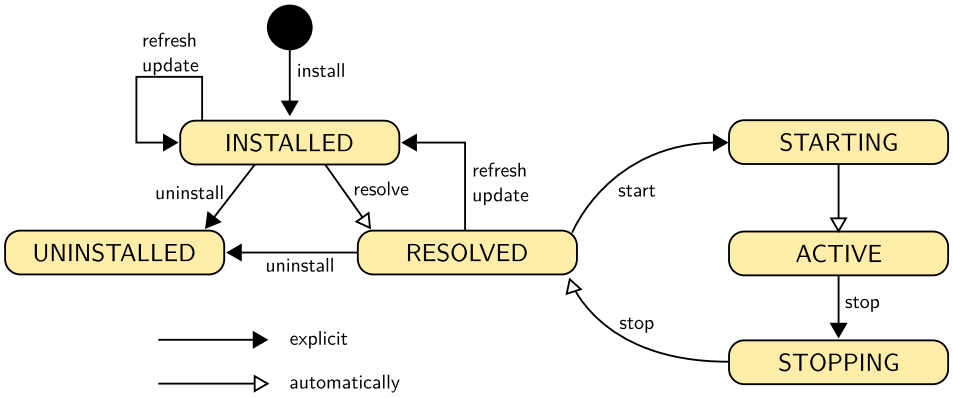


Abbildung 2.10: Der Lebenszyklus eines OSGi-Bundles [The12b]

Von hier aus kann ein Bundle gestartet werden. Daraufhin wechselt das Bundle in den Zustand *STARTING* und die `start()`-Methode der Activator-Klasse wird automatisch aufgerufen. Wird diese Methode erfolgreich abgearbeitet ist der Bundle-Zustand *ACTIVE* erreicht. Ein stoppen des Bundles führt dann zum Aufruf der `stop()`-Methode und zum Erreichen des *STOPPING*-Zustands und anschließend wieder in den Zustand *RESOLVED*.

Wird ein Bundle deinstalliert, wechselt es zunächst in den Zustand *UNINSTALLED*. In diesem Zustand bleiben alle exportierten Ressourcen für andere, in Abhängigkeit stehende Bundles weiterhin verfügbar. Erst wenn alle Abhängigkeiten aufgelöst wurden wird das Bundle komplett aus dem Framework entfernt.

Der Service-Schicht Die Module-Schicht und die Lifecycle-Schicht spezifizieren die Struktur der Komponenten und deren Lebenszyklus innerhalb des OSGi-Framework. Die Service-Schicht definiert nun die Kommunikation der Bundles untereinander.

Die Kommunikation erfolgt in Form von Services. OSGi-Services sind Java-Objekte, die innerhalb eines Bundles erzeugt werden. Sie können zur Laufzeit an der *OSGi Service Registry* unter einem Interface-Namen registriert werden und dort von anderen Bundles abgefragt und genutzt werden. Dies ermöglicht den Austausch von Objekten zwischen Services.

Services werden in OSGi typischerweise unter dem Interface-Namen registriert, das die öffentlichen Methoden des Service definiert. Zusätzlich kann bei der Registrierung ein Properties-Objekt übergeben werden, dessen Inhalt den Service beschreibt. Andere Bundles können den Service nun über den registrierten Namen oder über die Eigenschaften betreffende Filterausdrücke auffinden.

Um dynamisch auf das An- und Abmelden von Services reagieren zu können, wurde in OSGi das Konzept der *Service Listener* und darauf basierend der *Service Tracker* eingeführt. Sie kapseln den Umgang mit der Service Registry und erlauben das Überwachen von Services zur Laufzeit.

Das komplette Konzept der Service-Kommunikation in OSGi ist in Abbildung 2.11 veranschaulicht. Der Service-Anbieter exportiert das Service-Interface und registriert den Service in der Service Registry. Diese wird vom Client überwacht und der Service

abgerufen und verwendet.

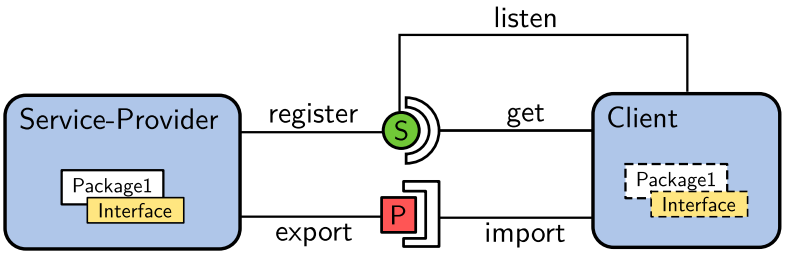


Abbildung 2.11: Service-Kommunikation in OSGi

Die Security-Schicht Die Security-Schicht definiert Konzepte zur Einschränkung von Ausführungsrechten einzelner Bundles. Das Security-Konzept basiert auf dem Sicherheitsmodell, das im JDK 1.2 eingeführt wurde und erweitert es um OSGi-spezifische Anforderungen wie den feingranularen Bezug auf einzelne Bundles. Die Unterstützung des Security-Konzepts ist optional.

Die OSGi-Standard-Services Neben der Spezifikation des OSGi-Frameworks definiert die OSGi-Allianz in der *OSGi Core Specification* [The12b] und in der *OSGi Compendium Specification* [The12a] eine Reihe von Standard-Services. Diese bieten Lösungen für unterschiedliche OSGi-spezifische Problemstellungen und können bei der Entwicklung eigener Anwendungen genutzt werden.

- *Event Admin Service:* Neben der Kommunikation über Services bietet die OSGi-Service-Plattform auch die Möglichkeit einer Event-basierten Kommunikation. Der Event Admin Service stellt dabei einen Mechanismus bereit, mit dem Bundle übergreifend Events propagiert werden können. Events bestehen aus einem Topic, welches das Event kategorisiert, und der Nutzlast. Bundles können nun Handler für spezifische Topics registrieren und erhalten so die über den Event Admin Service versendeten Events.
- *Configuration Admin Service:* Der Service ermöglicht über eine definierte Schnittstelle die zentrale Konfiguration beliebiger Bundles und Services zur Laufzeit. Über diese Schnittstelle können Konfigurationen angelegt, angezeigt, bearbeitet und wieder gelöscht werden. Jede Konfiguration ist einem bestimmten Bundle zugeordnet, eine Änderung wird dem Bundle automatisch übermittelt.
- *Package Admin Service:* Der Package Admin Service ermöglicht das Abfragen von Package-Abhängigkeiten zwischen installierten Bundles. Dieser Dienst ist Bestandteil des *Framework System Bundle* und ist somit immer vorhanden. Er wird immer benötigt, wenn die Abhängigkeiten eines neu installierten Bundles aufgelöst werden müssen.
- *Metatype Service:* Über den Metatype-Service können die Konfigurationen eines Bundles spezifiziert werden. So wird angegeben welche konfigurierbaren Variablen existieren, welchem Typ sie entsprechen und mit welchem Standardwert sie belegt

sind. Diese Informationen können von Management Agents abgefragt werden um Konfigurationen bspw. grafisch darzustellen.

- *Log Service:* Der Log-Service bietet die Möglichkeiten Log-Informationen, wie Warnungen, Debug-Meldungen oder Fehlermeldungen zu schreiben und über Log-Reader auszugeben.

Die Einsatzgebiete von OSGi Ursprünglich wurde OSGi für den Gebrauch in so genannten *Residential Internet Gateways* entwickelt [Hof01; Mar01]. Die Plattform dient dabei als Verbindungszentrale zwischen hausinternen, heterogenen Netzwerken und dem Internet im Bereich der Gebäudesystemtechnik. Für die Unterstützung neuer Geräte innerhalb des Gebäudes müssen auf dem Gateway neue Dienste zur Unterstützung dieses Geräte installiert werden. Mit Hilfe der OSGi-Service-Plattform kann dieser Vorgang ohne Neustart des kompletten Systems erreicht werden.

Eine neuere aber schon etablierte Anwendung betrifft den Automotive-Bereich. In den BMW 5er- und 7er-Reihen dient die OSGi-Service-Plattform als Grundlage für das Telematik- und Infotainment-System [Saa03]. Auch Volvo hat sich an dieser Entwicklung beteiligt und setzt ein entsprechendes System ein.

Seit Eclipse 3.0 bildet OSGi die Grundlage für Eclipse und hat damit auch den Bereich der Desktop-Anwendungen erschlossen [Gru05]. Zwar wird Eclipse typischerweise als Entwicklungsplattform genutzt, es kann mittlerweile aber auch als Rich-Client-Platform verwendet werden, so dass OSGi als Grundlage aller auf Eclipse basierenden Desktop-Applikationen dient [McA05].

Neben den Desktop-Anwendungen wird OSGi auch für serverseitige Anwendungen eingesetzt. So basieren sowohl der Websphere-Application-Server seit Version 6.1 auf der OSGi-Service Plattform, als auch der JBOSS-Application Server.

Mit dem JSR 232 (Mobile Operational Management) [Ami08] treibt die OSGi-Allianz selbst den Einsatz von OSGi auf Smartphones voran. Die Spezifikation definiert ein Komponenten-Management-Framework, das Smartphones erlaubt ihre Funktionalität zu erweitern, indem neue Komponenten zur Laufzeit nachinstalliert werden.

Der HealthCare-Bereich ist ein weiterer typischer Anwendungsbereich für die OSGi-Service-Plattform. Verschiedene OSGi-basierte Lösungen bieten Assistenzsysteme zur Unterstützung und Überwachung von bedürftigen Personen an [Hei09; Mar09]. Dabei werden zunehmend generische Plattformen entwickelt, die verschiedene HealthCare-basierte Basisdienste anbieten, um die spezialisierte Entwicklung von HealthCare-Applicationen zu vereinfachen [Zee10].

2.1.4 Geräteintegration

Die Anbindung von Geräten, Sensoren und Aktoren ist für die meisten AAL-Systeme von zentraler Bedeutung. Neben medizinischen Sensoren werden auch Geräte aus vielen anderen Domänen, wie dem Mobilfunk, dem Automotive-Sektor oder der Gebäudeautomatisierung eingebunden. In der Gebäudeautomatisierung definieren Feldbusse wie das *Local Operating Network (LON)*, *KNX* oder das *Building Automation and Control Network (BACnet)* herstellerunabhängige Möglichkeiten, um Sensoren und Aktoren anzubinden.

Diese Ansätze sind allerdings auf die jeweiligen Feldbusse als Netzwerktechnologie sowie auf die im jeweiligen Standard definierten Geräteklassen beschränkt.

Eine Hersteller übergreifende, oder gar Domänen übergreifende Lösung, wie sie das Internet of Things verfolgt, ist mit diesen Ansätzen nicht erreicht. Um dieses Problem zu lösen und die Kommunikation zwischen den Geräten und Applikationen der einzelnen Technologien zu ermöglichen, ist die Einführung von Gateways notwendig. In [Bov14] werden dazu unterschiedliche Möglichkeiten analysiert. Es wird zwischen $N-to-N$ und $N-to-1^*$ Protokollumsetzern unterschieden, wie Abbildung 2.12 verdeutlicht. Da für die erste Variante im schlechtesten Falle $\frac{n*(n-1)}{2}$ Mappings notwendig sind, wird die zweite Variante, mit nur maximal n Mappings empfohlen. In diesem Fall müssen die Geräte aller Technologien in ein Gateway integriert werden um dort über ein standardisiertes Protokoll zugreifbar zu sein.

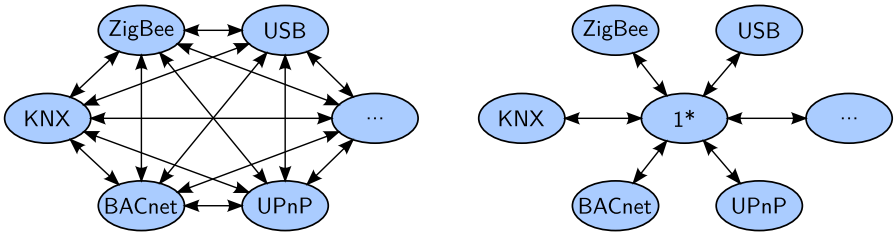


Abbildung 2.12: Gegenüberstellung von $N-to-N$ und $N-to-1^*$ Gateway-Lösungen

Wird dieses Gateway als komponentenbasiertes Framework realisiert, können dort auf einfache Art und Weise Applikationen erstellt werden, die Domänen übergreifend auf Sensoren, Aktoren und Geräte zugreifen. Es können Lösungen angeboten werden, in denen Geräte über Services zugreifbar sind, und es für Anwendungsentwickler keinen Unterschied in der Benutzung von Geräte-Services und in der Benutzung von Services zur inter-Komponenten-Kommunikation gibt. Da für diese Arbeit einzig die OSGi-Service-Plattform als komponentenbasiertes Framework relevant ist, werden nur Lösungen für die Geräteintegration innerhalb OSGi vorgestellt.

OSGi Device Access

Durch die ursprüngliche Ausrichtung von OSGi als Residential-Internet-Gateway war die Geräteintegration von Beginn an ein integraler Baustein der Plattform. Innerhalb der *Device Access Specification* ist der Umgang mit Geräten spezifiziert. Die Spezifikation beschreibt das Auffinden von Geräten, die anschließende Verknüpfung mit dem OSGi-Framework, sowie das Herunterladen und Installieren von Treibern zur Laufzeit für die Unterstützung einer Hot-Plug-Fähigkeit von Geräten.

Im wesentlichen besteht die Device Access Spezifikation aus drei Teilen, dem *Base Driver*, dem *Device Manager* und den *Driver Services*. Der Base Driver ist die zentrale Treiberkomponente zur Kommunikation mit den physikalischen Geräten und repräsentiert jeweils eine bestimmte Technologie (USB, UPnP, JINI etc.). Einige dieser Technologien unterstützen durch integrierte Discovery-Mechanismen das automatische Auffinden von Geräten. Wird durch diesen Mechanismus dem Base Driver ein neues Gerät gemeldet, kann es direkt weiterverarbeitet werden. Unterstützt eine Technologie keinen Discovery-

Mechanismus, wie zum Beispiel RS232, ist externe Hilfe für die Integration eines neuen Geräts erforderlich. Dieses Verhalten ist aber nicht weiter spezifiziert. Wurde ein neues Gerät vom Base Driver erkannt, so registriert dieser einen *Device Service*. Dieser Service repräsentiert das physikalische Gerät und wird durch Metadaten kategorisiert.

Der *Device Manager* reagiert auf die Registrierung von Device Services und koordiniert anschließend die Verknüpfung des Device Service mit einem passenden *Driver Service*. Diese Treiber müssen nicht im Framework enthalten sein, sondern können auch dynamisch heruntergeladen und im Framework installiert werden. Für die Auswahl eines passenden Treibers setzt der Device Manager einen fest spezifizierten Algorithmus um.

Der *Driver Service* implementiert nun die eigentliche Funktionalität des Gerätes. Die Art und Weise ist in der Spezifikation aber nicht festgelegt. Die OSGi-Allianz spezifiziert eine Reihe von verschiedenen Treiberkategorien, die alle unterschiedliche Aufgaben abdecken:

- *Refining Driver:* Ein Refining Driver verfeinert ein vorhandenes Gerät. So wird beispielsweise ein generisches USB-Gerät von diesem Treiber als Maus erkannt und ein Maus-Device-Service wird registriert. Die Mehrheit aller Treiber fällt in diese Kategorie.
- *Network Driver:* Ein Netzwerktreiber kann ein gesamtes Netzwerk als einzelnes Device repräsentieren. So können über Broadcast-Mechanismen alle Endpunkte des Netzwerks über den Device Service erreicht werden.
- *Composit Driver:* Manche physikalische Geräte können aus mehreren logischen Geräten bestehen. So kann ein USB-Gerät durch den Composit Driver z. B. in ein Audio- und ein Eingabe-Gerät unterteilt werden.
- *Multiplexing Driver:* Dieser Treiber fasst verschiedene physikalische Geräte zu einem logischen Gerät zusammen.

Die Device Access Spezifikation ist seit den Anfängen von OSGi ein wichtiger Bestandteil der Spezifikation. Folglich existieren mittlerweile eine Reihe von Base-Driver-Implementierungen für verschiedene Technologien. Der UPnP-Basedriver kann als Referenzimplementierung der Spezifikation angesehen werden und ist mittlerweile Teil der OSGi-Compendium-Spezifikation [The12a]. Ähnliche Technologien wurden mit der Veröffentlichung des DPWS-Base-Driver [Bot08] und des JINI-Base-Driver unterstützt. Innerhalb des Forschungsprojekts GiraffPlus, Teil der AALOA-Initiative, wurde die Unterstützung der Zigbee-Technologie realisiert [Gir13], die gerade im Umfeld ambienter Systeme eine hohe Relevanz hat. Weitere Base-Driver-Implementierungen existieren für die Technologien Bluetooth, USB und Tmote [Che12].

IoTSys

IoTSys ist eine Arbeit der Universität Wien und definiert einen kompletten Protokollstack für die Integration von Gebäudeautomatisierungs-Systemen in die Internet-of-Things-Welt [Jun12a; Jun12b]. Es wird eine $N-to-1^*$ Gateway-Lösung umgesetzt und innerhalb eines OSGi-Frameworks eine Kommunikation mit Geräten verschiedener Technologien über standardisierte Schnittstellen ermöglicht.

Auf der Anwendungsschicht basiert IoTSys auf dem von OASIS spezifizierten oBIX-Standard [Ehr06]. oBIX ist ein XML- und Webservice basierter Mechanismus zur Darstellung und Übertragung von Daten in der Gebäudeautomatisierung. Dazu werden RESTful-basierte Webservice-Schnittstellen für den Zugriff definiert. Die Daten werden in XML übertragen, deren Format über Schemata spezifiziert ist. Für ein besseres Verständnis zwischen den Parteien, werden verschiedene Datenmodelle wie sensorML oder domoML vorgeschlagen.

Zwischen oBIX auf der Anwendungsschicht und dem für das IoT typische IPv6 auf der Netzwerkschicht, werden auf der Transportschicht zwei Alternativen in der IoTSys-Architektur umgesetzt, die parallel betrieben werden können. Die oBIX-Protokollbindung für HTTP und SOAP basiert auf TCP und zielt auf die Verwendung in Enterprise-Umgebungen. Für stark ressourcenbeschränkte Umgebungen existiert eine weitere Protokollbindung für CoAP [She11].

Für die einzelnen Technologien werden nun Adapter in Form von OSGi-Bundles angeboten, die in die IoTSys-Architektur integriert werden. Für die Anwendungen werden schließlich uniforme Interfaces für CoAP und HTTP angeboten, über die mit den Geräten der Gebäudeautomatisierung auf die gleiche Weise wie mit Geräten des IoT kommuniziert werden kann.

SODA

Das Eclipse-Projekt SODA [Deu06] basiert ebenfalls auf OSGi und adressiert das Problem Geräte und deren Funktionen als Services im Sinne einer Service-orientierten Architektur zu kapseln. Die Architektur ist in mehrere Schichten aufgeteilt, wie Abbildung veranschaulicht. Auf der untersten Ebene befindet sich die *Device Layer*, in der die physikalischen Geräte zu finden sind. Übergeordnet ist die *Device Service Layer*, in denen die physikalischen Geräte der unterliegenden Schicht als Services gekapselt werden. Auf der obersten Schicht befindet sich die *Composition Layer*. Hier können die Device-Services nach SOA-Prinzipien [Erl05] kombiniert werden um höherwertige und komplexere Logiken anzubieten.

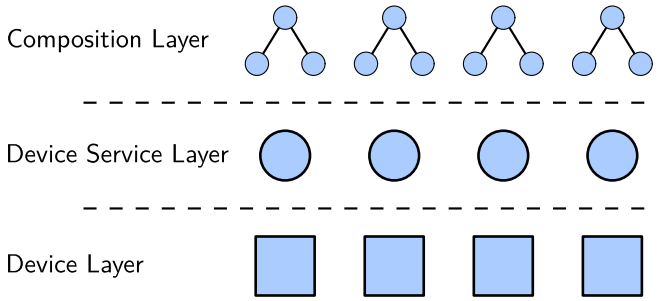


Abbildung 2.13: Architektur des SODA-Projekts zur serviceorientierten Geräteintegration

Weiterhin werden zwei Zwischenschichten definiert. Die Erste befindet sich zwischen *Device Layer* und *Device Service Layer*. Hier werden die Verbindungen zwischen den Geräten und den Services in Form von Hardware- und Software-Interfaces realisiert, die als so genannte Profile festgelegt sind. In der Schicht zwischen der *Device Service Layer* und der *Composition Layer* befindet sich eine weitere Schicht, in denen das Finden und

Binden von Services behandelt wird. Hier werden Funktionen des OSGi-Framework, wie die Service Registry, verwendet.

Durch diese feingranulare Unterteilung in die einzelnen Schichten kann der Implementierungsaufwand zwischen Geräteherstellern, den Dienstanbietern und den Anwendungsentwicklern gut aufgeteilt werden. Es stehen bereits eine Reihe von Implementierungen für unterschiedliche Kommunikationsprotokolle, wie zum Beispiel die serielle Schnittstelle, bereit. Durch das modulare Prinzip der SODA-Architektur können weitere Implementierungen hinzugefügt werden.

2.1.5 Verteilte Anwendungen

Für viele industrielle Anwendungen bildet die Zusammenarbeit von Programmen auf unterschiedlichen Rechnern das entscheidende Rückgrat. Ein solches System wird von Coulouris [Cou05] als *Verteiltes System* definiert:

> „Ein verteiltes System ist ein System, in dem sich Hardware- und Softwarekomponenten auf vernetzten Computern befinden und miteinander über den Austausch von Nachrichten kommunizieren."

Ein verteiltes System setzt sich folglich aus mehreren unabhängigen Bausteinen zusammen. Die Funktionalität des Gesamtsystems kann dabei nicht durch eine Einzelkomponente erbracht werden, sondern nur durch Kooperation der einzelnen Komponenten.

Verteilte Systeme haben seit ihren Anfängen eine enorme Entwicklung aufzuweisen und diese noch lange nicht abgeschlossen. Durch die, schon von Moore 1965 vorausgesagte rasche Weiterentwicklung von Rechnerhardware [Ham99; Moo65; Moo98] existiert mittlerweile nahezu keine Begrenzung für die Realisierung von Anwendungen mehr. Die Erhöhung von Übertragungsleistungen der Rechnernetze formte dann einen weiteren Baustein für die steigende Bedeutung von verteilten Systemen. Eine stetige Weiterentwicklung der Infrastruktur, wie die Erfindung des WWW als globale Informationsbereitstellung und Verarbeitung sowie die Ausbreitung von Mobilfunknetzen und mobilen, ortsunabhängigen Rechnern lenkte die Möglichkeiten für auf verteilten Systemen basierenden Anwendungen in neue Bahnen.

Solche Anwendungen werden gemeinhin als *Verteilte Anwendungen* bezeichnet. Eine verteilte Anwendung nutzt ein verteiltes System, um Anwendern eine in sich geschlossene fachliche Funktionalität zur Verfügung zu stellen. Dabei ist die Anwendungslogik in einzelne, weitgehend voneinander unabhängige Anwendungskomponenten unterteilt, die auf verschiedenen Rechnern des verteilten Systems liegen.

Zwischen dem verteilten System und der verteilten Anwendung liegt die *Middleware* [Ber96]. Ihre Aufgabe ist es, die verteilte Anwendung von den Aspekten der Netzwerkprogrammierung möglichst vollständig zu trennen (siehe Abbildung 2.14). Diese Abstraktion erleichtert den Entwurf und die Umsetzung verteilter Anwendungen und verringert so deren Fehleranfälligkeit. Ein weiteres Hauptziel der Middleware ist die so genannte Zugriffstransparenz. Zugriffstransparenz bedeutet, dass die Middleware die Vermittlung von Aufrufen im verteilten System übernimmt. So bleibt der Anwendung verborgen, ob genutzte Ressourcen lokal oder entfernt vorliegen.

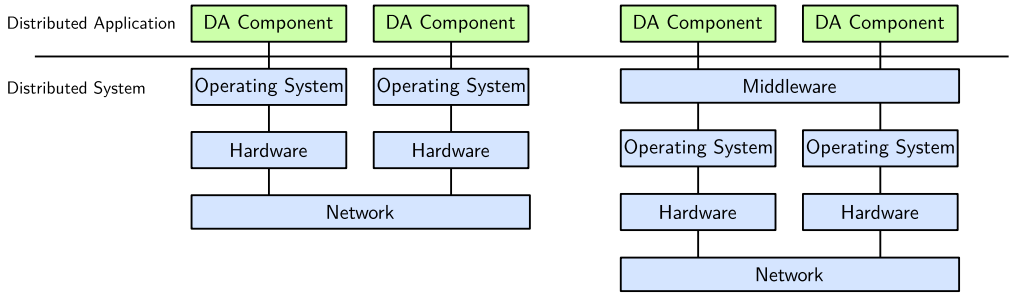


Abbildung 2.14: Funktion einer Middleware

Kommunikation in verteilten Systemen

Die Kommunikation in verteilten Systemen steuert den Austausch von Nachrichten und ermöglicht so Interoperabilität und Kooperation der einzelnen Knoten. Allgemein wird dabei zwischen *synchroner* und *asynchroner Kommunikation* unterschieden, Hammerschall [Ham05] gibt dazu folgende Definition:

> "Bei synchroner Kommunikation ist der Sender einer Nachricht so lange in seiner weiteren Ausführung blockiert, bis er vom Empfänger eine Antwort auf die Nachricht erhält. Bei asynchroner Kommunikation ist der Sender nicht blockiert. Sender und Empfänger werden parallel weiter ausgeführt."

Die synchrone Kommunikation eignet sich somit besonders für verteilte Anwendungen mit enger Kopplung und ermöglicht eine genaue Steuerung des Kommunikationsablaufs, während die asynchrone Kommunikation eine weitgehende Entkopplung der Kommunikationspartner ermöglicht.

Das Konzept des *Remote Procedure Call (RPC)* ermöglicht es, die Kommunikation in verteilten Systemen in Form eines (verteilten) Prozeduraufrufs durchzuführen. Abbildung 2.15 zeigt den Ablauf und die typische Architektur eines RPC-Systems.

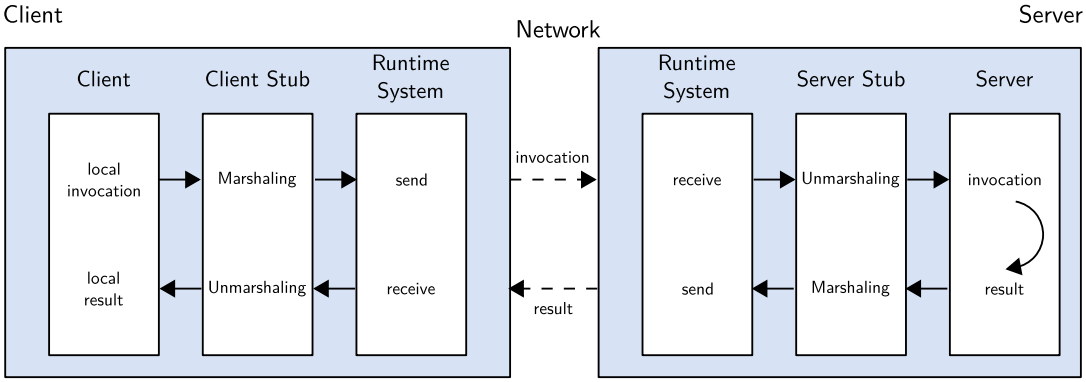


Abbildung 2.15: Remote Procedure Call (RPC) [Sch07]

Wenn ein Client-Rechner über ein Netzwerk die Dienstleistung eines Servers nutzen will, so werden zunächst aus der Schnittstellenbeschreibung der Server-Schnittstelle die so genannten *Stubs* generiert. Client-Stub und Server-Stub (auch *Skeleton* genannt) kapseln

alle notwendigen Funktionen um einen lokalen Aufruf an einen entfernten Rechner zu übermitteln bzw. am entfernten Rechner zu empfangen. Zusätzlich übernehmen sie die Konvertierung der Ein- und Ausgabeparameter in ein geeignetes Übertragungsformat (*Marshalling*) und wieder zurück (*Unmarshalling*). Der Client kann den gewünschten entfernten Aufruf auf transparente Weise als lokalen Aufruf am Client-Stub ausführen. Der Client-Prozess geht dann in einen Wartezustand über, bis die Antwort des Servers eintrifft. Es existieren eine Menge von Implementierungen des RPC-Konzepts, deren prominenteste das *Sun RPC* (auch *ONC RPC*) [Sun88] darstellt.

In verteilten Objektsystemen sind vorrangig die Objekte samt ihrer Eigenschaften und Methoden Gegenstand der Verteilung. Die Kommunikation zwischen entfernten Objekten findet mittels entfernter Methodenaufrufe (*Remote Method Invocation - RMI*) statt, einer Weiterentwicklung des RPC. Ähnlich wie beim RPC werden aus einer Schnittstellenbeschreibung Stub- und Skeleton-Komponenten für Client und Server erzeugt.

Zusätzlich wird als Teil des Client-Stubs ein Stellvertreterobjekt, auch *Proxy*-Objekt genannt, erzeugt (siehe Abbildung 2.16). Dieses Objekt implementiert die gleiche Schnittstelle wie das Server-Objekt. Aufrufe an dieses Objekt werden allerdings nicht lokal verarbeitet sondern vom Client-Stub über den Skeleton an das Server-Objekt weitergeleitet.

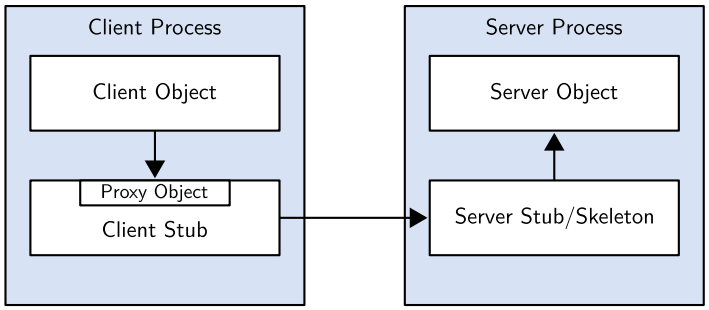


Abbildung 2.16: Remote Method Invocation (RMI) [Ham05]

Um den entfernten Objektzugriff auf einem Server, der verschiedene Anwendungsobjekte anbietet, einheitlich realisieren zu können, bietet es sich an, die Serverschnittstelle generisch zu realisieren. So können die entfernten Aufrufe unabhängig von den einzelnen Objektschnittstellen entgegen genommen werden. Diese Schnittstelle wird durch so genannte *object adapter* [Tan06] bereitgestellt.

Ein am Objektadapter eingehender Aufruf wird zunächst einem konkreten Anwendungsobjekt zugeordnet. Die dazu nötige Objektreferenz muss Teil der eingegangenen Nachricht sein. Anschließend wird anhand einer Aktivierungsstrategie (*activation policy*) des Objektadapters entschieden, welchem ausführenden Thread der Aufruf zugeordnet wird. Dabei kann es pro Server mehrere Objektadapter mit jeweils unterschiedlichen Aktivierungsstrategien geben. In diesem Fall müssen eingehende Nachrichten zunächst durch den *request demultiplexer* dem Objektadapter mit der richtigen Aktivierungsstrategie zugeteilt werden.

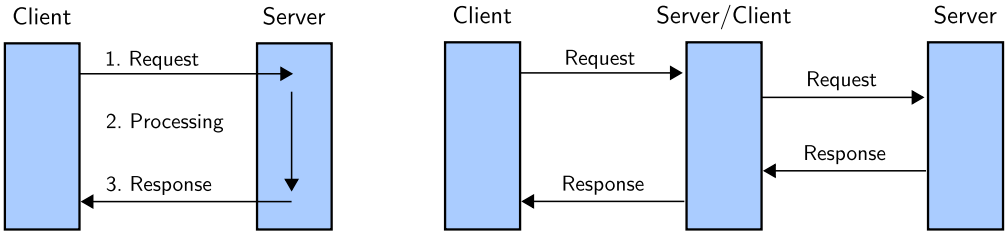
Da der Objektadapter generisch ist, werden diese Aufrufe wiederum über eine einheitliche Schnittstelle an die Skeleton-Objekte der konkreten Anwendungsobjekte übergeben.

Der Skeleton übernimmt dann das Marshalling/Unmarshalling der Nachrichten und die Methodenaufrufe.

Architekturen verteilter Systeme

Für die Realisierung von verteilten Systemen haben sich mehrere Architekturkonzepte entwickelt. Als Grundlegendes Modell hat sich das *Client/Server-Modell* etabliert, das durch das *Objekt-orientierte Modell* weiterentwickelt wurde. Ebenfalls im Fokus aktueller Forschungen und Anwendungen steht das *Ressourcen-orientierte* und das *Dienst-orientierte Modell*.

Client/Server-Modell Das Client/Server-Modell beschreibt die Rollen und den Ablauf der Zusammenarbeit verteilter Softwarekomponenten [Svo85]. Der Dienstnachfrager *(Client)* initiiert die Interaktion und schickt Anfragen an den Diensterbringer *(Server)*. Abbildung 2.17(a) illustriert den weiteren Ablauf. In beide Richtungen gilt eine 1:n-Beziehung. Ein Client kann mehrere Server anfragen und ein Server kann mehrere Kunden bedienen. Die Rollen von Client und Server können während des Ablaufs dynamisch wechseln, z. B. wenn ein Server zur Abarbeitung eines Auftrags einen weiteren Server befragen muss. Dieses Verhalten ist in Abbildung 2.17(b) dargestellt.



(a) Klassische Rollenverteilung **(b)** Rollenwechsel

Abbildung 2.17: Das Client/Server-Modell [Gei95]

Insgesamt beschreibt das Client/Server-Modell nur die logische Verteilung. Die Kommunikationsinfrastruktur wird nicht modelliert. So ist es auch möglich das Modell lokal ohne physische Verteilung umzusetzen.

In den 1990er Jahren stellte die *Open Software Foundation (OSF)* ein Konzept für eine standardisierte Verteilungsplattform vor. Das *Distributed Computing Environment (DCE)* [Joh91; Loc94] strukturiert sich nach dem Client/Server-Modell und verwendet zur Kommunikation der unterschiedlichen verteilten Programme untereinander das RPC-Konzept (DCE RPC). Dem DCE war allerdings kein durchschlagender Erfolg vergönnt, die zugrunde liegenden Konzepte gelten allerdings als wegweisend und haben sich bis heute durchgesetzt.

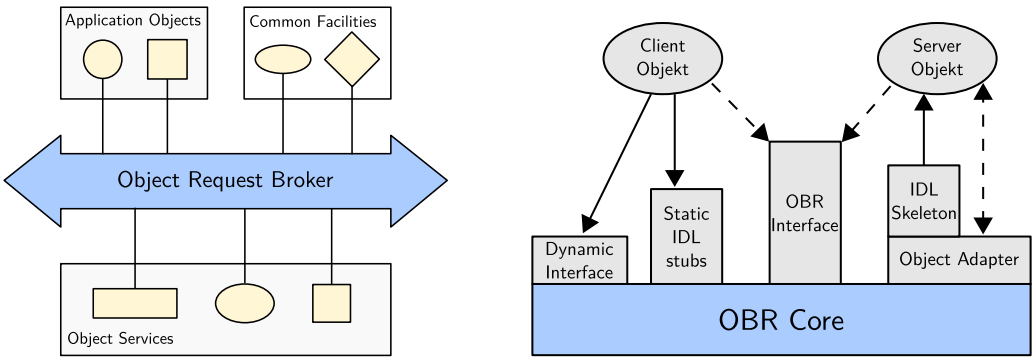
Objektmodell Das Objektmodell kann als eine Art Client/Server-Modell interpretiert werden. Die Einheiten der Kommunikation sind hier Objekte beliebiger Granularität, die weiterhin die vom Client/Server-Modell eingeführten Rollen einnehmen [Sch07]. Ein Objekt ist gekennzeichnet durch seinen Zustand, sein Verhalten und seine Identität [Boo91]. Sie können entsprechend des objektorientierten Programmiermodells verwendet werden,

was eine wesentlich höhere Flexibilität gegenüber dem Client/Server-Modell zur Folge hat. Ein weiterer Aspekt ist, dass großen, der Kommunikation dienenden Objekten, kleine Objekte direkt als Parameter übergeben werden können, die dann wiederum direkt lokal oder entfernt aufgerufen werden. Es wird also je nach Bedarf eine Call-by-value und eine Call-by-reference-Semantik unterstützt.

Das Objektmodell wird von der *Object Management Group (OMG)* vorangetrieben, deren primäres Ziel die Etablierung eines Architekturstandards zur Unterstützung der Integration verteilter Anwendungen ist. Dabei werden drei Anforderungen definiert: Wiederverwendbarkeit, Portabilität und Interoperabilität. Aus der Realisierung dieser Anforderungen mit dem Objektmodell ist schließlich die *Object Management Architecture (OMA)* [Wil95] entstanden.

Die OMA unterscheidet grundsätzlich vier Komponenten, illustriert in Abbildung 2.18(a):

- *Application Objects:* Definieren die Anwendung.
- *Object Request Broker:* Die Kernkomponente ORB vermittelt zwischen den verteilten Objekten. Methodenaufrufe werden zum Zielobjekt geleitet und die Ergebnisse des Aufrufs zurückgeführt.
- *Object Services:* Die Objektdienste unterstützen die verteilten Objektinteraktionen durch Basisfunktionen. Dazu gehören z. B. Objektdienste zu Event-Notifikationen oder Persistenz-Dienste.
- *Comon Facilities:* Sammlung von allgemein nützlichen Objekten.



(a) Object Management Architecture **(b)** CORBA-Übersicht

Abbildung 2.18: OMA-Architektur und deren Implementierung CORBA

Die wohl bekannteste Umsetzung der OMA ist die *Common Object Request Broker Architecture (CORBA)* [Obj95], die vor allem Aufbau, Funktionalität und Schnittstellen eines ORBs konkretisiert. Die Architektur ist in Abbildung 2.18(b) zu sehen. Die spezifizierten Schnittstellen können in Aufrufschnittstellen, damit ein Client-Objekt ein Server-Objekt verwenden kann, und in Schnittstellen zu den Infrastrukturdiensten des ORB unterteilt werden. Die Aufrufschnittstellen, definiert in der *Interface Definition Language (IDL)* [Sno89], können statisch zur Design-Zeit erzeugt werden oder dynamisch zur Laufzeit. Dem aufgerufenen Objekt bleibt diese Unterscheidung verborgen,

es erhält den Aufruf transparent über den Objektadapter und den IDL-Skeleton. Im Sinne der Transparenz bleibt dem Client wiederum verborgen, ob sich ein Objekt lokal oder auf einem entfernten Rechnerknoten befindet, sowie die konkrete Realisierung des Objekts (verwendete Hardware und Programmiersprache), so dass ein weiteres Ziel, die Integration von Anwendungen in verteilten heterogenen Umgebungen erreicht werden kann [Vin97]. Die Kommunikation innerhalb einer CORBA-Implementierung erfolgt über das *Internet Inter-ORB Protocol (IIOP)* [Ruh00], einem auf TCP basierendem Kommunikationsprotokoll.

Ressourcen-orientiertes Modell Der Begriff der *Ressourcen-orientierten Architektur (ROA)* wird von Richardson und Ruby [Ric07] geprägt. In einer ROA werden verteilte Einheiten, wie z. B. Daten, als *Ressourcen* modelliert. Der Zugriff auf diese Ressourcen erfolgt über uniforme Schnittstellen und ermöglicht das Abrufen, Manipulieren und Löschen. Dazu sind die Ressourcen einzeln adressierbar. Die Repräsentation der Ressourcen ist nicht festgelegt, in der Praxis werden häufig die Technologien HTML, JSON oder XML verwendet. Innerhalb der einzelnen Ressourcen können Verknüpfungen zu anderen Ressourcen platziert werden und so die Verbindung zwischen einzelnen Ressourcen realisiert werden. Weiterhin korrelieren innerhalb einer ROA Ressourcenaufrufe nicht, der Aufruf einer Ressource hängt also nie vom vorherigen Aufruf einer Ressource ab. Zusammengefasst besteht eine ROA also aus vier Konzepten – Ressourcen, deren Namen (URI), deren Repräsentation und deren Verknüpfung – und vier Eigenschaften – Adressierbarkeit, Zustandslosigkeit, Verbundenheit und ein uniformes Interface.

Das bekannte *Representional State Transfer (REST)*, eingeführt durch Fielding [Fie00], ist eine konkrete Umsetzung einer ROA. Dort werden die Prinzipien der Kommunikation im WWW erläutert, als REST zusammengefasst und deren Realisierung mittels des *Hypertext Transfer Protocol (HTTP)* beschrieben. Da der Zugriff auf einzelne Ressourcen sehr feingranular ist, und so das Erstellen verteilter Anwendungen einen erhöhten Aufwand verursacht, existieren Konzepte zur Abbildung von Geschäftsprozessen durch die Choreografie einzelner Zugriffe [Pau09].

Dienste-orientiertes Modell Erstmalig 1996 von Gartner [Sch96] erwähnt, werden in der *Service-orientierte Architektur (SOA)* Vorgänge und Funktionen durch Dienste abstrahiert. Ein Dienst ist eine Anwendungskomponente, die unabhängig von der zugrunde liegenden Implementierung, ihre Funktionalität über eine (öffentlich) spezifizierte Schnittstelle anbietet. Verglichen mit Objekten im Objektmodell haben Dienste in der Regel eine wesentlich höhere Granularität. Melzer [Mel10] beschreibt eine SOA als verteilte Anwendungsarchitektur, die durch die folgenden Eigenschaften charakterisiert wird:

- *Interoperabilität:* Die Trennung der Zuständigkeiten nach fachlichen Gesichtspunkten und die Kapselung technischer Details sind elementare Grundgedanken einer SOA. Sowohl zur Servicebeschreibung, als auch für Interaktion, Kommunikation und Nutzung der Services werden einheitliche, breit akzeptierte Standards verwendet, wodurch die Interoperabilität gewährleistet wird.

- *Schnittstellenorientierung:* Dienste werden über einen einheitlichen Mechanismus aufgerufen, der die Anwendungsbausteine plattformunabhängig miteinander verbindet und alle technischen Details der Kommunikation verbirgt. Die adressierte Schnittstelle ist dabei von den Implementierungsdetails abstrahiert. Sie ist selbst beschreibend, d. h. der Service ist über Metadaten technisch und fachlich vollständig spezifiziert.
- *Modularisierung:* Eine SOA beschreibt den Aufbau einer verteilten Anwendung aus einzelnen teilautonomen Bausteinen, die jeweils eine klar umrissene fachliche Aufgabe wahrnehmen. Dabei können nicht nur Funktionalitäten, sondern auch Ressourcen mit großer Abhängigkeit untereinander so zusammengefasst werden, dass sie gegenüber anderen Subsystemen eine möglichst geringe Abhängigkeit aufweisen.
- *Anwendungsorientierung:* Ein Dienst ist eine fest definierte Leistung, die als Element eines oder mehrerer größerer Verarbeitungsabläufe verwendet werden kann.

Dienste können innerhalb eines verteilten Systems angeboten, gesucht und verwendet werden. Dadurch ergibt sich das klassische SOA-Dreieck [Erl05; Hei05], veranschaulicht in Abbildung 2.19. Der *Service-Provider* stellt im Allgemeinen eine Funktionalität oder eine Ressource und eine korrespondierende Beschreibung zur Verfügung. Dort werden die Form der Interaktion (Schnittstellenbeschreibung) und auch nicht-funktionale Aspekte definiert. Die *Service-Registry* ist ein zentral oder dezentral organisierter Verzeichnisdienst für angebotene Dienste. Hier können Dienste durch Service-Provider registriert und durch *Service-Requestor* abgefragt werden. Die Vermittlung an den Client erfolgt dann auf Basis der registrierten Dienstebeschreibungen.

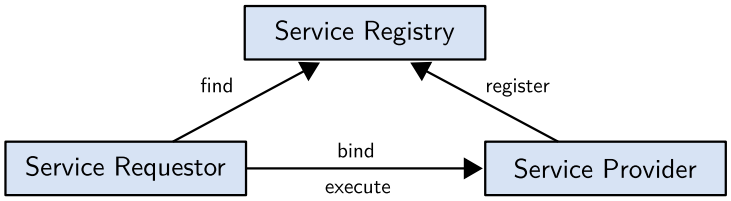


Abbildung 2.19: Komponenten einer SOA

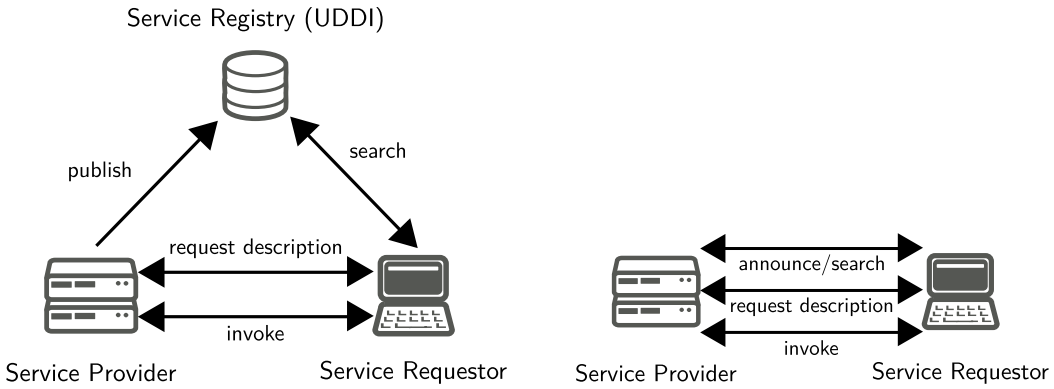
Durch lose Kopplung einzelner Dienste werden komplexe Geschäftsprozesse erstellt. Dieser Vorgang kann durch die Komposition von Diensten vereinfacht und strukturiert werden [Bar06; Pel03]. Dabei wird zwischen der Orchestrierung – dem Zusammensetzen mehrerer Dienste zu einem höherwertigen Dienst in deklarativer Form – und der Choreografie – der Kombination von Diensten zu Geschäftsprozessen durch die Spezifikation von Ablaufprozessen – unterschieden. Dieser Bereich wird in der aktuellen Forschung durchgehend untersucht. So werden z. B. Kompositionen durch formale Modelle beschrieben, die anschließend formal verifiziert werden können [Cam11; Car13] oder Wege einer semantischen Kompositionen erforscht [Med03].

Webservices (WS) sind die wohl bekannteste Umsetzung einer SOA [Alo04] und werden als komplettes Webservices-Protokoll-Framework vom W3C spezifiziert. Das WS-Framework umfasst eine Klasse von Protokoll-Spezifikationen (WS-*), die einer

gemeinsamen Architektur, der *Web Services Architecture (WSA)* [Boo04] unterliegen. Diese können untereinander zu bedarfsgerechten Lösungen kombiniert werden. Durch die Prinzipien dieser Architektur, die verbindlich innerhalb einer Spezifikation definiert sind, ist die Interoperabilität der W3C-Webservices sichergestellt.

Das Basiselement der WSA ist die Nachricht, die sowohl einen Dienstaufruf als auch eine Ressource abbilden kann. Durch die Abstraktion auf Nachrichten erreicht die WSA eine Unabhängigkeit von Betriebssystem und Programmiersprache und das Verbergen von Transportdetails für die oberen Schichten. Alle der WSA folgenden Spezifikationen haben daher ein universelles Nachrichtenformat und Austauschprotokoll gemeinsam. Diese sind SOAP [Gud07] als universelles Protokoll und Container für den Nachrichtenaustausch und XML für die Darstellung der Nachrichten. Die Übertragung der SOAP-Nachricht ist nicht auf ein bestimmtes Transportprotokoll fixiert, muss aber zur Laufzeit an ein solches gebunden werden (Binding). Die SOAP-Nachricht besteht aus einem SOAP-Envelope, der wiederum in einen Header und in den Body aufgeteilt ist. Der Body enthält die anwendungsspezifischen Nutzdaten, während der Header Kontrollstrukturen und Informationen der jeweiligen WS-Spezifikation enthält. Eine Auswahl an WS-Spezifikationen wird im Folgenden beispielhaft dargestellt:

- *WS-Discovery:* WS-Discovery befasst sich mit dem Auffinden von Diensten in einem Netzwerk. In einer klassischen SOA (vergleiche Abbildung 2.19) wird die Suche nach Services über einen zentralistischen Ansatz realisiert. Im Falle der WS-Welt geschieht dies über den Verzeichnisdienst *Universal Description, Discovery and Integration (UDDI)* [Cle04]. Im Gegensatz dazu definiert WS-Discovery einen dezentralen Ansatz. Serviceanfragen werden mittels Gruppennachrichten (Multicast) an Dienstgruppen versendet. Jeder passende Dienst sendet eine Bestätigung nun direkt an den Anfragenden. Ähnlich dem Registrierungskonzept im zentralistischen Ansatz melden sich neue Dienste mit entsprechenden Nachrichten in den Dienstgruppen an und bei einer Nichtverfügbarkeit wieder ab. Eine Gegenüberstellung der beiden Ansätze ist in Abbildung 2.20 zu sehen.



(a) Zentrales Discovery **(b)** Dezentrales Discovery

Abbildung 2.20: Gegenüberstellung von zentralem und dezentralem Discovery

- *WS-Addressing:* WS-Addressing spezifiziert den Austausch von Adressinformationen zur Realisierung einer dauerhaften und eindeutigen Ende-zu-Ende Adressierung von Endpunkten. Diese Informationen umfassen eine Datenstruktur für die Übergabe einer Referenz zu einem Webservice-Endpoint und einer Verknüpfung von Adressinformationen zu einer bestimmten Nachricht. Jede SOAP-Nachricht enthält in ihrem WS-Addressing-Header Informationen über den Absender, den Empfänger der Antwort und den Empfänger von Fehlernachrichten. Dadurch ist es möglich selbst dann auf SOAP-Nachrichten zu antworten, wenn die Nachricht des verwendeten Transportprotokolls nicht mehr existiert. Dieser Mechanismus unterstützt die asynchrone Kommunikation und sichert eine vollständige Unabhängigkeit von den darunterliegenden Schichten und Protokollen.
- *WS-Eventing:* Das wiederholte Aufrufen eines Dienstes, das so genannte Polling, verursacht eine hohe Systemlast und kann durch den WS-Eventing Mechanismus vermieden werden. Events werden gemeinhin als Nachrichten (Event-Notifications) über signifikante Zustandsänderungen definiert, für deren Empfang sich interessierte Clients anmelden müssen (Subscription). WS-Eventing definiert nun ein Protokoll für diesen Anwendungsfall. Über eine *Event-Subscription*-Nachricht registriert sich eine *Event-Senke* bei einer *Event-Quelle* für den Empfang von *Event-Notifications*. Um weitere unnötige Systemlast einzusparen werden den Subscriptions anwendungsspezifische Filterparameter übergeben und laufen zudem nach einer festgelegten Zeit ab.
- *WS-Transfer:* WS-Transfer definiert ein zustandsloses Protokoll um die mittels WS-Addressing adressierten Ressourcen in Form von XML-Repräsentationen einerseits zu versenden und zu empfangen und andererseits zu erstellen, zu löschen und zu manipulieren. Es setzt dabei auf die vorhandene WS-Infrastruktur auf (SOAP).
- *WS-MetadataExchange:* Metadaten bezeichnen die Dienstbeschreibungen, die zur Laufzeit zwischen zwei Endpunkten ausgetauscht werden müssen. WS-MetadataExchange spezifiziert nun, wie diese Metadaten auf WS-Transfer Ressourcen abgebildet werden, mittels WS-Addressing adressiert und über definierte Operationen abgerufen werden können.
- *WS-Security:* Der WS-Security-Standard wird derzeit bei OASIS weiterentwickelt und beschriebt im wesentlichen ein Kommunikationsprotokoll, das es ermöglicht Sicherheitsaspekte bei Webservices zu berücksichtigen. Die Spezifikation behandelt dabei die Möglichkeit Security-Tokens als Teil der SOAP-Nachricht zu übertragen, sowie das Signieren und die Verschlüsselung von Nachrichten.
- *Webservice Description Language (WSDL):* Um einen Dienst nutzen zu können wird eine genaue Beschreibung seiner Schnittstellen benötigt. Innerhalb einer WSDL-Datei werden in XML-Form die angebotenen Dienste, deren Methoden, die Ein- und Ausgabedaten in Form von Nachrichten, die Datentypen und sowie die unterstützen Transport-Bindings definiert. Viele WS-Frameworks benutzen WSDL-Dateien als Grundlage um automatisch Quellcode für die Implementierung von Webservice-basierten Anwendungen zu generieren.

Um die Interoperabilität von WS-basierten Anwendungen innerhalb der umfangreichen Welt der WS-*-Standards zu sichern, werden von verschiedenen Standardisierungsgremien Profile spezifiziert [Bal06; Dur13]. Diese Profile umfassen eine Auswahl der genutzten Standards, sowie Einschränkungen und Erweiterungen bei der Verwendung der einzelnen Spezifikationen.

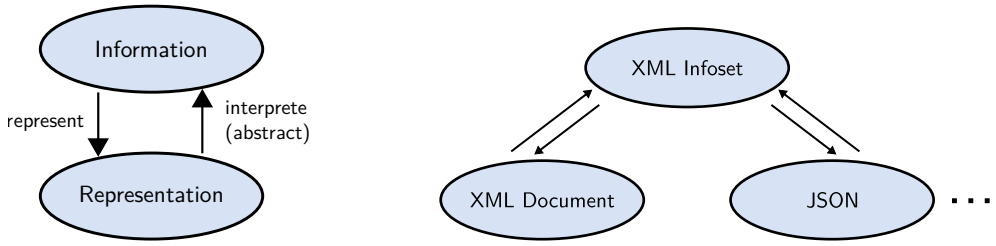
2.1.6 Datenrepräsentation

Das Kernproblem der Datenübertragung in verteilten Systemen ist die Heterogenität der Hardware, der Betriebssysteme und der Programmiersprache. So unterscheiden sich Hardwareplattformen bzgl. der Darstellung von Datentypen in der Reihenfolge der Bytes im Speicher, Betriebssysteme unterstützen unterschiedlichste Formate (EBCDIC, ASCII, Unicode etc.). Auch die verschiedenen Programmiersprachen kennen jeweils ihre individuelle Darstellung von Daten im Hauptspeicher.

Zur Lösung des Problems werden beim Marshaling und Unmarshaling der Daten übergeordnete Formate verwendet, die allen Kommunikationspartnern beziehungsweise der Middleware bekannt sind. Im Laufe der Zeit wurden einige externe (plattformunabhängige) Formate entwickelt, wie z. B. das *External Data Representation Format (XDR)* [Sri95], weitgehend durchgesetzt hat sich aber heute die Auszeichnungssprache Extensible Markup Language (XML). So ist XML elementarer Bestandteil der Webservices-Welt (SOAP, WSDL).

XML wird vornehmlich vom W3C gepflegt und umfasst eine ganze Familie von Spezifikationen. Bei XML wird zwischen der *Serialisierung*, der *Kodierung*, der *Grammatik* und dem *Vokabular* unterschieden.

Allgemein beschreibt die Serialisierung die Abbildung von abstrakten Informationen auf eine sequentielle, übertragbare Form (siehe Abbildung 2.21(a)). Im Kontext von XML werden durch die XML-Information-Set-Spezifikation (*XML-Infoset*) [Cow04] das abstrakte Datenmodell, also die für die Serialisierung zur Verfügung stehenden Strukturelemente beziehungsweise Informationsträger (das Dokument, die Elemente, die Attribute, die Kommentare, die Namensräume usw.) beschrieben. Diese werden dann typischerweise in XML-Dokumente, oder auch in alternative Repräsentationen wie z. B. JSON [Cro06] serialisiert (siehe Abbildung 2.21(b)).



(a) Allgemeine Informationsrepräsentation **(b)** Repräsentation von XML-Infosets

Abbildung 2.21: Allgemeine Datenrepräsentation und die Anwendung in XML

Die Struktur eines XML-Dokuments wird durch sein Vokabular und die dazugehörige Grammatik beschrieben. Das Vokabular definiert alle zur Verfügung stehenden konkreten Strukturelemente, beziehungsweise Informationsträger, die Reihenfolge und Struktur der

einzelnen Vokabeln wird durch die Grammatik festgelegt. Die Definition beider Teile kann innerhalb eines XML-Schema-Dokuments [Tho04] realisiert werden. Der Zusammenhang zwischen Vokabular und Grammatik innerhalb des Prozesses zur Serialisierung von binären Objekten ist in Abbildung 2.22 veranschaulicht.

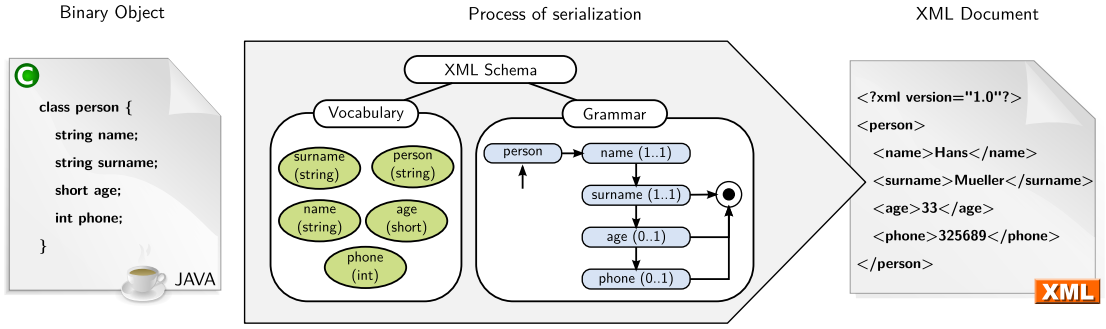


Abbildung 2.22: Erstellung eines XML-Dokuments aus einem Binärobjekt

Um serialisierte Objekte, also im typischen Fall XML-Dokumente, übertragen zu können, müssen diese als Datenstrom kodiert werden. Das Format der Kodierung ist dabei nicht festgelegt. Unterschiedliche Datenströme können somit identische Daten repräsentieren, wodurch das Darstellungsformat der Nachrichten vom Datenmodell der Serialisierung entkoppelt wird. XML legt in der XML-1.0-Spezifikation [Bra08] allerdings ein grundsätzliches, auf Unicode basierendes Datenformat fest.

2.1.7 Effiziente Kommunikation für eingebettete leichtgewichtige Systeme

Bei der Entwicklung für leichtgewichtige Systeme müssen die Eigenschaften dieser Systeme bei der Softwarearchitektur starke Berücksichtigung finden. Eingebettete Systeme haben zumeist wenig Ressourcen, wie Speicher oder CPU, zur Verfügung und müssen dennoch im verteilten Umfeld im Rahmen der Kommunikation das Gleiche leisten wie Enterprise-Systeme. Dieses Kapitel richtet den Blick auf eine effiziente Kommunikation im Umfeld leichtgewichtiger Systeme.

Kommunikationsprotokolle

Im Umfeld leichtgewichtiger Systeme existieren eine Reihe unterschiedlicher Kommunikationsprotokolle. Proprietäre Protokolle fördern Insellösungen, die im Rahmen dieser Arbeit explizit vermieden werden sollen. Daher werden nur Protokolle betrachtet, die offen spezifiziert und im betrachteten Umfeld zu finden sind.

CoAP Das *Constrained Application Protocol (CoAP)* [She11] ist ein Protokoll auf der Anwendungsschicht, das von der IETF entwickelt wird. Das Protokoll wird für Ressourcen beschränkte Umgebungen entwickelt, da das im Web-Umfeld häufig verwendete HTTP die Anforderungen dieser Umgebungen nicht vollständig erfüllen konnte.

CoAP ist konzeptionell stark an HTTP angelehnt, unterscheidet sich aber – dem Ziel der reduzierten Komplexität folgend – in Teilen signifikant. Verglichen mit HTTP basiert CoAP auf dem leichtgewichtigen UDP anstelle von TCP und reduziert allein dadurch die Komplexität enorm. Da TCP im Gegensatz zu UDP bereits auf der Transportschicht die

Zuverlässigkeit des Nachrichtenaustauschs garantiert, muss diese Eigenschaft in CoAP in der Anwendungsschicht reguliert werden. Innerhalb eines UDP-Pakets definiert CoAP einen 4 Byte langen Header mit zusätzlichen optionalen Teilen, so dass die Header-Größe eines typischen Aufrufs zwischen 10 und 20 Byte liegt [Bor12]. Der Header besteht aus einer Versionsnummer des Standards, dem Nachrichtentyp, der Token-Länge, einem Code zur Identifizierung einer Anfrage oder Antwort und einer Nachrichten ID. Die gesamte CoAP-Nachricht ist in Abbildung 2.23 illustriert. Es existieren vier Nachrichtentypen: *confirmable (CON)*, *non-confirmable (NON)*, *ackknowledgement (ACK)* und *reset (RST)*. Mit Hilfe dieser Nachrichtentypen wird die angesprochene Transportgarantie realisiert. Eine zu bestätigende Nachricht wird von der Quelle so lange zum Ziel gesendet, bis das Ziel den Empfang bestätigt hat. Um die Netzbelastung zu verringern werden Wiederholungen mit einem exponentiellen Back-Off gesendet. Der Empfang kann entweder über ein ACK-Paket oder ein RST-Paket quittiert werden. Ein RST-Paket bestätigt zwar den Empfang, teilt aber gleichzeitig mit, dass die empfangene Nachricht nicht interpretiert werden konnte. Um die Effizienz des Nachrichtenaustausch weiter zu optimieren, erlaubt CoAP bei einer Request-Response-Kommunikation die Nutzlast der Antwort direkt im ACK-Paket zu versenden. Dadurch wird die Anzahl der benötigten Nachrichten deutlich reduziert.

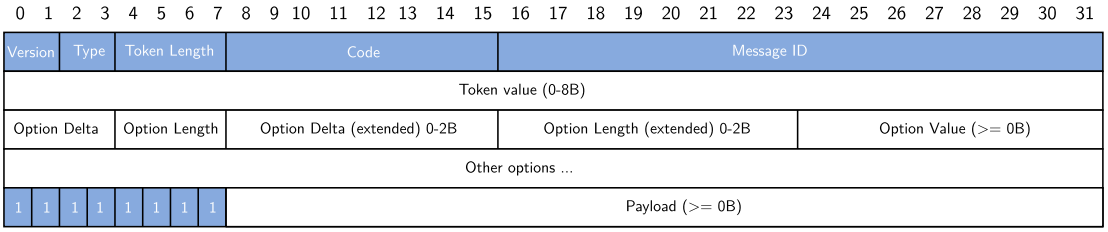


Abbildung 2.23: CoAP Nachrichtenformat

Die Ähnlichkeit zu HTTP wird deutlich, wenn die weiteren wesentlichen Merkmale von CoAP analysiert werden: Ressourcen als Informationsträger, die Manipulation dieser Ressourcen über die Methoden *GET*, *PUT*, *POST* und *DELETE*, Response Codes, die Media Types, URLs und Caching. All diese Konzepte sind durch das HTTP bekannt. Durch diese Gemeinsamkeiten ist es möglich, CoAP direkt in HTTP umzusetzen. In der CoAP-Spezifikation wurden dazu spezielle Proxy-Mechanismen, so genannte *Intermediaries*, definiert, so dass eine nahtlose Kommunikation zwischen Ressourcen beschränkten CoAP-Umgebungen und HTTP-basierten Internet-Umgebungen möglich ist (siehe Abbildung 2.24). Durch die Nutzung von UDP bleibt CoAP nicht wie HTTP auf Unicast-Nachrichten beschränkt, sondern kann auch Multicast-Nachrichten versenden und empfangen. Über die Intermediaries ist es dann sogar möglich einzelne HTTP-Anfragen in mehrere CoAP-Anfragen (über UDP-Multicast-Nachrichten) zu überführen und die Antworten wieder in eine einzelne HTTP-Antwort zu aggregieren. Dadurch werden die möglichen Kommunikations-Pattern um eine Gruppenkommunikation erweitert.

In HTTP sind Transaktionen jeweils vom Client initiiert. Dieser muss über einen Polling-Algorithmus GET-Operationen versenden um über den Status einer Ressource informiert zu bleiben. In Ressourcen beschränkten Umgebungen ist dieser Mechanismus zu teuer,

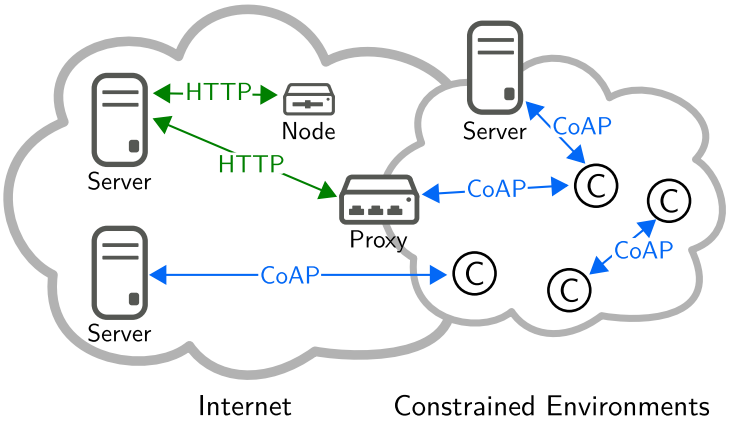


Abbildung 2.24: Kooperierende Kommunikation von CoAP und HTTP zwischen ressourcenbeschränkten und traditionellen Internet-Umgebungen [Bor12]

so dass CoAP eine asynchrone Benachrichtigung von Statusänderungen einführt und somit eine Event-basierte Kommunikation unterstützt [Har13]. Über GET-Nachrichten kann ein Client beim Server Interesse an einer Ressource anmelden und bekommt fortan entsprechende Notifikationen zugeschickt.

Weiterhin unterstützt CoAP rudimentäre Mechanismen zur Beschreibung von angebotenen Diensten, Daten (Metadaten) und dem Suchen und Finden von Geräten (Discovery).

DPWS Protokolle wie HTTP und CoAP eignen sich in besonderem Maße für das Austauschen von Nutzdaten und sind für die Verwendung von statischen Inhalten, wie z. B. Webseiten, optimiert. Da der Header dieser Protokolle klein und einfach gehalten, aber daher auch limitiert ist, können komplexere und anwendungsspezifische Kontroll- und Steuerungsinformationen, wie sie für dynamische Anwendungen im Machine-to-Machine-Umfeld benötigt werden, nicht umgesetzt werden. Diese Informationen müssen folglich innerhalb der Nutzlast einer Nachricht ausgeliefert werden. Ein standardisierter Ansatz für dieses Vorgehen existiert bei den genannten Protokollen aber nicht.

Eine Protokollfamilie, die auf diesem Konzept aufbaut, wurde als Webservices bereits in Absatz 2.1.5 vorgestellt und findet mit dem *Devices Profile for Web Services (DPWS)* Anwendung für leichtgewichtige Systeme. DPWS wird in seiner aktuellen Version 1.1 bei OASIS gepflegt [Nix09b]. Es definiert eine minimale Menge von WS-*-Spezifikationen und schränkt diese zum Teil sogar noch ein, um eine Geräte-basierte Kommunikation auf eingebetteten Systemen zu ermöglichen.

Durch den Einsatz in Geräte-basierten Umgebungen ähnelt DPWS grundsätzlich dem *Universal Plug and Play (UPnP)* [Don08] und wird irrtümlicherweise häufig als dessen Nachfolger bezeichnet. Im Gegensatz zu UPnP basiert das DPWS auf aktuellen Standards und Spezifikationen des W3C und von OASIS, wodurch die Interoperabilität mit Systemen, welche die Web Service Architecture [Boo04] umsetzen, gewährleistet ist.

Die angesprochene Gerätebasierung ist zentraler Bestandteil der DPWS-Architektur.

Ein Gerät (Device) in DPWS ist nicht zwingend als ein physikalisches Gerät zu verstehen sondern eher als ein Container für Dienste. So kann es auch vorkommen, dass Dienste von verschiedenen physikalischen Geräten zu einem virtuellen Device zusammengefasst werden. Zur besseren Unterscheidung wird ein DPWS-konformes Gerät auch als *Hosting Service* bezeichnet. Solche Hosting Services beinhalten eine spezifizierte Menge an Metadaten, die das repräsentierte Gerät beschreiben. Die Beschreibung erfolgt entsprechend der WS-MetadataExchange-Spezifikation und umfasst eine Menge an verschiedenen *Metadata Sections*. In diesen Bereichen werden unter anderem der Hersteller, die Bezeichnung oder Kategorisierungen abgelegt. Zusätzlich umfasst die Beschreibung eines Gerätes auch die Referenzen auf die angebotenen Dienste. Die Dienste, die innerhalb eines Hosting Service zusammengefasst werden, tragen die Bezeichnung *Hosted Service*. Diese Services sind konform zu den SOAP-Webservices und können von WS-Clients auch außerhalb des DPWS-Umfeldes verwendet werden. Die Beschreibung von Hosted Services erfolgt im Gegensatz zu den Hosting Services mittels der WSDL. Innerhalb der DPWS-Welt können nur der Hosting Service und seine Beschreibungen von Clients mittels WS-Discovery gesucht und gefunden werden.

Eine Übersicht des DPWS-Protokollstapels findet sich in Abbildung 2.25. Insbesondere umfasst der Stapel Protokolle zur Realisierung eines dynamischen Discovery, eines Eventing und einer Beschreibung von Metadaten. Der Transport der SOAP-Dokumenten zwischen zwei Endpunkten ist innerhalb des SOAP-Messaging-Framework [Gud07] beschrieben. Ein konkretes Transportprotokoll wird dabei nicht angegeben, vielmehr ist eine Entkopplung von diesen realisiert. Das konkret verwendete Protokoll wird erst vom Dienst-Entwickler innerhalb eines *Binding* definiert. Für das DPWS sind die beiden Transport-Bindings *SOAP-over-UDP* und *SOAP-over-HTTP* ein De-facto-Standard.

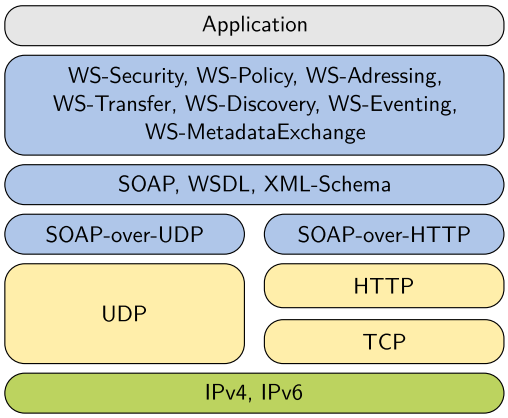


Abbildung 2.25: Der DPWS Protokollstapel

In SOAP-over-UDP werden die SOAP-Nachrichten einem UDP-Datagramm als Nutzlast hinzugefügt. Da das UDP-Protokoll zustandslos und verbindungslos ist, kann die SOAP-Nachricht nicht auf mehrere Nachrichten aufgeteilt werden. Dadurch ergibt sich eine Beschränkung der SOAP-Nachrichtengröße auf die maximale Größe eines UDP-Datagramms (64 kB, abzüglich Größe des Headers). Ein besonderer Vorteil in der Verwendung von UDP liegt in der Möglichkeit der Multicast-basierten Kommunikation. Dieser

Mechanismus ermöglicht die asynchrone Kommunikation mit mehreren Teilnehmern, wobei der Sender keine Kenntnisse über die Empfänger der Nachricht benötigt. Hiermit wird in DPWS das Discovery und das Eventing realisiert. Die asynchrone Kommunikation von UDP ermöglicht keine einfache Umsetzung des *Request-Response*-Transportmuster. Die Zuordnung von Anfragen und Antworten muss auf höheren Transportschichten realisiert werden, genau wie die Adressierung von Diensten.

Das *SOAP-over-HTTP*-Binding basiert auf dem HTTP-Protokoll. HTTP wurde durch die IETF spezifiziert [Fie99] und hat durch die Browser-basierte Interaktion mit Internetseiten einen hohen Bekanntheitsgrad erreicht. HTTP selbst ist bereits ein Protokoll der Anwendungsschicht, kann aber auch zum Transport von Daten eines anderen Protokolls der Anwendungsschicht verwendet werden. Dabei spricht man von einem Transporttunnel. Dieser Tunnel-Mechanismus verhalf SOAP-over-HTTP zum meist verwendeten Transport-Binding, da dadurch das Überwinden von Firewall-Mechanismen durch SOAP-Nachrichten vereinfacht wurde. HTTP selbst basiert auf dem TCP-Protokoll und sichert dadurch einen verlässlichen Nachrichtenaustausch. Auch das Adressieren von Diensten ist, im Gegensatz zu UDP, durch Angaben im HTTP-Header möglich. Die Nachrichtengröße ist durch die Fragmentierung von TCP nicht beschränkt. Allerdings kann von HTTP nur eine synchrone Unicast-Kommunikation realisiert werden. Für Discovery- und Eventing-Mechanismen ist daher das UDP-Protokoll unverzichtbar.

Neben dieser Standard-Transportbindungen existieren in der Forschung Ansätze für effizientere Transportbindungen um das DPWS besser im Bereich Ressourcen beschränkter Netzwerke verwenden zu können. Moritz [Mor12] führt in seiner Arbeit ein neues Transport-Binding für SOAP-Dokumente ein, das auf dem CoAP-Protokoll (siehe Absatz 2.1.7) basiert. *SOAP-over-CoAP* ist an die Limitierungen von Ressourcen beschränkten Netzwerken angepasst und bietet dennoch einen vergleichbaren Funktionsumfang wie *SOAP-over-HTTP*.

Insgesamt bietet das *SOAP-over-CoAP*-Binding vier wesentliche Vorteile gegenüber dem SOAP-over-UDP-Binding:

- Erhöhte Effizienz durch Basierung auf dem UDP-Protokoll
- Kompression der Nutzdaten mittels EXI
- Signifikant kleine Header-Größe
- Zuverlässiger Nachrichtenaustausch

CoAP basiert im Gegensatz zu HTTP auf dem leichtgewichtigen UDP, beinhaltet dadurch aber keinen auf der Transportschicht sichergestellten Mechanismus gegen Nachrichtenverlust. Dieser muss zusätzlich auf der Anwendungsschicht realisiert werden. Dazu werden als erstes die *WS-Addressing* IDs, die jede SOAP-Nachricht enthält, im `token`-Feld der CoAP-`options` gespeichert, um so eine Identifikation der Nachricht und eine Zuordnung von Anfrage-Nachricht und Antwort-Nachricht bereits auf CoAP-Ebene zu ermöglichen. Die Transportzuverlässigkeit wird dann über die CoAP-eigenen Nachrichten (NON, CON und ACK) erreicht. Abbildung 2.26 zeigt die verschiedenen Transportmodi des Bindings am Beispiel des *Request-Response*-Transportmuster.

Der einfachste Fall unterstützt keinen zuverlässigen Transport. Die Anfrage wird mittels einer NON-Nachricht versendet, ebenso die Antwort. Hier unterscheidet sich das

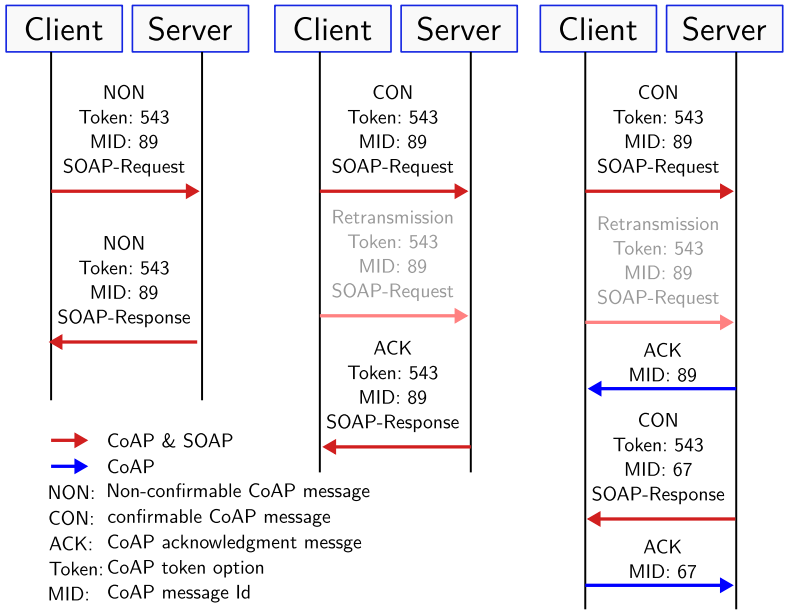


Abbildung 2.26: Transportszenarien des SOAP-over-COAP-Binding [Mor12]

Verhalten nicht vom *SOAP-over-UDP*-Binding. Im zweiten Fall wird für die Anfrage eine CON-Nachricht versendet. Wenn diese verloren geht, wird die Nachricht erneut versendet. Die ACK-Nachricht enthält direkt die SOAP-Antwort als Nutzlast. Im dritten Fall wird die Bestätigungs-Nachricht von der Antwort entkoppelt. Falls das Erzeugen der Antwort einen längeren Zeitraum dauert, kann der Empfang der Anfrage bereits asynchron über ein CoAP-ACK quittiert werden. Die Antwort wird zu einem späteren Zeitpunkt versendet. Weiterhin sind alle Kombinationen aus diesen drei Fällen anwendbar.

Kompressionsverfahren

Webservices und somit auch das DPWS verwenden SOAP als Nachrichtenformat und folglich XML als Datenrepräsentation, um so der Anwendung in heterogenen Umgebungen gerecht zu werden. Ein großer Kritikpunkt an der Verwendung von XML ist aber immer wieder der große Overhead, der gerade in Umgebungen leichtgewichtiger Systeme zu einem entscheidenden Nachteil werden kann. Verursacht wird der Overhead durch die typischerweise verwendete Unicode-Kodierung der Nachrichten. Aufgrund dieser Kodierung werden alle Informationsträger (Tags, Werte, Namensräume etc.) als Zeichenkette dargestellt. Diese Form der Darstellung hat den Vorteil menschenlesbar zu sein, was aber in dynamischen M2M-Anwendungen eigentlich zu vernachlässigen ist. Der Nachteil der Unicode-Kodierung liegt klar auf der Hand. Die Informationsträger können – verursacht durch den Konflikt zwischen der Menschenlesbarkeit und der Nachrichtengröße – eine für die effiziente Kommunikation nicht optimale Benennung haben. Das Problem kann durch die Verwendung von XML-Namespaces weiter verschärft werden. Namespaces werden von den WS-Spezifikationen verwendet um Überlappungen von Bezeichnungen innerhalb eines Dokuments zu vermeiden. Ein Namensraum wird innerhalb eines XML-Dokuments

über *Uniform Resource Identifier (URI)* referenziert und hat in der Praxis eine typische Länge von 40–100 Bytes. Da z. B. DPWS auf mehreren WS-Spezifikationen aufsetzt, ist die Verwendung von Namesräumen obligatorisch, die dann einen teilweise großen Anteil an der gesamten Nachrichtengröße einnehmen. Diese Problematik kommt gerade in Ressourcen beschränkten Umgebungen zum Vorschein, in denen häufig Sensoren zum Einsatz kommen, deren Dienste einfache Zahlenwerte übermitteln. Hier rutschen Nutzdaten und Nachrichtengröße in ein besonders schlechtes Verhältnis ab. Neben dem Overhead bzgl. der Nachrichtengröße gibt es einen weiteren Overhead in der Verarbeitung der Nachricht. Da durch die Kodierung als Zeichenkette nicht zwischen einzelnen Datentypen unterschieden werden kann, muss bei der Verarbeitung eine Typumwandlung der Werte vorgenommen werden. Insgesamt erscheint der Einsatz von XML in Ressourcen beschränkten Umgebungen daher nicht optimal.

Aus dieser Motivation heraus existieren verschiedene Forschungsansätze um den Einsatz von XML in Ressourcen beschränkten Umgebungen zu ermöglichen. Innerhalb der Webservice-Welt darf eine effizientere Darstellung der Nachrichten nicht am *SOAP Binding Framework* [Gud07] vorbei geplant werden. Dieses verlangt eine Serialisierung von SOAP-Dokumenten als XML-Infoset, die Kodierung ist aber nicht festgelegt. Betrachtet man zusätzlich die DPWS-Spezifikation, so wird hier lediglich verlangt, dass ein Dienst mindestens die Unicode-basierte Kodierung von SOAP-Dokumenten unterstützen muss. Weitere Kodierungen sind allerdings möglich. Für eine effizientere, d. h. komprimierte Darstellung von SOAP-Nachrichten muss also auf der einen Seite die Serialisierung bezüglich. XML-Infosets unterstützt werden, um die Konformität mit der WS-Architecture und weiterführenden Spezifikationen zu sichern. Auf der anderen Seite müssen die einzelnen Informationsträger von der Unicode-Kodierung gelöst und durch effizientere Kodierungen ersetzt werden.

Die existierenden Kompressionsverfahren in diesem Umfeld wurden in der Arbeit von Moritz [Mor12] übersichtlich zusammengefasst und können grob in zwei Klassen unterteilt werden, die generische Kompression und die XML-spezifische Kompression. Bei der generischen Kompression wird die Nachricht als rein binärer Datenstrom behandelt, wodurch das Verfahren nicht auf die Kompression von Nachrichten beschränkt ist. Grob betrachtet arbeitet das Verfahren mit einer Vermeidung von Bytefolgen-Wiederholungen. Die Bytefolgen werden im originalen Datenstrom identifiziert und mit binären Schlüsselwerten belegt. Im komprimierten Datenstrom kommen diese Bytefolgen dann nur noch einmal vor und werden ab dann über die Schlüsselwerte referenziert. Bei den XML-spezifischen Kompressionsverfahren wird aus den XML-Dokumenten der Informationsgehalt extrahiert und effizienter dargestellt. Dabei werden auf der einen Seite Teile vermieden, die keinen Informationsgehalt bieten – wie z. B. Leerzeichen, Zeilenumbrüche, Kommentare etc. – und auf der anderen Seite die XML-Informationsträger selbst behandelt. Hier kann erneut eine Unterteilung von verschiedenen Verfahrensklassen vorgenommen werden. Es existieren einfache Kompressionsverfahren, Verfahren, die ein bekanntes Vokabular voraussetzen und Automaten-basierte Verfahren.

Einfache XML-spezifische Kompressionsverfahren setzen ähnlich wie generische Kompressionsverfahren auf die Vermeidung von Redundanzen, allerdings beschränken sich diese Verfahren auf die Informationsträger und komprimieren den eigentlichen Inhalt

```xml
<register>
 <person>
  <name>Hans Mueller</name>
  <age>33</age>
  <gender>M</gender>
 </person>
 <person>
  <name>Heike Scholz</name>
  <age>33</age>
  <gender>F</gender>
 </person>
</register>
```

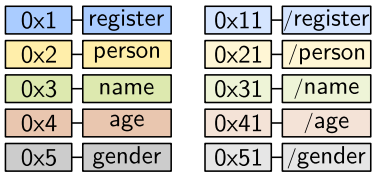
(a) XML-Instanz

```xml
<xs:simpleType name="gType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="M"/> <xs:enumeration value="F"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="pType"> <xs:sequence>
  <xs:element name="name"    type="xs:string" min="1"/>
  <xs:element name="age"     type="xs:string" min="0" max="1"/>
  <xs:element name="gender"  type="gType" min="0" max="1"/>
</xs:sequence> </xs:complexType>

<xs:complexType name="registerType"> <xs:sequence>
 <xs:element name="person" type="pType" min="0"/>
</xs:sequence> </xs:complexType>
```

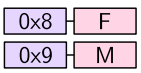
(b) XML-Schema

Listing 2.1: Eine XML-Instanz mit dem dazugehörigen XML-Schema

nicht. Das für die Referenzierung benötigte Vokabular wird zur Laufzeit während der Verarbeitung des eigentlichen Datenstroms gebildet. Abbildung 2.27(a) zeigt ein exemplarisches XML-Dokument und Abbildung 2.28(a) den zugehörigen Vokabular-Stack. Im komprimierten Datenstrom findet sich nun jeder Informationsträger genau einmal als Zeichenkette wieder, und wird fortan über binäre Schlüsselwerte referenziert (siehe Abbildung 2.29(a)). Genau hier setzen Verfahren an, die das Vokabular vor der Verarbeitung der Nachricht generieren. In diesem Fall enthält der komprimierte Datenstrom in den Informationsträgern keinerlei Zeichenketten mehr, sondern besteht ausschließlich aus binären Referenzen (siehe Abbildung 2.29(b)). Die Informationen für die Generierung des Vokabulars werden im Vorfeld aus Schemabeschreibungen wie einer *Document Type Definition (DTD)* oder eines – von der WS-Architecture verbindlich vorgeschriebenen – XML-Schemas, für das Beispiel exemplarisch in Abbildung 2.27(b) dargestellt, extrahiert.



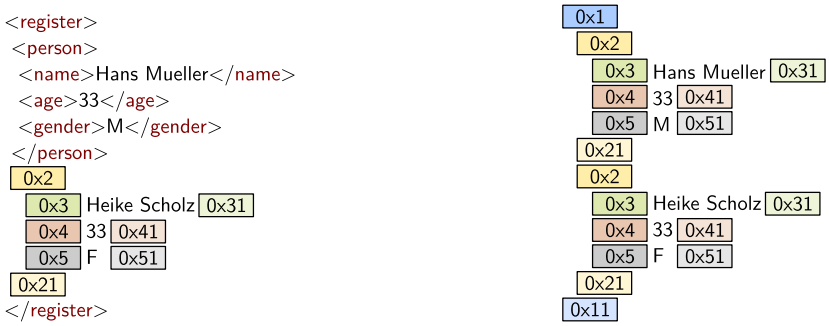
(a) Vokabular-Stack aus Informationsträgern



(b) Erweitertes Vokabular um inhaltliche Elemente

Abbildung 2.28: Vokabular des XML-Dokuments 2.1

Eine weitere Optimierung XML-spezifischer Kompressionsverfahren setzt neben einem bekannten Vokabular auch eine bekannte Grammatik des XML-Dokuments voraus. Mit Hilfe dieser Vorkenntnisse kann bereits im Vorfeld der Nachrichtenverarbeitung ein Akzeptor generiert werden. Der Automat für das bekannte Beispiel ist in Abbildung 2.30 illustriert. Die Zustandsmenge dieses Automaten umfasst neben den einzelnen Informationsträgern auch inhaltliche Elemente (siehe Abbildung 2.28(b)), wenn sie innerhalb der Schemadefinition (siehe Abbildung 2.27(b)) vorgegeben sind. Die Zustandsübergänge werden durch die vorhandene Grammatik definiert. Der Datenstrom kann nun auf unter-



(a) Einfache XML-spezifische Kompression **(b)** Kompression mit bekanntem Vokabular

Abbildung 2.29: Gegenüberstellung von XML-spezifischer Kompression mit und ohne bekanntem Vokabular (aus Abbildung 2.28(a))

schiedliche Arten kodiert werden. Zum einen besteht die Möglichkeit die Zustände des Automaten direkt zu kodieren, zum anderen reicht aber auch die Kodierung der Zustandsübergänge aus. Da die Menge der Zustandsübergänge eines einzelnen Zustands wesentlich geringer ist als die Menge der gesamten Zustände des vollständigen Automaten, ist dadurch eine effizientere Kodierung möglich. Nachteilig wirkt sich allerdings die Tatsache aus, dass bei dieser Art der Kodierung die Nachricht von Anfang an verarbeitet werden muss, da der für die Ableitung der Zustände der Kontext (Position im XML-Dokument) bekannt sein muss.

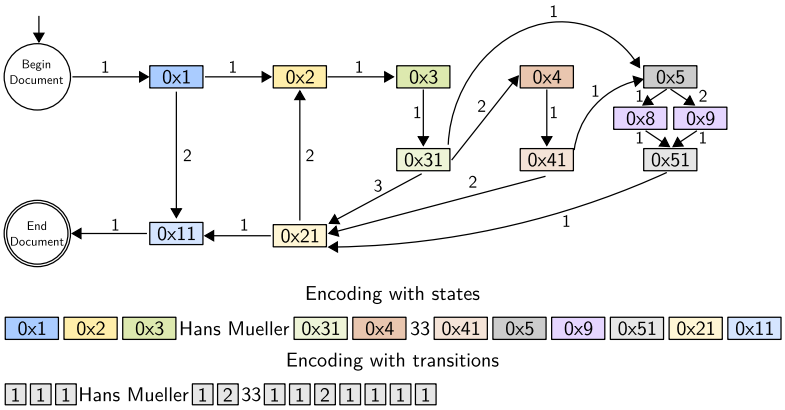


Abbildung 2.30: Automaten-basiertes Verfahren zur XML-Komprimierung

Abbildung 2.31 zeigt eine Übersicht gebende Klassifizierung der verschiedenen Kompressionsverfahren und deren konkrete Umsetzungen. Die umfangreiche Welt der generischen Kompressionsverfahren wird unter anderem von Salomon in [Sal06] dargestellt. Für den Einsatz in Web-Umgebungen hat sich vor allem der gzip-Algorithmus [Deu96] etabliert. Hier erreicht der Algorithmus eine gute Kompressionsrate, da die zu übertragenden HTML-Dateien ein sehr begrenztes Vokabular haben und somit eine erhöhte Wiederholung von Zeichenketten. Weitere bekannte Vertreter der generischen Kompressionen sind rar, zip und arj. Die wichtigste Entwicklung im Bereich der einfachen XML-spezifischen Kompressionen wurde im Jahr 2000 unter dem Namen XMill publiziert [Lie00]. XMill

versteht sich selbst vor allem als Ergänzung zu den generischen Kompressionsverfahren, da diese auf einen XMill-kodierten Strom angewendet eine noch kompaktere und rein binäre Form des Datenstroms erzeugen.

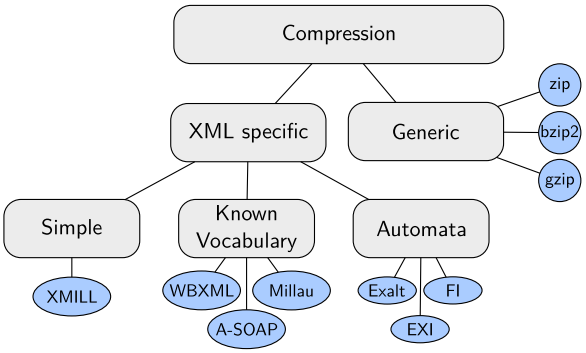


Abbildung 2.31: Klassifizierung der unterschiedlichen Kompressionsverfahren

Adaptive SOAP [Ros07] ist als Verfahren mit bekanntem Vokabular eingeordnet, arbeitet auf den ersten Blick aber wie ein einfaches XML-spezifisches Verfahren. Während der Kommunikation zwischen zwei Endpunkten entsteht auf beiden Seiten ein gemeinsames Vokabular. Die Informationen werden mit binären Schlüsselwerten versehen und können von nun an referenziert werden. Das Besondere ist, dass dieses Vokabular über mehrere Nachrichten hinweg bekannt bleibt, und die Kommunikation im Laufe der Zeit immer effizienter wird. Dieses Verfahren ist für SOAP-basierte Webservices besonders geeignet, da sich in aufeinander folgenden SOAP-Nachrichten viele Teile wiederholen. Das WAP Binary XML (WBXML) Verfahren [Mar99] hingegen bezieht sein Vokabular durch die Spezialisierung auf konkrete XML-basierte Beschreibungssprachen, vor allem auf die *Wireless Markup Language (WML)*. Somit ist das Vokabular schon in der Spezifikation festgelegt. Millau [Gir03] erweitert das Konzept von WBXML um die Behandlung von unbekannten Strukturelementen.

Fast Infoset [Int05] ist ein bekannter Vertreter der Automaten-basierten Kompressionsverfahren und generiert vor der Nachrichtenverarbeitung Automatenstrukturen. Das Vokabular und die Grammatik werden aus der Schemabeschreibung extrahiert. Im Gegensatz zu Exalt [Tom04] verwendet Fast Infoset neben dem Informationsgehalt aus den XML-Strukturen zusätzlich die konkrete Serialisierungsform. Die aufkommende Bedeutung von Kompressionsverfahren im Bereich von SOAP und XML wurde auch vom W3C erkannt und von 2004 an innerhalb der *XML Binary Characterization Working Group* und ab 2005 in der *Efficient XML Interchange Working Group* bearbeitet. Hier entstand das bekannte Kompressionsverfahren *Efficient XML Interchange (EXI)* [Sch08], eine offizielle W3C-Spezifikation.

In EXI werden aus der bekannten Grammatik und dem bekannten Vokabular, also aus den Informationen des XML-Schemas, Automaten erzeugt, die als *EXI-Grammar* bezeichnet werden. Dadurch entsteht ein EXI-Datenformat, in dem lediglich Informationen über die Kanten zwischen den Zuständen des Automaten enthalten sind. Während der Verarbeitung eines *EXI-Streams* ist der Automat zusätzlich in der Lage auch neue, bisher unbekannte Strukturen zu erlernen und anschließend zu verarbeiten. Im einfachsten Fall

hat der Automat zu Beginn keinerlei Kenntnisse über die zu erwartenden Strukturen. Alles wird neu erlernt. Dieser Modus wird auch als *schema-less* bezeichnet. Dem gegenüber steht der *schema-informed*-Modus, in dem, wie beschrieben, der Automat bereits im Vorfeld erzeugt wird.

Das Auftreten von Übergängen im Automaten wird als *EXI-Event* bezeichnet, so dass ein EXI-Stream aus EXI-Events und den Elementinhalten besteht. Innerhalb des Datenstroms können die Events und die Daten in separaten Containern eingebettet werden, so dass sie nicht direkt aufeinander folgen müssen. Durch diesen Mechanismus können generische Kompressoren besonders effizient auf den Datenstrom angewandt werden.

Eine weitere Besonderheit von EXI ist die Kodierung der Automatenzustände und Übergänge. Da für die meisten möglichen Zustände des EXI-Automaten deutlich weniger als 256 EXI-Events möglich sind, weicht EXI von der üblichen Byte-Orientierung ab und nutzt zur Kodierung lediglich exakt so viele Bits, wie notwendig sind. Dabei werden, ähnlich wie bei der Huffmann-Kodierung, häufig auftretende Events mit weniger Bits kodiert als selten auftretende Events. Dieser Modus wird als *bit-aligned* bezeichnet und kann durch den *byte-aligned*-Modus ersetzt werden, in dem immer auf volle Bytes aufgefüllt wird.

Programmierung

Für die Programmierung von verteilten Anwendungen in Ressourcen beschränkten Umgebungen gibt es eine Reihe von möglichen Programmiersprachen. Im Objekt-orientierten Umfeld haben sich Java und .Net etabliert, die konzeptionell sehr ähnlich sind und beide auch für den Einsatz auf Ressourcen beschränkten Geräten spezielle Laufzeitumgebungen anbieten.

In Java ist für diese Systeme besonders die *Java Micro Edition (J2ME)* von Interesse. Die Java Micro Edition definiert die Umsetzung der Programmiersprache Java für eingebettete Systeme, wie beispielsweise PDAs und Handys. Einen Überblick über die Java-ME-Komponenten und ihre Beziehung zur gesamten Java-Plattform ist in Abbildung 2.32 dargestellt. Die Java Micro Edition unterscheidet zusätzlich noch die beiden Profile *Connected Device Configuration (CDC)* und *Connected Limited Device Configuration (CLDC)*, die beide minimale Laufzeitumgebungen für die Ausführung von Java-Programmen für unterschiedliche Klassen von Geräten definieren. Im OSGi-Umfeld muss mindestens eine CDC-Laufzeitumgebung vorliegen, da KVM, auf der das CLDC basiert, nur einen einzelnen Classloader anbietet, und somit wesentliche Merkmale einer OSGi-Plattform nicht unterstützt werden können.

Ähnlich zu dieser Idee bietet .Net mir dem *.Net Compact Framework* eine Umgebung an, die ebenso speziell für die Nutzung auf mobilen Endgeräten ausgerichtet ist. Auch hierbei handelt es sich um eine Laufzeitumgebung, welche die normale .Net-Umgebung um eine größere Anzahl von Klassen reduziert, die nicht für den Einsatz auf Ressourcen beschränkten Geräten optimiert sind, da sie z. B. zu viel Speicherplatz belegen.

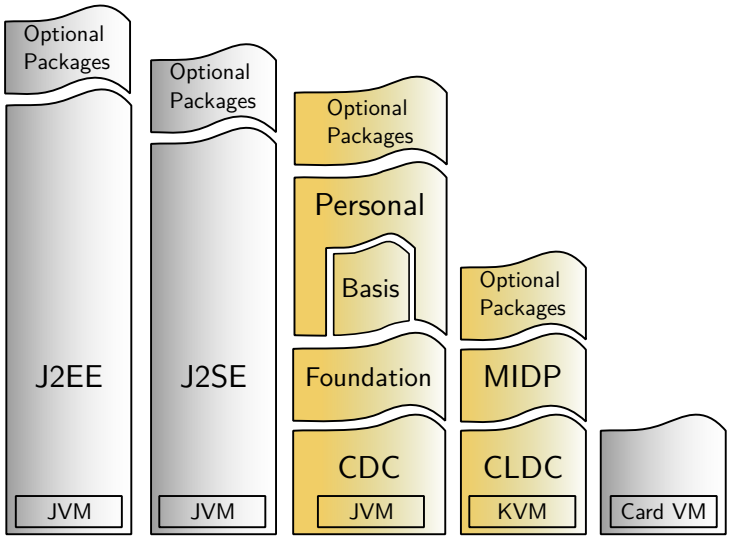


Abbildung 2.32: JavaME-Komponenten

2.1.8 Softwarequalität

Die wirtschaftliche Bedeutung von herausragenden Produktqualitäten ist im Bereich von Ingenieursprodukten weltweit anerkannt. Softwareprodukte allerdings stehen im Ruf häufig Qualitätsprobleme zu haben, obwohl die wirtschaftliche Bedeutung der Qualität von Software der von Ingenieursprodukten in nichts nachsteht. Fehlerhafte Software kann einen hohen Schaden verursachen, und eine hohe Qualität kann im Konkurrenzkampf der Unternehmen einen entscheidenden Einfluss haben.

Um Softwarequalität zu definieren muss zuerst ein allgemeines Verständnis von Qualität vorliegen. Geiger definiert Qualität als „Relation zwischen realisierter Beschaffenheit und geforderter Beschaffenheit“ [Gei08]. Ähnlich, jedoch mit einem stärkeren Kundenbezug, versteht Reeves Qualität als „Erfüllen oder Übertreffen von Kundenerwartungen“, und dem gegenüber, aus der Sicht der Entwickler, als „Einhaltung von Spezifikationen“ [Ree94]. In der Arbeit von Shewhart entspricht Qualität sehr allgemein „allen charakteristischen Eigenschaften eines Gegenstandes“ [She31]. Allein aus diesen Aussagen ist leicht zu erkennen, dass es verschiedene Blickwinkel gibt, aus denen Softwarequalität betrachtet werden kann. Garvin [Gar84] beschreibt diese explizit in seinem Papier.

All diese Definitionen lassen sich auch auf die Softwarequalität übertragen. Allerdings besteht weiterhin die Fragestellung, welche Attribute für eine gute Softwarequalität entscheidend sind. Qualitätsmodelle bieten eine Abstraktion von Merkmalen, Kriterien und Eigenschaften um Softwarequalität systematisch zu definieren, zu beurteilen und zu messen. Dabei existieren Standards, wie die ISO-Norm 9126 [ISO01], das Qualitätsmodell nach McCall [McC77], firmeninterne Richtlinien und akademische Ansätze [Dei07; Fra12]. Sowohl McCall, als auch die ISO-Norm definieren dabei eine Anzahl von unterschiedlichen Qualitätsmerkmalen (wie z. B. Zuverlässigkeit, Funktionalität, Effizienz, Wartbarkeit etc.), die durch messbare Kriterien verfeinert werden.

Qualitätsmodelle bieten einen Ansatz, wie Softwarequalität definiert und gemessen

werden kann. Allerdings wird Software in den verschiedensten Bereichen eingesetzt, in denen viele, sehr unterschiedliche Anforderungen gestellt werden. Die normierten Modelle bieten keinen hinreichenden Ansatz um auf diese individuellen Anforderungen und auf den individuellen Entwicklungsprozess eines Software-Systems einzugehen. Daher müssen Qualitätsmodelle in den jeweiligen Software-Projekten angepasst verwendet werden. Der Grad der Anpassung bewegt sich dabei in einem Spektrum zwischen der kompletten Eigendefinition eines Qualitätsmodells und der Verwendung eines bestehenden Modells. Bei der Eigendefinition geben Metamodelle, wie im SQUID-Ansatz [Kit97] definiert, einen Rahmen zur Modellierung vor.

Um die Qualitätsmerkmale innerhalb der aufgestellten Modelle zu erfassen und vergleichbar zu machen, müssen diese operationalisierbar sein, also vermessbar und durch Metriken und Maße beschreibbar sein. Eine Softwaremetrik [Gil77] ist eine Quantifizierung von Software-Eigenschaften, die auf verschiedenen Skalentypen liegen können (beispielsweise eine Ordinalskala “niedrig”, “mittel”, hoch“) und dadurch nur eingeschränkt operationalisierbar sind. Lediglich Vergleichsoperationen, aber keine Multiplikationen oder Substraktionen von Werten sind erlaubt. Diese Lücke wird durch Softwaremaße [Mun02] geschlossen. Im Gegensatz zur Metrik besitzen Maße eine verhältnisskalierte Eigenschaft und eine Maßeinheit und sind damit vollständig operationalisierbar.

Softwaremessungen ermitteln diese Kennzahlen und wurden schon frühzeitig untersucht [McC76]. Heutzutage sind sie Bestandteil umfangreicher Messsysteme. Hier bietet die ISO-Norm 15939 [ISO02] einen geeigneten Ansatz. Mit der gleichen Argumentation wie die gegen statische Qualitätsmodelle müssen auch Messsysteme individualisiert werden. Generische Vorgangsweisen zur Entwicklung angepasster Messsysteme werden durch die Methoden *Factors-Criteria-Metrics (FCM)* [McC77] und *Goal/Question/Metric (GQM)* [Bas92] eingeführt. Im GQM-Ansatz werden die Messziele über Fragen auf konkrete Informationsbedürfnisse heruntergebrochen, die anschließend auf Metriken abgebildet werden. Innerhalb der Messbewertung, gestützt durch statistische Verfahren [Sin12], kann anschießend die Softwarequalität bewertet und gleichzeitig auch rekursiv gezeigt werden, ob die verwendeten Messmethoden sich auch künftig für treffende Bewertungen eignen.

Die gewonnenen Daten werden heutzutage nicht nur zur Analyse von Softwarequalität genutzt, sondern auch immer häufiger zur Sicherstellung der Einhaltung konkreter Unternehmensziele eingesetzt. Messziele werden mit Unternehmenszielen verknüpft und die gewonnenen Daten können anschließend zur Entscheidungsfindung und Steuerung des Entwicklungsprozesses eingesetzt werden. Dieser Aspekt wird durch die Ansätze CMMI [Chr03] und GQM+Strategy [Bas07] unterstützt.

2.1.9 Technisches Management

Die Verwaltung verteilter Geräte- und Service-Systeme zur dynamischen Anpassung an wechselnde Bedingungen, ist in den betrachteten Domänen von entscheidender Bedeutung, da statische Systeme die gestellten Anforderungen nicht umsetzen können. Die Bestrebungen das technische Management zu standardisieren, führten zu einer Vielzahl unterschiedlicher Management-Konzepte:

OSI-Management

Die ersten Standardisierungen unternahm die ISO im Rahmen der Spezifikation des OSI-Referenzmodells statt. Innerhalb des OSI-Managements [ISO89] werden einzelne Problemfelder eines Managementsystems, so genannte *fuctional areas*, auch als *FCAPS* bekannt, identifiziert. In FCAPS werden die möglichen Aufgaben eines Managementsystems in fünf Bereiche aufgeteilt:

- **F**ault Management
  Das Fehlermanagement beschäftigt sich mit dem Entdecken, Eingrenzen und Beheben von abnormalem Systemverhalten. Zu diesem Zweck werden Fehlerprotokolle geführt und ausgewertet, auf Fehlermeldungen reagiert, Ablauffehler erkannt, verfolgt und isoliert, sowie Fehlverhalten korrigiert.
- **C**onfiguration Management
  Das Konfigurationsmanagement beschreibt den Vorgang des Konfigurierens als Aktivität, also das Setzen und Ändern von Parametern, die den Normalbetrieb eines Systems regeln. Weiterhin beinhaltet dieser Punkt die Darstellung der aktuellen Konfiguration.
- **A**ccounting Management
  Das Abrechnungsmanagement sorgt für eine Umlegung auftretender Kosten auf die Kostenverursacher. Dazu werden Verbrauchsdaten erfasst, Kosten und Gebühren den Benutzern zugeordnet und Verbrauchsstatistiken geführt.
- **P**erformance Management
  Das Leistungsmanagement setzt sich zum Ziel, dass das Gesamtsystem im Sinne einer definierten Dienstgüte „gut“ läuft. Dazu werden Ressourcen und Leisungsengpässe überwacht und die gesammelten Daten aufbereitet und ausgewertet.
- **S**ecurity Management
  Mit diesem Bereich ist das Management der Datensicherheit gemäß definierter Sicherheitsvorgaben in einem verteilten System gemeint. Je nach Sicherheitsvorgaben umfasst dieser Punkt Authentifizierung, Durchführung einer Zugriffskontrolle, Sicherstellung der Vertraulichkeit und Sicherstellung der Integrität.

Management-Objekte sind definiert als die OSI-Management-Sicht einer Ressource. Diese verfügen über Attribute zur Abbildung eines Zustandes und über Management-Operationen. Alle Management-Objekte einer Ressource sind in der zugehörigen *Management Information Base* (MIB) hinterlegt, welche die Objekte strukturiert und damit die für das Management relevanten Informationen in einer einheitlichen Sicht anbietet.

Web-Based Enterprise Management (WBEM)

Das *Web-Based Enterprise Management (WBEM)* [09] ist ein Plattform und Ressourcen unabhängiger Standard der *Distributed Management Task Force* (DMTF), der ein gemeinsames Modell für das Management verteilter Systeme definiert. Das *Common Information Model* (CIM) bildet den Kern der WBEM und ist ein objektorientiertes Informationsmodell zur einheitlichen Beschreibung von Management-Informationen und -Funktionen. Durch CIM können physikalische und logische Objekte abgebildet werden, die wiederum zu Klassen zusammengefasst werden können.

Web Services Distributed Management (WSDM)

Das *Web Services Distributed Management (WSDM)* [06] wurde von OASIS für das Management Service-orientierter Architekturen entwickelt und beschreibt einen plattform- und herstellerunabhängigen Ansatz zur Definition von Management-Schnittstellen für Ressourcen. Die WSDM-Spezifikationen unterteilen sich hauptsächlich in zwei Bereiche: Das *Management Using Web Services* (MUWS) behandelt das Management von Ressourcen unter Verwendung von Web Services, das *Management of Web Services* (MOWS) definiert ein Modell für das Management von Web Services.

Policy-basiertes Management

Beim Policy-basierten Management werden abstrakte Policies ausgehend von Unternehmenszielen ermittelt. Diese Policies können in unterschiedlichen Abstraktionsebenen definiert werden und legen das gewünschte Systemverhalten fest [Wie94].

Um die abstrakten Policies in ein Laufzeit-Managementsystem zu verwenden, müssen diese in technische Low-Level Policies übersetzt werden, das so genannte Policy-Refinement. Dieser Ansatz setzt eine eine Top-Down Vorgehensweise um, die ausgehend von den Unternehmenszielen konkrete, technische Regeln ableitet. Beim herkömmlichen Management werden die Unternehmensziele hingegen über einen Bottom-Up Ansatz erreicht.

2.2 Zwischenfazit

Ambiente System bezeichnen vernetzte, eingebettete Systeme mit intelligenter Sensorik und Aktorik. Gegenstände des Alltags werden um Kommunikationsfähigkeit und Intelligenz erweitert, so dass diese Systeme Menschen Hilfe im Bereich des täglichen Lebens anbieten können. In diesem Kapitel wurden Technologien, Paradigmen und Standardisierungen analysiert, die im Bereich ambienter Systeme vorherrschen. Für die Entwicklung von ambienten Anwendungen sind dabei mehrere Aspekte von besonderer Relevanz. Diese Aspekte betreffen die Softwarearchitektur, die Kommunikation und die Softwarequalität.

Für die Softwarearchitektur verteilter Anwendungen wurden verschiedene Stile vorgestellt. Ausgehend vom Client/Server-Modell hat sich das Objektmodell entwickelt, das durch die Implementierung *CORBA* Bekanntheit erlangte. Weitestgehend etabliert haben sich heute aber das Ressourcen-orientierte Modell und das Dienste-orientierte Modell. Das Dienste-orientierte Modell eignet sich besonders zur Vermeidung von Insellösungen. Diese entstehen durch die Verwendung von propriäteren Protokollen und nicht spezifizierten Interfaces, so können Geräte und Software unterschiedlicher Hersteller nicht miteinander interoperieren. Die Anwendung einer SOA, die konträr dazu offen spezifizierte Schnittstellen vorschreibt, unter Verwendung von offenen, wohl definierten Standards, ist ein wichtiger Baustein zur Lösung dieses Problems. Ein offenes Datenformat wie XML fügt einen weiteren Baustein hinzu. Wird dieses Konzept konsequent angewendet, und Insellösungen somit vermieden, werden dem Benutzer neue Möglichkeiten der Anwendungsentwicklung eröffnet. Nunmehr können Domänen übergreifende Anwendungen aus Bereichen wie die Gebäudeautomatisierung, der Automotive-Bereich, oder die Medizintechnik einfach erstellt werden.

Ein weiteres relevantes Architekturkonzept ist die Software-Modularisierung. Die Gliederung einer Software in logische Komponenten ermöglicht die Wiederverwendung von Basisfunktionalitäten und somit die Konzentration auf Neuentwicklungen. Anwendungen können so aus einzelnen Komponenten flexibel zusammengesetzt werden. Dieser Ansatz bringt dem Softwareentwickler einen großen Gewinn hinsichtlich der Produktivität und auch der Fehlerreduzierung. Wiederverwendete Komponenten sind bereits getestet und die Implementierung von kleinen Teilen birgt wesentlich weniger Gefahren für Programmierfehler. Umsetzungen der komponentenbasierten Softwareentwicklung sind die .Net-Plattform, JavaEnterpriseBeans und die OSGi-Plattform. Letztere hat sich in den letzten Jahren im Bereich Java-basierter Anwendungen etabliert und wird ständig weiterentwickelt.

Bei der verteilten Kommunikation werden verschiedene Seiten des OSI-Modells betrachtet. Sensoren und Geräte kommunizieren meistens auf der Sicherungsschicht miteinander. In drahtlosen Sensornetzwerken sind Protokolle wie Bluetooth oder ZigBee gängige Praxis. Eine Homogenisierung der Kommunikation kann bei diesen Technologien erst auf der Anwendungsschicht vollzogen werden. Sollen diese Geräte zudem in eine SOA integriert werden, so müssen Service-Schnittstellen für die Funktionalitäten der Geräte erstellt werden. Lösungen zu dieser Problematik wurden mit der *OSGi-Device-Access*-Spezifikation, IoTSys und dem SODA-Projekt vorgestellt.

Bei der Kommunikation auf der Anwendungsschicht sind bei datenlastigen Anwendungen Protokolle wie CoAP oder HTTP eine bevorzugte Wahl. In komplexen, dynamischen Anwendungen in stark heterogenen Umgebungen wird aber zumeist ein weiterführender Funktionsumfang benötigt. Dieser Umfasst beispielsweise Mechanismen zum Suchen und Finden von Diensten und Geräten (Discovery), asynchrone Benachrichtigungen über Zustandsänderungen (Eventing), Beschreiben von Geräten und Diensten (Description), sowie ganzheitliche Sicherheitskonzepte. In höherwertigen Diensteumgebungen haben sich hierfür die SOAP-Webservices des W3C durchgesetzt. Das Devices Profile for Webservices (DPWS) bildet dabei ein spezielles Protokoll für den Einsatz Webservice-basierter Kommunikation auf Ressourcen beschränkten Geräten.

Eine hohe Softwarequalität sorgt für eine steigende Akzeptanz bei den Benutzern und wird von Softwareentwicklern häufig bereits während der Entwicklung durch systematische Qualitätskontrollen sichergestellt. In diesem Bereich wurden Qualitätsmodelle vorgestellt, deren Eigenschaften, wie Zuverlässigkeit, Funktionalität, Effizienz oder Wartbarkeit, während des Entwicklungsprozesses gemessen und umgesetzt werden. Die Effizienz einer Software spielt dabei eine wichtige Rolle und wird insbesondere in der Kommunikation realisiert. Hierfür bietet sich der Einsatz von leichtgewichtigen Protokollen wie CoAP oder DPWS im Webservice Umfeld an. DPWS verwendet als SOAP als Nachrichtenformat und somit XML als Datenrepräsentation. Für die effiziente Verwendung von XML-basierter Kommunikation wurden verschiedene Kompressionsverfahren vorgestellt, die das Datenaufkommen deutlich reduzieren. Im Bereich der Kommunikation von DPWS-fähigen Geräten hat sich insbesondere das EXI-Verfahren etabliert.

Damit ambiente Systeme dynamisch auf wechselnde Anforderungen reagieren können sind Anpassungen des Programmablaufs zur Laufzeit notwendig. Hierzu sind im Bereich des technischen Management verschiedene Standards etabliert. Der bekannteste wird im

OSI-Management definiert und umfasst die bekannte FCAPS-Unterteilung. Das Web-Based Enterprise Management und das Web Services Distributed Management sind weitere relevante Standards in diesem Bereich. Ein weit verbreitetes Konzept um Systeme für die Umsetzung spezieller Unternehmensziele zu konfigurieren ist das Policy-basierte Management. Hier existieren auch Ansätze um abstrakte Unternehmensziele in technische Konfigurationen abzuleiten.

3 Remote Services

Innerhalb dieses Kapitels wird die Verteilung von verschiedenen OSGi Plattformen grundlegend motiviert und deren Entwicklung in der Wissenschaft dargestellt. Dabei werden sowohl Standardisierungen als auch Forschungsarbeiten betrachtet. Insgesamt gibt das Kapitel Übersicht über folgende Aspekte:

- Welchen Vorteil bietet die Entwicklung verteilter Anwendungen unter Software-Modularisierung?
- Wo liegt die Motivation für die Entwicklung einer verteilten OSGi-Plattform?
- Welche Forschungsansätze existieren in diesem Themenumfeld?
- Wie hat sich die Standardisierung in diesem Bereich entwickelt?

3.1 Problemstellung

Das modulare Software-Design ist in der Softwaretechnik ein wichtiger Eckstein. Durch einen hohen Einsatz an Forschung und Entwicklung von Konzepten zur Verwaltung von Modulen und deren Abhängigkeiten ist das modulare Design weit verbreitet. Neben der Strukturierung von lokalen Systemen wurde auch die Entwicklung und Strukturierung von verteilten Anwendungen immer bedeutender. In der Vergangenheit wurde die Verteilung der Funktionalitäten einer Anwendung über entfernte Methodenaufrufe realisiert (siehe Unterabschnitt 2.1.5). Solche Modelle erfordern aber einen hohen Programmieraufwand, da sich lokale von entfernten Methodenaufrufen deutlich unterscheiden. Dieser Unterschied betrifft vor allem das Implementieren technologieabhängiger Schnittstellen und eine besondere Fehlerbehandlung, z. B. für Netzwerk-bedingte Fehler.

Durch die Einführung einer Middleware-Schicht können technologieabhängige Implementierungen weitestgehend vermieden und allein dadurch der Programmieraufwand

deutlich verringert werden. Eine zusätzliche Hilfe für diesen Aspekt, gerade im Bereich der Entwicklung verteilter Anwendungen bietet die komponentenorientierte Softwareentwicklung. Komponentenplattformen, wie OSGi, sind dazu entwickelt worden das Installieren und Deinstallieren von Komponenten zur Laufzeit zu verarbeiten. An dieser Stelle greifen Event-Mechanismen, die über Verfügbarkeiten von Services informieren. Dieses Modell kann im verteilten Umfeld verwendet werden um Netzwerkfehler in gleicher Weise wie das Entfernen eines Services zu behandeln. Auf diese Weise wird das Programmiermodell einer verteilten komponentenbasierten Anwendung gegenüber einer lokalen Anwendung nicht verändert. Unter diesem Gesichtspunkt können Anwendungen in verteilte Umgebungen überführt werden, indem einfach deren Komponenten auf unterschiedliche Knoten verteilt werden.

Die OSGi-Service-Plattform ist ursprünglich nicht für die Erstellung von verteilten Anwendungen entwickelt worden, sondern realisierte zu Beginn nur eine lokale komponentenbasierte Plattform innerhalb der Grenzen einer Java VM. Im Laufe der Zeit entstanden aber immer mehr Anwendungsfälle, in denen eine Kommunikation zwischen OSGi-Services verschiedener Plattformen wünschenswert, beziehungsweise notwendig wurde. An diesem Punkt setzte die Forschung ein, um die grundsätzlichen Überlegungen und Vorteile einer verteilten Anwendung über Software-Modularisierung für die OSGi-Plattform zu realisieren.

3.2 Historische Entwicklung und initiale Arbeiten

Die Entwicklung von OSGi, von einer lokalen Komponentenplattform hin zu einem Framework zur Erstellung verteilter Anwendungen, lässt sich gut in verschiedene Entwicklungsstränge strukturieren. Abbildung 3.1 zeigt diese Strukturierung. Auf oberster Ebene kann eine Unterteilung in Forschungsarbeiten und in Realisierungen von bestehenden Standards vorgenommen werden. Die Forschungsarbeiten stellen neue Konzepte und Ideen vor und liefern zudem grundsätzliche Ansätze für die Standardisierung. Die Ergebnisse der Forschungsarbeiten lassen sich in jene, die volle Kompatibilität zum OSGi-Standard gewährleisten, und in jene, die das Ziel durch Modifikationen des Standards erreichen, unterteilen. In der Standardisierung gibt es zwei große Felder: Zuerst die Device Access Spezifikation, in der die Kommunikation mit entfernten Knoten über die Integration von Geräten unterschiedlicher Technologien realisiert wird. Zweitens, Standardisierungen, die den Bereich der Verteilung von lokalen OSGi-Services auf entfernte OSGi-Plattformen umfassen. Der betrachtete Zeitraum liegt zwischen den Jahren 2003, der Veröffentlichung der OSGi-R3-Spezifikation, und dem Jahr 2014.

3.2.1 Forschungsarbeiten

In der Forschung wurden viele Anstrengungen unternommen das Ziel einer verteilten OSGi-Plattform zu erreichen. An dieser Stelle kann aber keine vollständige Übersicht gegeben werden, vielmehr sollen Meilensteine und innovative Projekte vorgestellt werden.

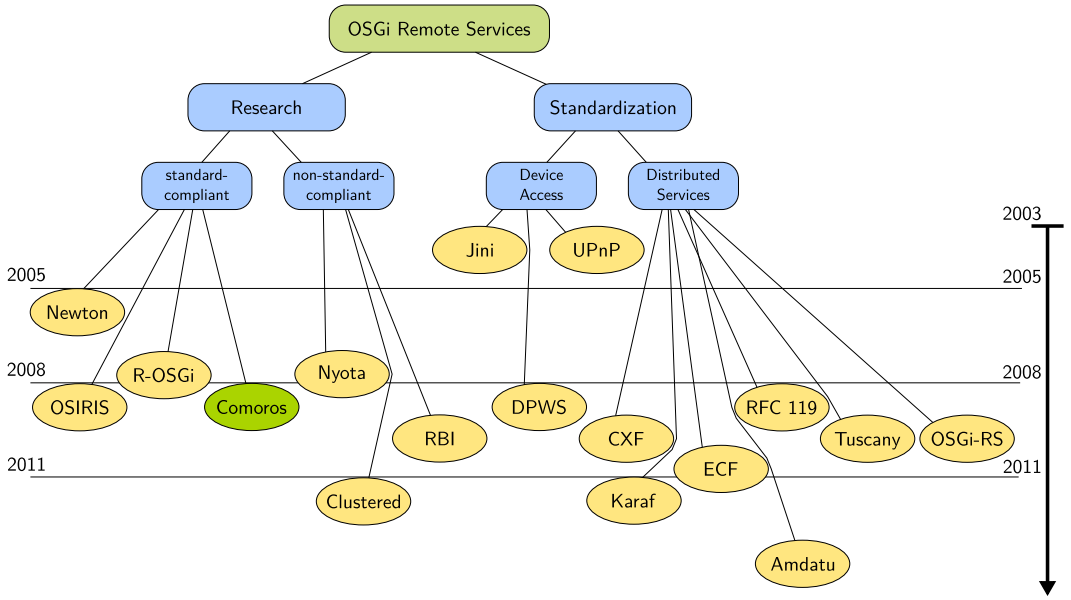


Abbildung 3.1: Übersicht über Forschungsarbeiten und Realisierungen von Standards im Bereich OSGi-Remote-Services

Newton

Bereits im Jahr 2005 veröffentlichte die Firma Paremus das Newton-Projekt[1]. Newton realisierte ein auf OSGi basierendes Komponentenframework, das explizit die Entwicklung verteilter Anwendungen auf Basis der Komponentenorientierung unterstützte. Dazu basiert Newton auf den Technologien JINI, der *Service Compendium Architecture (SCA)* und dem *SpringDM*-Framework [Wal09]. JINI dient der Kommunikation zwischen den Knoten im Netzwerk und dem Auffinden von entfernten Services. Die Verknüpfung und Komposition der Komponenten wird über deklarative Beschreibungen im Sinne der SCA und SpringDM realisiert. Wurde eine Service-Nutzung in dieser Beschreibung definiert, so formuliert der entsprechende OSGi-Client eine Suchanfrage im LDAP-Format. Ist eine lokale Suche ergebnislos, transformiert Newton die Suchanfrage in eine JINI-konforme Form und sucht in zuvor erstellten JINI-Repositories, in denen entfernte Services verzeichnet sind. Bei einer erfolgreichen Suche wird ein Proxy erstellt der vom lokalen Client verwendet werden kann.

Mittlerweile wurde das Newton-Projekt von der Firma Paremus eingestellt.

R-OSGi

Ebenso wie das Newton-Projekt gehört R-OSGi zu den standardkonformen Forschungsarbeiten, also jenen Arbeiten, die keine Modifikationen am OSGi-Framework vornehmen und keine zusätzlichen Anforderungen an die Laufzeitumgebung stellen. Entwickelt wurde R-OSGi an der Eidgenössischen Technischen Hochschule Zürich zum Ende des Jahres

1 Ed.: Paremus Ltd, URL: http://newton.codecauldron.org/, Abruf: Offline

2007 und ermöglicht die transparente Verteilung von Software-Modulen einer Applikation durch die Nutzung von Diensten über die Grenzen eines OSGi-Framework hinweg [Rel07c].

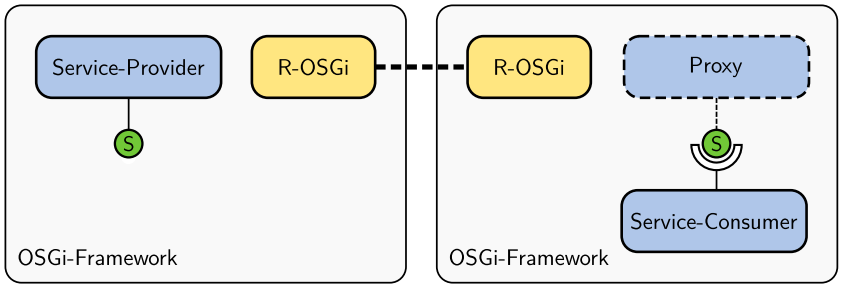


Abbildung 3.2: R-OSGi-Architektur [Rel07c]

Abbildung 3.2 zeigt die vereinfachte R-OSGi-Architektur, in der ein Service-Nutzer einen entfernten Service aufruft. Hierbei ist die Architektur so konzipiert, dass sie standardkonform ist, trotz verteilter Umgebung keine neuen Fehlermodelle einführt, die Menge der Services für eine Verteilung nicht wesentlich eingeschränkt wird und Ressourcen schonend für den Einsatz auf Kleinstgeräten ist.

Eines der Hauptziele von R-OSGi ist die transparente Nutzung von entfernten Services, d. h. es gibt für den Client keinen Unterschied in der Nutzung entfernter und lokaler Services. Dazu führt R-OSGi vier Haupttechniken ein:

1. Dynamische Proxy-Erstellung,
2. Verteilte Service-Registry,
3. Abbilden von Netzwerkfehlern auf lokale Events,
4. Typinjektion.

Die dynamische Erstellung von Proxies wird in R-OSGi durch das Erzeugen von Bytecode zur Laufzeit realisiert. Dazu kommt die ASM-Bibliothek [Bru02] zum Einsatz, mit dessen Hilfe die notwendigen Klassen für den Proxy erzeugt werden. Die Proxies realisieren zudem Punkt 2). Ohne Erweiterung des OSGi-Framework ist es nicht möglich eine verteilte Service-Registry, die wie eine lokale erscheint, zu erstellen. Services, die per R-OSGi verteilt angeboten werden sollen, werden über das Service Location Protocol (SLP) [Gut99] im Netz verfügbar gemacht. Auf dem entfernten Framework wird durch SLP der Service erkannt und ein Proxy generiert. Die erstellten Proxies nutzen, wie gewohnt, die lokale Registratur.

Für die Sicherstellung von Punkt 3) nutzt R-OSGi eine Eigenschaft des OSGi-Framework aus. Da innerhalb eines Frameworks Module jederzeit entfernt werden können, werden im Falle eines Netzwerkfehlers die betroffenen Proxies deregistriert. Dadurch ergeben sich keine neuen Fehlermodelle.

Unter der Typinjektion versteht R-OSGi ein Konzept, um die Abhängigkeiten eines Proxys aufzulösen. Eine OSGi-Servicemethode kann Typen als Ein- oder Ausgabeparameter verwenden, die außerhalb der Standard-Java-Klassen liegen. Diese Typen müssen übertragen werden, damit der Proxy aufgelöst werden kann. Sie werden in die so genannte

Injection list eingetragen, in der importierte und Bundle-eigene Typen unterschieden werden. Diese Liste wird als Service Property übertragen und der Proxy kann alle notwendigen Typen im- und exportieren.

Das Kommunikationsmodell von R-OSGi ist nachrichtenbasiert. Nachrichten werden aus Effizienzgründen binär kodiert. Die Nachrichtenkanäle in R-OSGi sind persistente TCP-Verbindungen, die so lange offen gehalten werden, wie Traffic innerhalb einer Timeout-Periode existiert. Dadurch wird der Overhead der Handshake-Phase eingespart. Nachrichten in R-OSGi können zudem durch HTTP getunnelt werden, um Kommunikation durch Firewalls zu ermöglichen.

Remote Batch Invocation

Die *Remote Batch Invocation (RBI)* [Ibr09], entwickelt von Ibrahim et. al. im Jahre 2009, ist eine Middleware für verteilte Systeme, die unter dem Namen RBI/OSGi [Kwo10] auch eine Plattform übergreifende Kommunikation in OSGi realisiert. Die grundsätzliche Idee von RBI ist den Code in Blöcke für lokale und entfernte Methodenaufrufe zu unterteilen und die Kommunikation gebündelt auszuführen. Innerhalb dieser *batch statement* genannten Blöcke werden die Methoden also nicht einzeln, sondern mittels der *remote-evaluation*-Technik [Sta90] ausgeführt. Dabei werden auch die Daten für sämtliche Methodenaufrufe gesammelt und in einem Aufruf versendet. Durch diese Technik kann der Kommunikationsaufwand deutlich verringert werden.

Die Definition der *batch statements* erfolgt in Java durch eine neu eingeführte batch-Umgebung. Diese Umgebung ist nicht Teil der offiziellen Java API und erfordert sowohl einen speziellen Compiler als auch eine spezielle Laufzeitumgebung. Demnach gehört RBI/OSGi nicht zu den standardkonformen Lösungen.

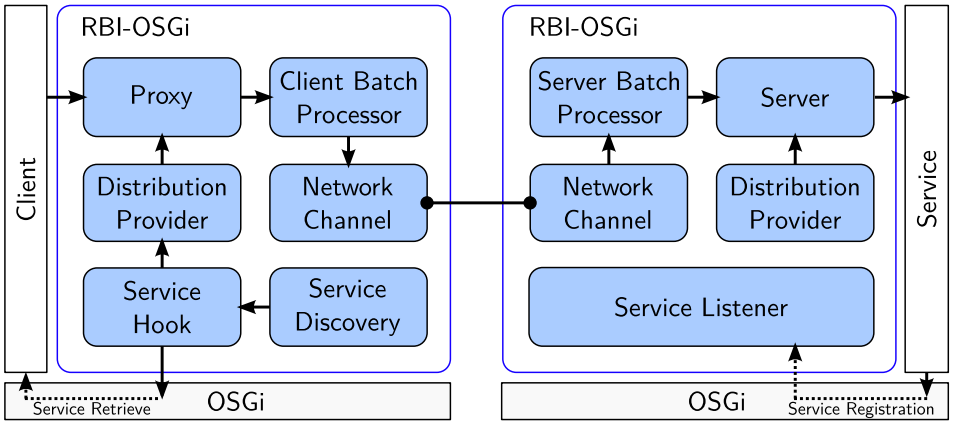


Abbildung 3.3: RBI/OSGi-Architektur [Kwo10]

Die Architektur von RBI/OSGi, illustriert in Abbildung 3.3, besteht im wesentlichen aus dem *batch processor* und dem *distribution provider*. Sobald der Service-Anbieter einen Service im OSGi-Framework registriert, instantiiert der Distribution Provider einen Server. Auf Seiten des Service-Nutzers wird der entfernte Service entdeckt und vom Distribution Provider ein Proxy erstellt. Der Client kann nun die Services des Proxys aufrufen, die der Batch Processor sammelt und zu einem Aufruf aggregiert. Der Batch

Processor auf der Service-Seite interpretiert diesen Aufruf, setzt die entsprechenden lokalen Service-Aufrufe ab und schickt das Resultat zurück zur Client-Seite.

Nyota

Nyota ist ein Projekt der *compeople AG* um eine verteilte Kommunikation über OSGi-Plattformen zu realisieren und wurde initial im Jahr 2007 veröffentlicht [07]. Ebenso wie R-OSGi verfolgt Nyota dabei den Ansatz einer transparenten Service-Nutzung für den Entwickler. Aufrufe von lokalen und entfernten Services sollen sich nicht unterscheiden, ebenso soll das OSGi-Programmierparadigma zum Registrieren und Finden von Services nicht verändert werden. Im Gegensatz zu lokalen Services verlangt Nyota bei der Registrierung lediglich das Angeben von bestimmten Service-Properties die den Endpunkt beschreiben, unter dem entfernte Plattformen den Service abrufen können.

Das tatsächliche Finden und Veröffentlichen übernimmt das Nyota Bundle. Es besteht aus drei Kernkomponenten:

- Service Publisher
- Remote Service Factory
- Remote Service Registry

Abbildung 3.4 zeigt das Zusammenspiel der Komponenten. Der *Service Publisher* erzeugt für OSGi-Services, die verteilt werden sollen, protokollspezifische Service-Endpunkte. Daraufhin erzeugt die *Remote Service Factory* auf der Client-Seite Proxy-Referenzen für die Endpunkte, die in der *Remote Service Registry* registriert werden. Auf diese kann jedes Bundle zugreifen und die Erstellung eines OSGi-Service für die Proxy-Referenz einleiten. Optional ist es möglich eine Discovery-Komponente zu laden, welche die programmierte Konfiguration des Endpunktes überflüssig macht.

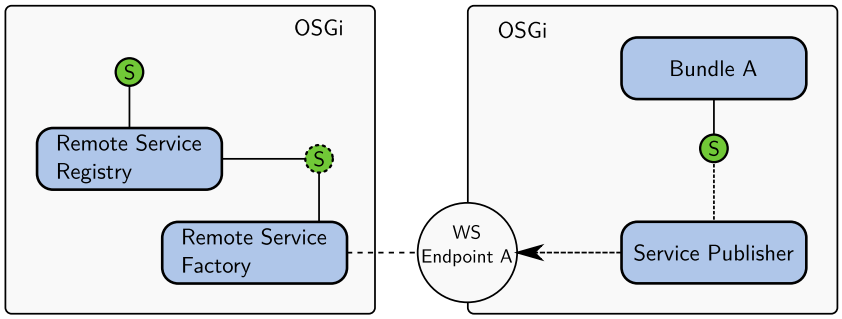


Abbildung 3.4: Nyota Architektur [07]

Nyota unterstützt die Verwendung unterschiedlicher Transportprotokolle. Aus diesem Grund existieren unterschiedliche protokollspezifische Implementierungen für den Publisher und für die Service Factory. Die Auswahl des Protokolls erfolgt auf Basis der bei der Registrierung angegebenen Property. Aktuell existieren zwei Protokoll-Implementierungen: Hessian[1] und XFire[2].

1 Ed.: Codehaus, URL: http://xfire.codehaus.org/, Abruf: 28.07.2014
2 Ed.: Caucho, URL: http://hessian.caucho.com/, Abruf: 28.07.2014

Ein OSGi Service ist keinerlei Beschränkung in der Verwendung von Typen unterworfen. Diese Typen sind aber nur innerhalb eines Bundles bekannt, es sei denn, sie werden explizit exportiert. Das Nyota-Bundle muss also alle Abhängigkeiten eines Services besitzen, um dessen Typen verwenden zu können. Dieses Problem wird mit Hilfe des Eclipse-BuddyPolicy-Konzepts [Gru05] für das Classloading gelöst. Damit ist Nyota nicht vollständig standardkonform zu OSGi und in der Interoperabilität stark eingeschränkt. Einzig die Eclipse-Equinox-Implementierung kann gegenwärtig verwendet werden.

OSIRIS

OSIRIS ist ein internationales ITEA-Projekt und erforscht die Entwicklung einer Domänen übergreifenden Open-Source-Service-Plattform. In diesem Umfeld wurde 2008 unter anderem eine Lösung für die Kommunikation verschiedener OSGi-Frameworks entwickelt [Mar08].

Im OSIRIS-Projekt wird die OSIRIS-Domäne als eine Menge von OSIRIS-Knoten definiert, die alle innerhalb einer Multicast-Gruppe, also innerhalb eines LAN liegen. Ein OSIRIS-Knoten enthält wiederum jeweils ein OSGi-Framework auf dem eine beliebige Anzahl von OSIRIS-Services zugreifbar sind.

Ein OSIRIS-Service unterscheidet sich leicht von Standard-OSGi-Services. In OSGi kann jedes beliebige Objekt als Service registriert werden. Auf dem OSIRIS-Knoten muss ein Service sowohl das `OsirisService` als auch das `java.rmi.Remote` Interface implementieren. Erst dadurch ist der *OSIRIS Domain Connector (ODC)* in der Lage, die Verteilung der Services auf andere Plattformen zu realisieren. Neben dem entfernten Aufruf von OSIRIS-Services ermöglicht der ODC das Life-Cycle-Management der Services und die Suche nach entfernten Services.

Neben dem ODC, der die Knoten nur innerhalb eines LANs verbindet, kümmert sich der *OSIRIS Internet Connector (OIC)* um den Zugang zum Internet. Über den OIC können für jeden Service Webservice-Endpunkte erstellt werden, deren Zugang nicht auf das LAN beschränkt ist.

Aus Sicht eines OSGi-Entwicklers unterscheidet sich das von OSIRIS eingeführte Programmiermodell stark von dem Standard OSGi-Programmiermodell. Neben der Restriktionen bezüglich der Services wird auf einem OSIRIS-Knoten eine eigene Service-Registry eingeführt. Services werden nicht mehr in der OSGi-Registry abgelegt, gesucht und gebunden, sondern in der OSIRIS-speziellen Implementierung. Zwar bleibt das OSIRIS-Modell standardkonform, allerdings ist der Eingriff in das Programmiermodell so stark, dass eine Nutzung für OSGi-Entwickler große Hindernisse aufstellt und die Anpassung bestehender Systeme nicht einfach ist.

Clustered OSGi

Im Jahre 2011 stellten Gelibert et. al. eine Lösung für eine verteilte OSGi-Plattform vor, die OSGi Services im Gegensatz zu allen anderen Ansätzen nicht mit Hilfe einer kommunikationsorientierten Middleware verteilten, sondern Komponenten über einen *Distributed Shared Memory (DSM)* replizierten [Gel11].

Um einen DSM in Java zu realisieren existieren verschiedene Ansätze. Die Funktionalität kann im Betriebssystem, direkt innerhalb der JVM, oberhalb der JVM in Form von

zur Laufzeit geladenen Bibliotheken oder im Compiler implementiert werden. Für den *Clustered-OSGi*-Ansatz wurde von den Entwicklern das Projekt *Terracotta* [Ter08] gewählt, das sich aus einem Server und Bibliotheken oberhalb der JVM zusammensetzt. Somit ist für Clustered OSGi eine spezielle Laufzeitumgebung notwendig und fällt in die Gruppe der nicht standardkonformen Lösungen.

DSM Lösungen werden im allgemeinen eingesetzt um den internen Zustand von Applikationen über mehrere Hosts zu verteilen. Als Beispiel dienen Applikationsserver-Cluster, deren einzelne Instanzen jeweils die selben Applikationen, Versionen und Implementierungen anbieten. Im Gegensatz dazu werden durch Clustered OSGi in den Applikationen, die aus einzelnen Modulen – Bundles und Services in OSGi – zusammengesetzt sind, nur Teile des Applikationszustands, also einzelne Module repliziert.

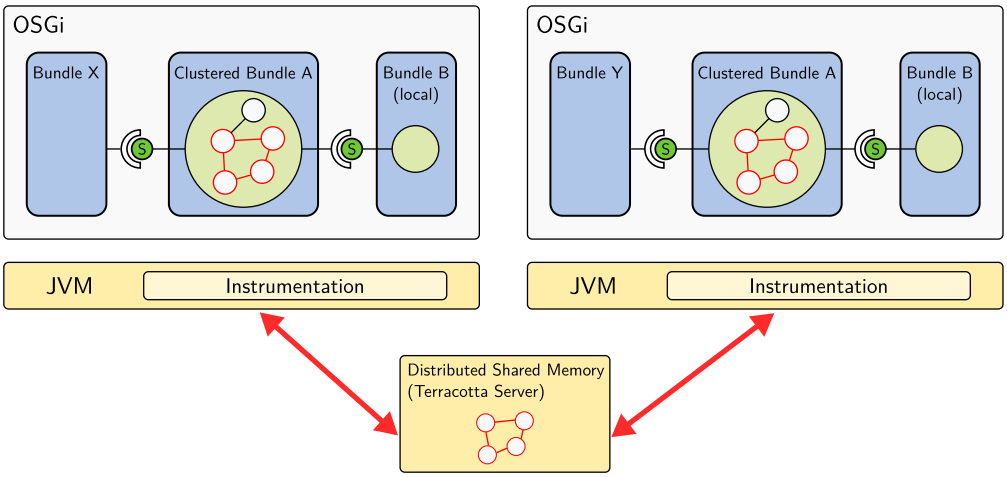


Abbildung 3.5: Aufgeteilte Komponenten über verteilte OSGi Plattformen [Gel11]

Abbildung 3.5 illustriert dieses Konzept. In Clustered OSGi werden Komponenten und deren Bindungen über das iPOJO-Framework[1] definiert. In Bundle A ist eine iPOJO-Komponente definiert, die einen Service anbietet und einen weiteren Service nutzt. Über Annotationen werden nun die Teile einer Komponente gekennzeichnet, die repliziert werden sollen, der Rest verbleibt lokal. Die Granularität reicht dabei bis auf die Ebene einzelner Variablen. Der gemeinsame Speicher für die Replikationen wird dabei in einem Terracotta-Server verwaltet. Für die lokalen Bundles B ist die Verwendung der replizierten Bundles transparent.

Durch dieses Konzept erreicht Clustered OSGi eine verteilte OSGi-Plattform ohne die Verwendung entfernter Methodenaufrufe. Neben der transparenten Nutzung entfernter Services wird über den DSM-Mechanismus auch eine Event-basierte Kommunikation realisiert.

Zu womöglich auftretende Probleme hinsichtlich der Synchronisation der unterschiedlichen Prozesse, so dass keine Zustandsänderungen verloren gehen, wird innerhalb der Veröffentlichungen keine Stellung genommen.

1 Ed.: The Apache Foundation, URL: http://ipojo.org/, Abruf: 28.07.2014

3.2.2 Standardisierung

Neben Forschungsarbeiten hat sich auch die OSGi-Allianz direkt mit der Realisierung verteilter Anwendungen befasst. In diesem Umfeld entstanden verschiedene Spezifikationen. Diese, und wichtige Referenzimplementierungen werden in diesem Abschnitt vorgestellt.

Device Access Specification

Bereits im Jahr 2003 wurde seitens der OSGi Allianz die Kommunikation über die Grenzen einer in sich abgeschlossenen OSGi-Plattform spezifiziert. Die Integration von Geräten mit den Technologien JINI und UPnP war Bestandteil der R3-Spezifikation. Mittlerweile wurde der JINI-Basedriver wieder aus der Spezifikation entfernt. Andere Technologien, wie der für diese Arbeit besonders interessante DPWS-Basedriver [Bot08] verließen nie den Status eines RFC und waren folglich zu keinem Zeitpunkt Teil der offiziellen OSGi-Spezifikation. Damit verbleibt UPnP als einzig offiziell spezifizierte Realisierung des Device-Access-Ansatzes. Eine tiefergehende Beleuchtung der Funktionsweise dieses Ansatzes findet sich bereits in Unterabschnitt 2.1.4 und muss an dieser Stelle damit nicht weiter betrachtet werden.

RFC119 - Distributed OSGi

Da die Anwender immer mehr auf eine Nutzung verteilter OSGi-Services drängten, und die Forschung erste Ansätze lieferte, nahm die OSGi-Allianz die Herausforderung an und veröffentlichte ihrerseits mit dem *RFC 119 - Distributed OSGi* [New08] einen Standardentwurf zu dieser Problemstellung. Im Jahre 2008 war dieser Entwurf Bestandteil des Early Drafts zur OSGi-R4.2-Spezifikation.

Die RFC119-Spezifikation versteht sich in diesem Zusammenhang nicht als Alternative zu bestehenden serviceorientierten Technologien, sondern stellt eine abstrakte und überwiegend generische Lösung vor, das OSGi-Framework um die Kommunikation zwischen Services unterschiedlicher Plattformen anzureichern. Dazu werden eine Menge von Anforderungen definiert:

- *Hohes Abstraktionsniveau:* In der Spezifikation werden nur rudimentäre Schnittstellen definiert, ansonsten erfolgt die Beschreibung des Verhaltens der eingeführten Komponenten abstrahiert von Implementierungsdetails, wie Kommunikationsprotokollen oder Datenformaten.
- *Konsistenz zum Programmiermodell:* Zwar werden für die Kennzeichnung entfernt aufrufbarer Services speziell definierte Properties verwendet, und die Suche nach entfernten Services basiert auch auf solchen Properties, allerdings wird das Programmiermodell nicht verändert.
- *Transparenz:* Eine vollständige Transparenz bei der Nutzung entfernter Services kann nicht erreicht werden, da durch die Verteilung die Komplexität des Gesamtsystems erhöht wird. Verteilungsbezogene Fehler und höhere Latenzen müssen in Betracht gezogen werden. Somit werden in der Spezifikation unterschiedliche Grade der Transparenz definiert, welche die unterschiedlichen Verteilungsaspekte berücksichtigen.

Um die funktionalen Anforderungen bezüglich der Verteilung von OSGi Services umzusetzen führt die Spezifikation im wesentlichen zwei Komponenten ein. Die *Distribution Software (DSW)* ermöglicht durch die Bereitstellung eines oder mehrerer Kommunikationsprotokolle den entfernten Serviceaufruf. Auf der Seite des Servers wird dazu ein *Remote Endpoint* erzeugt und auf Seite des Clients ein Proxy in der lokalen Service Registry registriert. Somit können Clients sowohl lokale als auch entfernte Services über die lokale Registry finden und abrufen. Der optionale *Discovery Service* unterstützt die DSW bei der Realisierung der Systemdynamik. Die Metadaten des lokal veröffentlichten Remote Endpoints kann der Discovery Service über beliebige Discovery-Protokolle Plattform übergreifend veröffentlichen. Auf der anderen Seite kann er die DSW über entfernt veröffentlichte Services informieren.

Die Konfiguration zur Laufzeit wird komplett über Service-Properties realisiert. Über diese Properties werden Services, beziehungsweise deren Interfaces, zur Veröffentlichung markiert und der zu erstellende Endpunkt konfiguriert. Dies beinhaltet z. B. die Wahl des Transportprotokolls, sowie die Angabe der IP-Adresse, des Ports und eines Pfades. Die Suchanfragen nach entfernten Services werden ebenfalls über Properties definiert. In Form eines LDAP-Ausdrucks wird für eine Suchanfrage ein Filter definiert, der die Suche auf lokale oder entfernte Services beschränkt. Die eigentliche Suchanfrage bleibt darüber hinaus unberührt. Über den Intent-Mechanismus können weiterhin Angebote und Anforderungen im Sinne eines *Service Level Agreement* definiert werden. So kann ein Client beispielsweise einen Remote Service anfordern, der als SOAP-basierter Webservice realisiert wurde. Aus der Menge der Services, welche die eigentliche Suchanfrage erfüllen, wird nun eine Schnittmenge bezüglich dieser Anforderung gebildet und nur die passenden Services werden als Suchergebnis zurückgegeben. Der Intent-Mechanismus ist aus der SCA-Spezifikation abgeleitet und verwendet das dort definierte Vokabular.

OSGi R4.2 - Remote Services

2009 veröffentlichte die OSGi-Allianz schließlich die finale R4.2-Spezifikation und, darin enthalten, die *Remote-Services*-Spezifikation. Im Wesentlichen basiert die Spezifikation auf dem RFC 119, jedoch wurden viele Details entfernt und dem Entwickler dadurch mehr Spielraum eingeräumt.

Im Gegensatz zum RFC 119 sind keinerlei Schnittstellen mehr fest vorgegeben. Die Spezifikation beschränkt sich auf die Einführung der DSW-Komponente und die Beschreibung des zu erwartenden Verhaltens. Der im RFC 119 eingeführte Discovery Service wurde dagegen komplett entfernt. Die Suche nach entfernten Services bleibt damit unklar und dem Entwickler überlassen. Erst einige Zeit später wurde dieser Punkt in der *Remote-Service-Admin*-Spezifikation aufgenommen und ausgiebig behandelt.

Lediglich die *Distribution Properties* sind fest spezifiziert und sichern so die Interoperabilität zwischen den einzelnen Implementierungen. Die Properties werden in unterschiedliche Kategorien eingeteilt:

- *Distribution Properties:* Diese Menge von Properties konfigurieren, wie im RFC 119, den Export und den Import von Services, abstrahiert von konkreten Technologien. Die `services.exported.interfaces`-Property gibt an, für welche Interfaces des

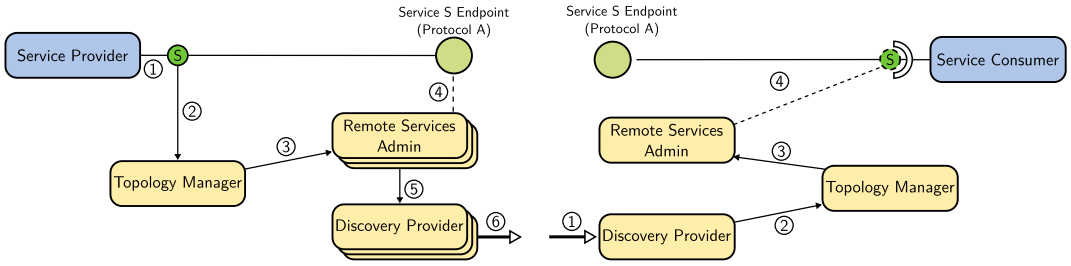
Service ein Remote Endpoint erstellt werden soll. Auf der Client-Seite werden alle Proxys mit der Property `service.imported` registriert, so dass die Suche eines Clients z. B. auf entfernte Services beschränkt werden kann.

- *Configuration Types:* Diese Properties konfigurieren den zu erstellenden Endpunkt und nehmen damit Bezug auf konkrete Technologien. Über diese Properties wird die passende, die Technologie unterstützende DSW ausgewählt, die anschließend den Endpunkt der Konfiguration folgend erzeugt.
- *Intents:* Intents beschreiben Anforderungen von Clients an Services, sowie Fähigkeiten der DSW und der Services. Stellt ein Client eine Anforderung, z. B. an die Sicherheit, so muss diese sowohl von der DSW als auch vom Service-Provider unterstützt werden. Ansonsten wird der Service nicht an den Client gebunden.

Neben der Komponentenbeschreibung und den Properties enthält die Spezifikation noch generelle Hinweise. Diese betreffen eine Beschreibung für die Suche nach entfernten Services, Limitierungen im Bereich der zu verwendenden Datentypen sowie die Behandlung von Fehlern, die durch die Verteilung entstehen (z. B. Netzwerkfehler).

Remote Service Admin Die Spezifikation des *Remote Service Admin* [The10b] ist eng mit der *Remote-Services*-Spezifikation verknüpft und wurde erst im Nachhinein in der Mitte des Jahres 2010 veröffentlicht. Sie schließt die Lücke der Verwaltung von entfernten Services. Diese Verwaltung beinhaltet das Auffinden von entfernten Services *(Discovery)* und weitgehende Richtlinien zum Binden von Services an Clients auf unterschiedlichen Plattformen.

Zu diesem Zweck werden drei Komponenten eingeführt. Der *Remote Service Admin (RSA)*, der *Topology Manager* und der *Distribution Provider*. Das Zusammenspiel der Komponenten ist in Abbildung 3.6 dargestellt.



(a) Server-Seite **(b)** Client-Seite

Abbildung 3.6: Funktionsweise des Remote Service Admin

Auf der Server Seite wird ein Service S mit der aus der *Remote-Services*-Spezifikation bekannten Property `services.exported.interfaces` registriert (1). Dieser Service wird vom Topology Manager erkannt (2). Im Topology Manager können beliebige Policies hinterlegt werden, die beispielsweise steuern, welche Services auf welchen Plattformen entfernt zugreifbar sein sollen. Auch komplexere Policies, wie das Ersetzen eines Services bei Überlast durch eine Alternative, können an dieser Stelle realisiert werden. Der Topology Manager informiert nun die registrierten Remote Service Admins (3). Unterschiedliche

RSAs unterstützen jeweils andere Technologien. Der RSA, der den registrierten Service unterstützt, erstellt einen *Remote Service Endpoint (RSE)* (4) mit der entsprechenden Technologie. Der RSE wird mit dem Service gebunden. Anschließend übermittelt der RSA dem Discovery Provider die *Endpoint Description* des RSE (5). Diese Beschreibung enthält alle relevanten Metadaten des RSE, wie z. B. die Framework ID oder die Endpoint ID, in Form einer Sammlung von Properties. Über ein nicht festgelegtes Protokoll werden diese Informationen an entfernte Discovery Provider versendet (6).

Auf der Client Seite empfängt der Discovery Provider die Informationen über den neu erstellten RSE (1), der diese an den Topology Manager weiterleitet (2). Der Topology Manager weiß, dass es von einem Client eine Suchanfrage für diesen Service gab und sendet daher die Endpoint Description an den für diesen Endpunkt zuständigen RSA (3). Der RSA erstellt den lokalen Proxy (4), der vom Service Consumer verwendet werden kann.

Apache CXF

Apache CXF ist ein Open-Source-Framework für Webservices, das als Weiterentwicklung von *Codehaus XFire* entstand. Innerhalb des Unterprojekts *Distributed OSGi* entstand im Jahre 2009 die Referenzimplementierung für den RFC 119 [The09].

Die Verteilung der OSGi-Services basiert auf Webservices, wobei die Übertragung der SOAP-Nachrichten über HTTP erfolgt. Für jeden erstellten *Remote Endpoint* wird automatisch eine WSDL erzeugt, die als Service-Beschreibung dient. Auf der Client-Seite wird aus dieser Beschreibung ein *Dynamic Client* abgeleitet, der als Proxy agiert.

Für das Marshaling der Daten verwendet Apache CXF bestehende Technologien zur XML-Datenbindung, wie beispielsweise *Apache XMLBeans* oder *JAXB* [Ort13]. Diese Technologien ermöglichen eine Abbildung von Java-Datentypen auf XML-Schema und umgekehrt. Die Integration von Legacy Services ist nicht direkt möglich, sondern bedarf der Anwendung des *Extender Models*, für das *Spring DM* in die CXF-Distribution integriert wurde.

Für das Suchen und Finden von entfernten Services, das noch Bestandteil des RFC 119 war, bietet Apache CXF einen Discovery Service an, der auf *Apache ZooKeeper Server* basiert. Einzelne Discovery Services werden vom ZooKeeper-Server synchronisiert, indem er ein virtuelles Dateisystem zur Verfügung stellt, das sämtliche Discovery-Informationen zentral verwaltet [The10a]. Falls der als optional spezifizierte Discovery Service nicht zur Verfügung steht, können die Remote Endpoints der verteilten OSGi Services auch statisch über eine XML-Datei beschrieben werden.

Eclipse Communication Framework

Das *Eclipse Communication Framework (ECF)*[1] erweitert die OSGi-Implementierung *Eclipse Equinox* um eine API zur Interprozesskommunikation Java-basierter Anwendungen. Diese API ist in die *ECF Remote Service API* und die *ECF Discovery API* aufgeteilt. Das Framework abstrahiert von konkreten Kommunikationsprotokollen und kapselt diese in so genannte *Provider*, von denen einige Implementierungen für beide

1 Ed.: The Eclipse Foundation, URL: http://www.eclipse.org/ecf/, Abruf: 28.07.2014

APIs zur Verfügung stehen. Auf Seiten der Remote Services API handelt es sich dabei z. B. um Implementierungen für die Protokolle REST, SOAP, ActiveMQ, XMPP oder R-OSGi. Für die Discovery API werden unter anderem die Protokolle Zeroconf, SLP und ZooKeeper unterstützt.

Um einen OSGi-Service über die Remote Services API verteilt anzubieten, muss dieser explizit über einen Provider als *Remote Service* registriert werden. Auf der anderen Seite muss ein Client einen Remote Service ebenso explizit bei einem Provider anfragen. Dieser Remote Service implementiert das spezielle Interface *IRemoteService*, das zwei Varianten des Serviceaufrufs anbietet, einen synchronen Aufruf und einen asynchronen Aufruf.

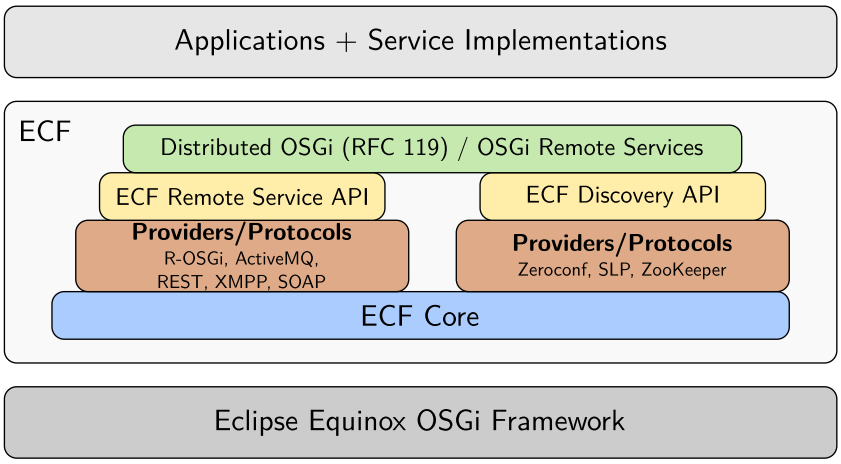


Abbildung 3.7: Distributed OSGi / Remote Services mit dem ECF

In der Version 3.0 des ECF wurde 2010 erstmalig die RFC-119-Spezifikation adaptiert. Abbildung 3.7 zeigt die gesamte Architektur. Über den Schnittstellen *ECF Remote Service API* und *ECF Discovery API* ist eine weitere Abstraktionsschicht eingeführt worden, welche die Benutzung nach dem RFC 119 ermöglicht und später auch die Kompatibilität zum *OSGi-Remote-Services*-Standard sicherstellt. So werden nun die Transparenzvorgaben aus dieser Spezifikation eingehalten und die Verteilungscharakteristika eines Service über das RFC-119-Metamodell beschrieben. Services werden für eine Verteilung nicht mehr über die einzelnen Provider, sondern in der lokalen OSGi-Service-Registry mit den entsprechenden Distribution-Properties registriert. Die Wahl des passenden Providers wird nun implizit über das Setzen der Property `osgi.remote.configuration.type` gesteuert. Auf der Client Seite wird die Referenz auf einen entfernten Service ebenso nicht mehr über den Provider, sondern über die Service-Registry bezogen. Daraufhin erhält der Client eine ServiceReference als Verweis auf einen Proxy, der einen synchronen Aufruf des entfernten Service ermöglicht.

Der Zugriff über die *IRemoteService*-Schnittstelle ist somit nicht mehr notwendig. Dennoch besteht die Möglichkeit weiterhin, um einen asynchronen Service-Aufruf zu ermöglichen. Der RFC 119 verlangt, dass jeder Proxy mit der Property `osgi.remote` registriert wird. Der Wert ist dabei nicht festgelegt. Das ECF hinterlegt als Wert eine *IRemoteService*-Instanz. So kann jeder Client diese Instanz einfach abrufen und für asynchrone Aufrufe nutzen.

Die im RFC 119 als optional deklarierte Discovery-Komponente wird im ECF durch die ECF Discovery API abgedeckt. Anfragen an den Discovery-Service werden an diese API weitergeleitet, die vorgefertigte Provider mit unterschiedlichen Protokollen anspricht. Auf diese Weise kann eine Suche nach entfernten Services im Sinne der RFC-119-Spezifikation durchgeführt werden.

Weitere Implementierungen: Tuscany, Karaf, Amdatu

Im Laufe der Zeit entwickelten sich immer mehr Implementierungen für den *OSGi-Remote-Services*-Standard. Diese Implementierungen sind meist nur Teil eines großen Frameworks. Im Tuscany-Projekt[1], erstmals 2009 veröffentlicht, wird die Distribution Software über einen SCA-Container realisiert, auf deren Technologie schon das Newton-Projekt aufsetzte. 2011 wurde eine weitere Remote-Services-Implementierung auch Teil des Karaf-Projekts[2]. Die aktuellste Enwicklung wurde in das Amdatu-Projekt[3] integriert. Dort wurden gleich mehrere Protokolle, Serialisierungen und Discovery-Mechanismen umgesetzt. Es existieren Varianten mit HTTP+JSON, HTTP+JavaSerialization und für das Discovery werden das SLP, Multicast DNS [Che11b] und Hazelcast[4] eingesezt.

3.3 Zwischenfazit

Die Entwicklung modularer und flexible strukturierter verteilter Anwendungen hat, wie beschrieben, seit 2008 eine schnelle Entwicklung vollzogen. Im Bereich Java-basierter Anwendungen hat sich als Komponenten-Framework die OSGi-Plattform etabliert. Sie bietet den Entwicklern ein umfangreiches Life-Cycle-Management, ein ausgereiftes Komponentenmodell und eine umfangreiche Sammlung von Basisdiensten.

Für die Entwicklung verteilter OSGi-Anwendungen muss eine Plattform übergreifende Kommunikation realisiert werden, die es ermöglicht einzelne Komponenten einer Applikation auf verschiedene Knoten zu verteilen. Dazu müssen lokale OSGi-Clients die Fähigkeit besitzen auf entfernte OSGi-Services zuzugreifen. Neben dem als Request/Response bekannten Kommunikationsmuster muss für eine vollständige Kommunikation zusätzlich auch die Event-basierte Kommunikation Unterstützung finden. Events, die innerhalb einer lokalen OSGi-Plattform veröffentlicht werden, müssen auch in entfernten Plattformen empfangen werden können. Damit die verteilten OSGi-Anwendungen auch vollständig von den Eigenschaften der OSGi-Plattform profitieren, und sich somit wie eine lokale Applikation verhalten können, müssen die besonderen Merkmale der Plattform, wie z. B. das Life-Cycle-Management auch in einer verteilten Umgebung unterstützt werden.

Das beschriebene Szenario umfasst bisher nur die reine Verteilung einer OSGi-Umgebung. Gerade im Bereich der Anwendungsentwicklung in heterogenen und gewachsenen Strukturen, wie man sie häufig in der Gebäudeautomatisierung und im AAL-Bereich vorfindet, reichen die so erlangten Möglichkeiten nicht aus. Strukturen in diesen Domänen umfassen häufig eine Vielzahl von Geräten und Diensten unterschiedlichster Technologien, so dass

1 Ed.: Apache Foundation, URL: http://tuscany.apache.org/, Abruf: 28.07.2014
2 Ed.: Apache Foundation, URL: http://karaf.apache.org/, Abruf: 28.07.2014
3 Ed.: Amdatu Foundation, URL: http://amdatu.org/, Abruf: 28.07.2014
4 Ed.: Hazelcast Inc., URL: http://hazelcast.com/, Abruf: 28.07.2014

eine Integration dieser Komponenten in die verteilte OSGi-Umgebung sinnvoll erscheint. Dieser Anwendungsfall umfasst die Integration beliebiger Dienste in die OSGi-Umgebung, OSGi-Clients müssen also in der Lage sein externe Dienste jeder Technologie auf transparente Weise zu nutzen. Geräte und Sensoren mit seriellen Schnittstellen wie USB oder RS232 müssen demnach so in die Plattform integriert werden, so dass sie über Service-Schnittstellen in ebenso transparenter Weise verwendet werden können.

Die transparente Nutzung ist ein entscheidendes Kriterium für eine komfortable Erstellung verteilter OSGi-Anwendungen. Die OSGi-Plattform gibt ein Programmiermodell vor, das auch für verteilte Anwendungen nicht verändert werden sollte. Nur so kann eine große Akzeptanz, eine Integration bestehender Komponenten und somit eine komfortable Nutzung gewährleistet werden. In die gleiche Kerbe schlägt die Verwendung und die Einhaltung offener Standards. Auf diese Weise wird einerseits die Akzeptanz erhöht und andererseits die Integration von Altkomponenten vereinfacht.

Da die OSGi-Plattform auch im mobilen Sektor, beispielsweise bei eHealth-Anwendungen, und auf Kleinstgeräten, beispielsweise in der Gebäudeautomatisierung, eingesetzt wird, muss die Verteilung der Dienste in effizienter Weise realisiert werden. Im Detail muss die Kommunikation in effizienter Weise ablaufen und die für die Verteilung verantwortliche Middleware muss entsprechend leichtgewichtig sein.

3.3.1 Problemlösungen

Die aufgestellten Anforderungen werden von den hier vorgestellten Projekten nur teilweise umgesetzt:

1. *Plattform übergreifende Kommunikation:* Die Kommunikation zwischen unterschiedlichen OSGi-Plattformen wird von allen beschriebenen Ansätzen realisiert. Dabei unterscheiden sich die Ansätze hinsichtlich der Architektur, der verwendeten Protokolle und der Beschreibung der Komponenten. Die verbreitetste Architektur basiert auf dem Ansatz des *Remote Procedure Call*. Auf der Server-Seite wird ein Skeleton erzeugt und anschließend auf der Client-Seite ein entsprechender Proxy. Die entfernten Methodenaufrufe werden nun mit Hilfe des implementierten Protokolls ausgeführt. Dieser Ansatz wird von R-OSGi, RBI, Nyota, OSIRIS, ApacheCXF und dem ECF umgesetzt. Bei den verwendeten Protokollen herrscht eine große Vielfalt. R-OSGi verwendet als einzige Lösung ein Protokoll der Transportschicht, nämlich TCP. Alle anderen Ansätze verwenden allesamt Protokolle der Anwendungsschicht: JINI (Newton), Hessian, XFire (Nyota), JavaRMI (OSIRIS), Webservices (ApacheCXF), REST, ActiveMQ und XMPP (ECF).

   Der ClusteredOSGi-Ansatz setzt als Einziger auf ein anderes Architekturprinzip und realisiert die Plattform übergreifende Kommunikation über einen *Distributed-Shared-Memory*-Ansatz. Demnach ist bei dieser Lösung kein Transportprotokoll definiert.

   Sowohl der offizielle *OSGi-Remote-Services*-Standard als auch dessen Vorgänger *Distributed OSGi* geben keine konkrete Architektur vor. Vielmehr werden nur Komponenten und deren erwartetes Verhalten definiert. Konkrete Technologien, wie z. B. ein Kommunikationsprotokoll, sind ebenso nicht vorgeschrieben.

2. *Integration von OSGi-fremden Services und Clients:* Die Integration OSGi-fremder Services und Clients in das verteilte OSGi-Umfeld wird von den meisten Ansätzen nicht behandelt. Newton, R-OSGi, RBI, OSIRIS und ClusteredOSGi ermöglichen keinen Zugriff auf Services, die nicht innerhalb einer OSGi-Plattform laufen, und ermöglichen Clients außerhalb einer OSGi-Plattform auch keinen Zugriff auf OSGi-Services.

   Anders sieht es bei den Implementierungen der *OSGi-Device-Access*-Spezifikation aus. Hier werden Geräte in die OSGi-Plattform integriert. Handelt es sich bei diesen Geräten um solche, die eine Geräte-basierte SOA umsetzen, wie UPnP oder DPWS, können die Services dieser Geräte in OSGi verwendet werden. Die Integration von OSGi-fremden Clients wird aber auch hier nicht realisiert.

   Diese Integration wird innerhalb der Projekte Nyota und dem ECF realisiert. Allerdings erscheint diese Realisierung eher zufällig und findet in den entsprechenden Veröffentlichungen auch keine Erwähnung. Nyota verwendet als Kommunikationsprotokoll Webservices. Für jeden OSGi-Service, der entfernt zugreifbar sein soll, wird ein Webservice-Endpoint erstellt. Dieser Endpoint ist dann natürlich auch von jedem beliebigen Webservice-Client nutzbar. Genauso können die REST-Endpunkte aus dem ECF von beliebigen REST-Clients verwendet werden. In der Theorie können nun natürlich auch Webservice- oder REST-Endpunkte außerhalb der OSGi-Plattform erstellt werden, die sich an die Form der durch die entsprechende Software automatisch erstellten Endpunkte halten. Diese könnten ohne weiteres in die OSGi-Plattform integriert werden und wären dann durch lokale Clients nutzbar. Da dieses Vorgehen aber in keinem der Projekte spezifiziert ist, ist diese Möglichkeit in der Praxis nicht zu verwenden.

   Diese Lücke schließt das ApacheCXF-Projekt. Für jeden verteilt angebotenen Service wird ein Remote Endpoint mit einer WSDL-Beschreibung erzeugt. Dadurch können OSGi-Services von beliebigen Webservice-basierten Clients genutzt werden. Ebenso kann aus einer WSDL- Beschreibung automatisch ein Dynamic Client erzeugt werden. Beliebige Webservices mit einer WSDL-Beschreibung werden so innerhalb der OSGi-Umgebung nutzbar.

   Von Seiten der OSGi-Allianz wurde dieser Anwendungsfall innerhalb des RFC 119 behandelt und spezifiziert. In dem offiziellen Release der *OSGi Remote Services* wurden diese Teile aber wieder entfernt, da sich innerhalb des Konsortiums kein Konsens finden ließ.

3. *Integration von Geräten:* Die Integration von Geräten in die OSGi-Plattform wird von der *OSGi-Device-Access*-Spezifikation behandelt. Offiziell wird nur noch UPnP unterstützt. Es gibt aber Basedriver-Implementierungen für eine Reihe weiterer Technologien (siehe dazu Unterabschnitt 2.1.4). Alle anderen Ansätze behandeln die Integration von Geräten nicht.

4. *Umsetzung einer Event-basierten Kommunikation:* Die Event-basierte Kommunikation findet lediglich in einem Projekt explizite Erwähnung. Innerhalb des ClusteredOSGi-Projekts werden Komponenten ganz oder teilweise repliziert und

greifen fortan auf einen gemeinsamen Speicher zu. Dadurch können auch alle, für die Event-basierte Kommunikation notwendigen Komponenten verteilt verwendet werden. Alle anderen Projekte unterstützen Events nicht explizit, können diese Kommunikation dennoch teilweise anbieten. Da der *OSGi Event Admin* dem Whiteboard-Pattern [The04] folgt, werden `EventHandler` als Service in der Registry registriert. Werden nun Events in der Plattform veröffentlicht, ruft der Event Admin diesen Service auf. Durch eine einfache Verteilung des EventHandler-Service können dann auch Events entfernter Plattformen empfangen werden.

5. *Realisierung einer effizienten Verteilung:* Die Realisierung eines effizienten verteilten OSGi-Umfelds wird im wesentlichen durch zwei Punkte erreicht. Zum einen muss eine effiziente Kommunikation gewährleistet werden. Zum anderen muss die Software, welche die Verteilung implementiert, leichtgewichtig sein, so dass ein Einsatz auf Kleinstgeräten möglich ist. Die meisten Projekte legen keinen Augenmerk auf eine effiziente Umsetzung oder sind sogar im Enterprise-Umfeld angesiedelt und entsprechend schwergewichtig. In der Protokollfrage setzt R-OSGi mit dem einfachen TCP und binär kodierten Nachrichten auf ein effizientes Protokoll mit wenig Overhead. Das ECF bietet verschiedene Varianten in der Kommunikation an, unter anderem auch einen R-OSGi-Provider, so dass die Kommunikation über R-OSGi-Mechanismen realisiert wird. Somit kann auch vom ECF eine effiziente Kommunikation angeboten werden. RBI hingegen führt die *batch invocation* ein. Diese erhöht die Effizienz durch das Sammeln von Nachrichten und das anschließend gebündelte Versenden.

   Der Footprint, als zweite entscheidende Größe, schwankt bei den verschiedenen Projekten sehr stark. Während R-OSGi auf dem Concierge-Framework [Rel07a], einem OSGi-Framework für eingebettete Systeme, läuft, umfasst die ApacheCXF-Distribution 30 Bundles mit einer Codegröße von 8 Megabyte, so dass ein Einsatz auf Ressourcen beschränkten Plattformen unmöglich ist. Weiterhin ist R-OSGi das einzige Projekt, das auf dem Java-CDC-Profil basiert, und damit die notwendige Voraussetzung für den Einsatz auf eingebetteten Systemen erfüllt.

6. *Unterstützung eines verteilten Life-Cycle-Managements:* Ein verteiltes Life-Cycle-Management kann nur durch eine verteilte Service Registry erreicht werden. Wird ein Bundle lokal gestoppt oder aktualisiert, muss diese Information auch an die entfernten Plattformen weitergeleitet werden. Durch das Proxy/Skeleton-Prinzip, das die meisten Projekte umsetzen, ist dieses Ziel einfach erreicht. Wird ein lokales Bundle gestoppt, werden alle Proxies auf entfernten Plattformen ebenfalls gestoppt. Ein Client erkennt so keinen Unterschied, ob es sich um eine lokale oder um eine entfernte Komponente handelt und kann wie gewohnt auf das Ereignis reagieren. Eine weitere Lösung für eine verteilte Registry wird durch das ClusteredOSGi-Projekt umgesetzt. Durch die Verwendung eines gemeinsamen verteilten Speichers wird auch die Registry auf den einzelnen Knoten repliziert und ist von allen Knoten aus transparent zu nutzen. Das OSIRIS-Projekt hingegen setzt ganz auf die Einführung einer eigenen verteilten Registry, welche die lokale OSGi-Service-Registry ersetzt.

Für eine verteilte Registry muss auch das dynamische Suchen und Finden von entfernten Services umgesetzt werden. Dazu führen die meisten Projekte separate Discovery-Komponenten ein. Im groben fangen diese Komponenten die lokalen Suchanfragen ab und leiten diese über ein Discovery-Protokoll an die entfernten Plattformen weiter. Diese Protokolle sind z. B. JINI (Newton), SLP (R-OSGi, ECF) oder Zeroconf (ECF). Ferner setzt Apache CXF mit dem ZooKeeper auf einen zentralen Discovery-Server über den einzelne Discovery-Services synchronisiert werden.

Eine Discovery-Komponente wurde bereits im RFC 119 spezifiziert, fand aber keine Berücksichtigung in der finalen Remote-Services-Spezifikation. Diese Lücke wurde erst mit der Remote-Services-Admin-Spezifikation geschlossen.

3.3.2 Lösungsbewertung

Betrachtet man die einzelnen Problemstellungen, fällt auf, dass diese durch einzelne Projekte gut umgesetzt werden, der Blick auf eine ganzheitliche Lösung aber fehlt. Die Projekte bieten also zumeist eine Speziallösung für einzelne Problemstellungen an – z. B. RBI für eine effiziente Kommunikation zwischen OSGi Services – betrachten aber kein schlüssiges Einsatzgebiet und dessen gesamten Anforderungen. Bei den einzelnen Projekten ergeben sich somit folgende Schwachstellen:

- *Keine Einbindung von Geräten und Diensten außerhalb von OSGi:* Einzig das ApacheCXF-Projekt und die *Device-Access*-Spezifikation bieten Lösungen in diesem Bereich, wobei CXF die Kommunikation mit fremden Services und die *Device-Access*-Spezifikation die Kommunikation mit Geräten unterstützt. Kein Projekt bietet die Unterstützung beider Anwendungsfälle.
- *Beschränkung auf Nachrichten-basierte Kommunikation:* In allen Projekten, mit Ausnahme des ClusteredOSGi-Projekts, ist die Kommunikation nachrichtenbasiert. Für die Übertragung großer und kontinuierlicher Datenmengen ist diese Art der Übertragung aber ineffizient. Eine Unterstützung von Datenströmen wäre hilfreich, wird aber von den einzelnen Lösungen nicht angeboten.
- *Nicht standardkonform:* Ein wichtiger Punkt zur Erhöhung der Akzeptanz einer Software und zur Erhöhung der Interoperabilität und damit der Vermeidung von Insellösungen, ist die Standardkonformität. Ein großer Nachteil des ECF ist z. B. seine Kopplung an das Eclipse-Equinox-Framework. Laut Aussage des ECF-Projektleiters Scott Lewis, wurde die Möglichkeit einer Portierung auf andere OSGi-Plattformen noch nicht hinreichend eruiert [Lew10]. Andere Projekte, wie das ClusteredOSGi-Projekt setzen eine spezielle Laufzeitumgebung vorraus. Beides schränkt die Verwendung in erheblichen Maße ein und steht im Gegensatz zu der OSGi-Remote-Services-Spezifikation.
- *Einführung eines neuen Programmiermodells:* Für OSGi-Entwickler, die bisher lokale OSGi-Anwendungen entwickelt haben, ist es wichtig, dass sich das Programmiermodell nicht ändert. Oberstes Ziel sollte also die Homogenität auf der

Anwendungsschicht sein, so dass spezielle Verteilungsaspekte dem Entwickler verborgen bleiben können. Kann der Entwickler einer verteilten Anwendung diese wie eine lokale Anwendung entwickeln, so ist die Implementierung einer Anwendung weniger arbeitsintensiv, da keine Einarbeitung notwendig ist, und folglich ist auch die Akzeptanz höher.

In diesem Punkt fallen vor allem die Projekte OSIRIS und RBI auf. OSIRIS zwingt den Entwickler verteilte Services mit vorgegebenen Interfaces zu implementieren, eine Vorschrift, die für lokale Anwendungen nicht gilt. Weiterhin werden Services nicht in der normalen Service Registry angemeldet, sondern in einer neu eingeführten Registry für verteilte Services. Durch diese beiden Punkte ändert sich die Implementierung einer verteilten Anwendung stark. Eine deklarative Beschreibung der Services und deren Bindungen, wie in der *OSGi-Declarative-Services*-Spezifikation eingeführt, ist so nicht mehr möglich. Ein noch stärkerer Eingriff wird von RBI umgesetzt. Die *batch statements*, die für eine gebündelte Kommunikation sorgen, müssen über neu eingeführte Java-Sprachkonstrukte definiert werden. Dies erfordert eine hohe Einarbeitung, vor allem ein Umdenken bezüglich der bisher gängigen Kommunikationsmuster. Somit unterscheidet sich die Entwicklung einer RBI-Anwendung stark von der Entwicklung einer lokalen OSGi-Anwendung und verlangt vom Software-Entwickler eine intensive Einarbeitung.

- *Keine Flexibilität:* Gerade im Bereich ambienter Systeme ist eine adäquate Reaktion auf wechselnde Umwelteinflüsse und Anforderungen unumgänglich. Einige der vorgestellten Projekte implementieren verschiedene Protokollalternativen, aus denen der Entwickler die passende für den jeweiligen Anwendungsfall auswählen kann. Besonders erwähnenswert ist an dieser Stelle das ECF, das mit ActiveMQ, REST, XMPP, SOAP und R-OSGi gleich fünf Protokolle anbietet und beliebig um weitere Protokolle erweitert werden kann. Auch andere Projekte, wie Nyota bieten Protokollalternativen (Hessian, XFire). Der Rest beschränkt sich aber auf die Umsetzung eines Protokolls. Auch eigenständige Erweiterungen sind nicht vorgesehen.

  Um eine hohe Flexibilität zu erreichen, müssen diese Protokolle, wie auch weitere Anwendungsparameter zur Laufzeit verändert werden können. Hierzu müssen Konfigurationspunkte definiert und über spezifizierte Schnittstellen zugreifbar sein. Keine der vorgestellten Projekte behandeln aber diesen Punkt.

- *Keine Unterstützung von Altkomponenten:* Keine der hier vorgestellten Lösungen bietet eine nahtlose Integration von Altkomponenten in ein verteiltes OSGi-Umfeld. Gerade in der komponentenbasierte Entwicklung ist die Wiederverwendung von bestehenden Komponenten aber ein Dogma. Zwar sind bei den Lösungen, die das bestehende OSGi-Programmiermodell beibehalten, die Änderungen im Programmcode sehr gering und beschränken sich häufig auf das Setzen zusätzlicher Properties, jedoch liegen viele Altkomponenten nur im Binärformat vor, so dass eine Änderung gar nicht möglich ist. Bei dieser Problematik verweisen Apache CXF und auch das ECF auf das Extender Model und bieten somit keine eigenen Mittel zur Verteilung von Legacy Services. Für mobile Anwendungsumgebungen mit hoher Dynamik ist

das Extender Model jedoch zu unflexibel. Die anderen Projekte behandeln diese Thematik nicht.

- *Keine ausreichende Unterstützung einer Event-basierten Kommunikation:* Wie erwähnt, wird von den meisten Projekten eine indirekte Event-Unterstützung über den *OSGi Event Admin Service* angeboten. Dabei wird der *EventHandler*-Service auf andere Plattformen verteilt. Keine Unterstützung bieten die Projekte aber für das Publish/Subscribe-Pattern. Dieser Event-Mechanismus verlangt die Registrierung der Handler direkt am Framework, eine Verteilung ist so nicht möglich. Auch wenn das Publish/Subscribe-Pattern keine Best-Practise-Lösung ist, ist dieser Punkt insofern besonders erwähnenswert, da alle OSGi-System-Events über dieses Pattern versendet werden. Eine Verteilung dieser Events ist somit ausgeschlossen. Eine Event-basierte Kommunikation mit OSGi-fremden Services oder Clients wird von keinem Projekt realisiert.
- *Keine ausreichende Effizienz:* Für den Einsatz auf Kleinstgeräten ist der Footprint eine wichtige Größe. Für die meisten Projekte ist dieser aber kein entscheidendes Kriterium. Stattdessen werden schwergewichtige Bibliotheken oder Frameworks zur Lösung unterschiedlicher Anwendungsfälle eingesetzt. Die Projekte Newton, ApacheCXF und ClusteredOSGi nutzen z. B. zur Beschreibung der Komponenten und deren Assoziationen mächtige Frameworks wie SpringDM oder iPojo. Auch ein großer Vorteil von Apache CXF, die flexible XML-Datenbindung, geht zu Lasten des Footprints. Technologien wie XMLBeans oder JAXB ermöglichen es komplexe Java-Typen zu (de-)serialisieren und damit die Data-Fencing-Restriktionen aufzuheben. Diese Bibliotheken haben aber selbst einen hohen Footprint und folgen zusätzlich nicht dem Java-CDC-Profil und verhindern somit den Einsatz auf Kleinstgeräten.

  Die zweite wichtige Anforderung, eine effiziente Kommunikation, findet bei einigen Projekten, wie RBI, besondere Beachtung. Die R-OSGi-Lösung, binär kodierte TCP-Nachrichten und die Verwendung des Java Serialization-Mechanismus zur Serialisierung der Nutzdaten, bietet zwar eine effiziente Kommunikation, hat aber im Gegenzug einen großen Nachteil. Durch diese Art der Kommunikation wird eine Plattformabhängigkeit geschaffen und eine weitergehende Integration von OSGi-fremden Services ausgeschlossen.
- *Keine Sicherheitsaspekte:* Keine der Lösungen nimmt Stellung zur Datensicherheit. Es werden keine besonderen Möglichkeiten angeboten die Kommunikation gegen Übergriffe abzusichern.

Es ist deutlich zu sehen, dass die Projekte keine ausreichende Lösung für eine Entwicklung verteilter Anwendungen im Bereich ambienter Systeme liefern. Zwar werden viele Teilaspekte behandelt und gut gelöst, doch gibt es kein Projekt mit einem ganzheitlichen Ansatz für diesen Anwendungsfall.

Es fällt weiterhin auf, dass dem Software-Entwickler und seinem Arbeitsaufwand keine entscheidende Bedeutung beigemessen wird. Die Einführung neuer Programmiermodelle behindert die Arbeit der Entwickler und führt so zu einer geringen Akzeptanz. Stattdessen sollte der Entwickler durch die Realisierung eines hohen Programmierkomforts in seiner

Arbeit unterstützt werden. Darunter ist zu verstehen, dass es möglichst einfach und komfortabel möglich sein sollte, eine verteilte Anwendung zu erschaffen und anschließend zur Laufzeit an wechselnde Anforderungen anzupassen. Der Komfort wird dabei nicht nur durch den Programmcode definiert, sondern z. B. auch durch den Einsatz von Entwicklungstools. Hier bietet aber einzig das R-OSGi-Projekt ein Werkzeug für ein einfaches Deployment an [Rel07b]. In allen weiteren Projekten ist keine Werkzeugunterstützung vorgesehen.

4 Anforderungen

Dieses Kapitel dient der Anforderungsanalyse. Dabei werden im groben folgende Fragen beantwortet:

- Wie ist die Systemlandschaft von Comoros definiert?
- Welche funktionalen Anforderungen existieren für Comoros?
- Welche nicht-funktionalen Anforderungen existieren für Comoros?
- Welche Anforderungen werden an die Softwareentwicklung gestellt?

Die in Kapitel 3 vorgestellten Projekte haben in der anschließenden Analyse große Schwächen hinsichtlich einer umfassenden Lösung für eine OSGi-basierte Dienste- und Geräte-Infrastruktur gezeigt. Diese Lücke wird von Comoros geschlossen. Um die Anforderungen an die Software exakt definieren zu können, wird zuerst die resultierende Systemlandschaft mit den gewünschten Eigenschaften präsentiert. An dieser Stelle können dann Anwendungsfälle identifiziert werden, aus denen die Anforderungen entstehen.

Die Systemlandschaft ist in Abbildung 4.1 illustriert. Sie besteht aus einer Menge von OSGi-Service-Plattformen, Services und Clients unterschiedlicher Technologien und Geräten, die sich alle in unterschiedlichen lokalen Netzwerken befinden und untereinander kommunizieren. Die Aufgabe von Comoros ist der Aufbau einer verteilten Dienste- und Geräte-Infrastruktur und die Unterstützung in der Entwicklung verteilter Anwendungen innerhalb dieser Infrastruktur. Die Hauptkomponente ist die OSGi-Service-Plattform, eine Komponentenplattform für Java, die alle Vorteile komponentenorientierter Softwareentwicklung realisiert. Diese Plattform wird um die Fähigkeit erweitert mit Diensten aus entfernten Komponenten zu interagieren. Das beinhaltet sowohl den Operationsaufruf von entfernten OSGi-Services als auch den Aufruf von OSGi-fremden Services durch lokale OSGi-Clients. Umgekehrt sollen auch OSGi-fremde Clients OSGi-Services nutzen können.

Um eine möglichst hohe Kompatibilität der Plattform übergreifenden Kommunikation zu gewährleisten, muss zusätzlich eine Event-basierte Kommunikation ermöglicht werden. Sowohl der Operationsaufruf als auch die Event-basierte Kommunikation wird dabei über unterschiedliche Kommunikationsprotokolle realisiert. Liegt das Hauptaugenmerk auf dem DPWS-Protokoll, werden aber auch effizientere Protokolle, wie das CoAP-Protokoll, direkt in Comoros umgesetzt. Möglichkeiten zur Erweiterung um zusätzliche Protokolle müssen dabei geschaffen werden. Durch die Protokollvielfalt ergeben sich zusätzlich Herausforderungen in der Datenrepräsentation, die beachtet werden müssen. Auch die Datensicherheit muss sichergestellt werden.

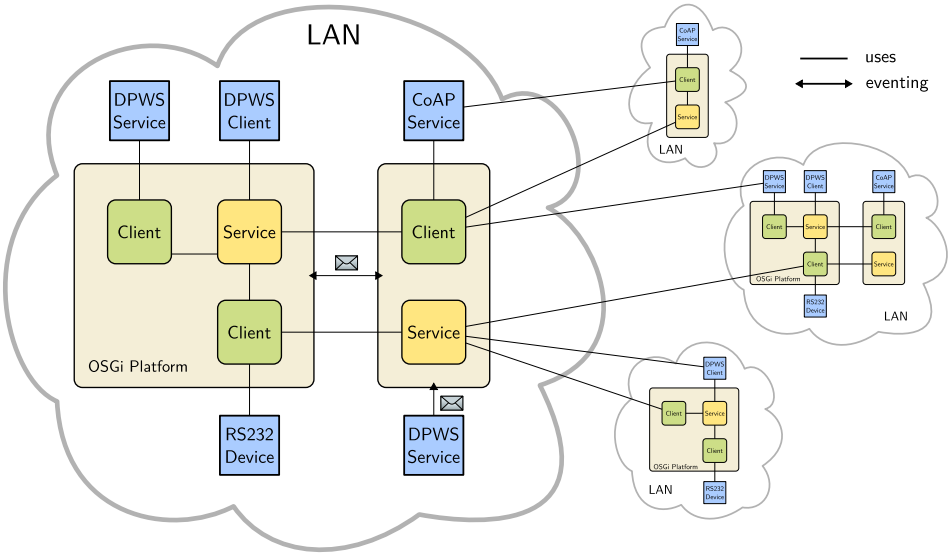


Abbildung 4.1: Überblick über das durch Comoros aufgespannte verteilte System

Die OSGi-Plattform bietet als wesentliche Eigenschaft ein umfangreiches Life-Cycle-Management, das auch in einer verteilten Umgebung beibehalten werden muss. Auch DPWS-Geräte unterliegen einem Life-Cycle, so dass insgesamt eine hohe Dynamik entstehen kann, die von Comoros behandelt werden muss.

Alle Anpassungen und Erweiterungen, die von Comoros durch den Aufbau einer verteilten OSGi-basierten Dienste- und Geräte-Infrastruktur entstehen, dürfen zudem bestehende Standards und Programmiermodelle nicht verändern. Auf diese Weise wird auch die Weiterverwendung von Altkomponenten nicht verhindert.

Da das entstehende System sehr komplex, dynamisch und zudem wechselnden Anforderungen unterworfen ist, muss Comoros adaptierbar sein. Es müssen sowohl Schnittstellen zur Überwachung des Systems angeboten werden, als auch Schnittstellen zur Konfiguration und zur Parametrisierung zur Laufzeit.

Mit Blick auf die Softwareentwicklung muss Comoros einen hohen Standard realisieren, um einerseits die gegebenen Aufgaben effizient umzusetzen, und um andererseits dem Entwickler von verteilten Anwendungen einen hohen Komfort anzubieten. Dieser Punkt beinhaltet auch die Bereitstellung geeigneter Entwicklungswerkzeuge.

Tabelle 4.1: Überblick über die Anforderungen an die Comoros Middleware

Funktionale Anforderungen	*Nicht-funktionale Anforderungen*
OSGi-Client nutzt entfernten OSGi-Service	Unterstützung und Maskierung heterogener Strukturen
OSGi-fremder Service nutzt entfernten OSGi-Service	Umsetzung einer Verteilungstransparenz
OSGi-Client nutzt entferten OSGi-fremden Client	Überwachung und Konfiguration der Middleware
Plattform übergreifende Event-basierte Kommunikation	Effiziente Umsetzung
Plattform übergreifende Life-Cycle-Unterstützung	Hoher Entwicklungs- und Anwendungs-Komfort
OSGi-Client nutzt Gerätefunktionen	Gute Wartbarkeit und Erweiterbarkeit
Plattformübergreifende Event-basierte Kommunikation	Kompatibilität und Interoperabilität
	Datensicherheit

Tabelle 4.1 listet alle aus den Überlegungen entstandenen Anforderungen auf. Im Folgenden wird die Vielfalt der Systemlandschaft genau analysiert und die Anforderungen detailliert spezifiziert, so dass der Funktionsumfang und die Eigenschaften der Comoros-Middleware deutlich werden.

4.1 Aufbau einer verteilten Service- und Geräte-Infrastruktur

Wie im vorherigen Abschnitt erläutert, strebt Comoros nicht nur die Kommunikation zwischen verschiedenen OSGi-Plattformen sondern auch die Integration OSGi-fremder Umgebungen an. Auf diese Weise entsteht eine verteilte OSGi-Plattform die heterogene IT-Systeme miteinander kombiniert. Durch die konsequente Anwendung einer SOA entstehen Anwendungsfälle, die in zwei Kategorien unterteilt werden können: Die Kommunikation zwischen Klienten und Diensten im Sinne einer SOA und die SOA-Integration von Geräten und Sensoren, welche die Eigenschaften einer SOA von sich aus nicht unterstützen.

4.1.1 OSGi in einer verteilten SOA

Die Umsetzung einer verteilten SOA für OSGi wird von Comoros im Kern durch die Realisierung der OSGi-Remote-Services-Spezifikation erreicht. Hier werden funktionale Anforderungen definiert, deren Umsetzung obligatorisch sind. Andere Anforderungen sind nicht Teil der Spezifikation.

Ein OSGi-Client nutzt einen entfernten OSGi-Service

Dieser Anwendungsfall ist integraler Bestandteil der RS-Spezifikation und wird in Abbildung 4.2 illustriert. Innerhalb dieses Anwendungsfalls muss der für die entfernte Nutzung bereitzustellende OSGi Service auf Kompatibilität analysiert werden. Einzelne Aspekte eines lokalen Service, wie die Verwendung von Call-by-Reference-Objekten im Service-Interface können die Verteilung verhindern und diese Services müssen demnach ausgefiltert werden. Anschließend wird eine zum Java-Interface des OSGi-Service passende Beschreibung mittels der WSDL abgebildet und ein DPWS-Skeleton erzeugt. Diese WSDL muss alle Informationen, die den OSGi-Service beschreiben, enthalten. Die Beschreibung geht über die Maße einer normalen Schnittstellendefinition hinaus, da nicht nur die Nutzung

beschrieben wird, sondern Informationen für die vollständige Rekonstruktion des Service enthalten sein müssen. Auf der Client Seite werden diese Informationen ausgelesen und ein entsprechender OSGi-Proxy erzeugt. Bei Service-Aufrufen übernimmt die Software die Aufgaben des Marshaling, die Überführung von Java-Objekten in XML-Dokumente und umgekehrt.

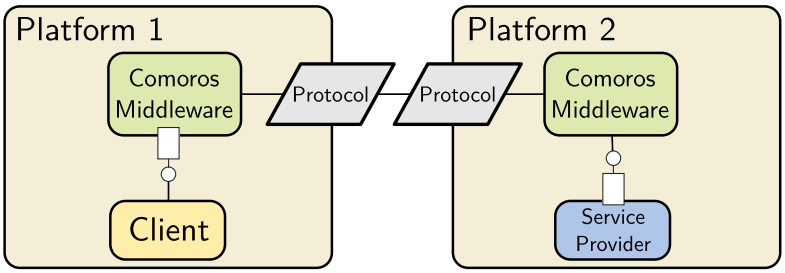


Abbildung 4.2: Ein OSGi-Client nutzt mit Hilfe der Comoros-Middleware einen entfernten OSGi-Service. Die Verteilung wird über verschiedene Kommunikationsprotokolle realisiert. Die Plattformen liegen entweder auf unterschiedlichen Rechnern oder auf dem selben Rechner aber in unterschiedliche JVM-Instanzen.

Ein OSGi-fremder Client nutzt einen entfernten OSGi-Service

Ein OSGi-fremder Client im Sinne von ws4d.Comoros ist ein Client im Sinne einer SOA, ist nicht innerhalb einer OSGi-Plattform integriert, auf keine Plattform festgelegt und kann beliebige Kommunikationsprotokolle, wie DPWS oder CoAP, implementieren. Solche Clients repräsentieren oftmals existierende Applikationen und können nicht modifiziert werden. Auch in diesem Anwendungsfall ist die Analyse des OSGi-Service auf Kompatibilität obligatorisch. Anschließend wird ein Service-Endpunkt in der entsprechenden Technologie erzeugt. Kann der Client wie oben erwähnt nicht modifiziert werden, muss die Schnittstelle des Service der erwarteten Schnittstelle des Clients entsprechen. In diesem Fall ist eine automatische Erzeugung des Skeletons nicht möglich, der Entwickler muss zusätzliche Informationen von außen an Comoros übergeben. Ist der Endpunkt erstellt, kann der OSGi-fremde Client diesen ansprechen, das Daten-Marshaling übernimmt ws4d.Comoros. Abbildung 4.3 zeigt den gesamten Anwendungsfall.

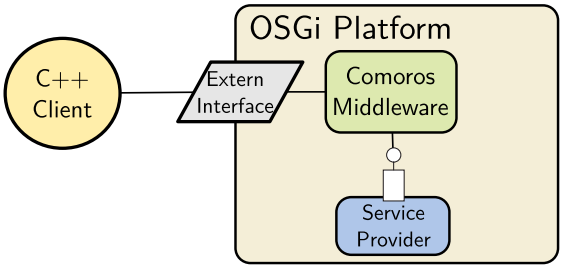


Abbildung 4.3: Ein OSGi-fremder Client nutzt mit Hilfe der Comoros-Middleware einen entfernten OSGi-Service. Die Plattform symbolisiert zugleich die Rechner- beziehungsweise die JVM-Grenze. Der Client ist nicht an eine bestimmte Sprache oder ein bestimmtes Kommunikationsprotokoll gebunden.

Ein OSGi-Client nutzt einen entfernten OSGi-fremden Service

Für diesen Anwendungsfall, illustriert in Abbildung 4.4, gelten die gleichen Bedingungen wie für den Vorherigen. Der OSGi-fremde Service wird analysiert, ein der Schnittstellen-Beschreibung entsprechendes Java-Interface generiert und in die OSGi-Plattform eingebunden. Ein OSGi-Proxy, der dieses Interface implementiert ermöglicht OSGi-Clients die Nutzung dieses Services. Falls ein bestehender OSGi-Client den Service nutzen will, d. h. das Interface festgelegt ist, muss auch hier ein gesteuertes Mapping auf das passende Java-Interface vollzogen werden. Das Marshaling der Daten übernimmt ebenfalls die Middleware.

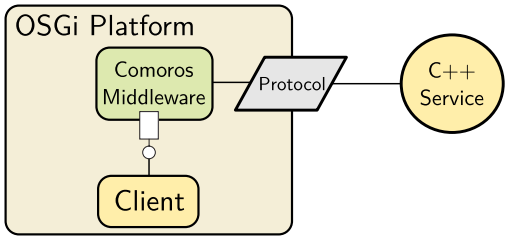


Abbildung 4.4: Ein OSGi Client nutzt mit Hilfe der Comoros-Middleware einen entfernten OSGi-fremden Service. Der Service ist nicht an eine bestimmte Sprache oder ein bestimmtes Kommunikationsprotokoll gebunden.

Unterstützung einer Event-basierten Kommunikation

Dieser Anwendungsfall unterscheidet zwei wesentliche Punkte, die Übertragung von Events zwischen OSGi-Plattformen und die Übertragung zwischen OSGi-Plattformen und OSGi-fremden Clients beziehungsweise Services (siehe Abbildung 4.5). Für die Übertragung von lokalen OSGi-Events an entfernte Plattformen wird standardmäßig das DPWS als Übertragungsprotokoll gewählt, da dieses eine ausgereifte Event-basierte Kommunikation über WS-Standards (WS-Eventing) enthält. Zu diesem Zweck wird eine DPWS-Event-Schnittstelle innerhalb einer OSGi-Plattform angeboten, über die entfernte Plattformen Events ausgewählter Topics abonnieren und als lokale OSGi-Events veröffentlichen können. Weiterhin können über dieselbe Schnittstelle sowohl native DPWS-Clients OSGi-Events abonnieren als auch Events von nativen DPWS-Geräten empfangen und als OSGi-Events veröffentlicht werden. Für nicht modifizierbare OSGi-fremde Clients/Services müssen erneut passende Schnittstellen generiert werden. Eine Unterstützung für weitere Technologien neben dem DPWS, wie z. B. das CoAP, kann analog realisiert werden.

Unterstützung des Life-Cycle von OSGi und OSGi-fremder Technologien

OSGi-Services können zur Laufzeit registriert, verändert und deregistriert werden. Auch Services OSGi-fremder Technologien können gestartet und gestoppt, und zumeist auch modifiziert werden. Die Topologie der von Comoros aufgespannten Netzwerkinfrastruktur kann sich folglich ständig ändern. Services verschwinden, ändern ihre Metadaten oder neue Services erscheinen. Diese Dynamik muss sowohl innerhalb der OSGi Service Plattform als auch im gesamten verteilten System durch Comoros berücksichtigt werden.

Aus diesem Grund werden geeignete Discovery-Mechanismen in Comoros integriert. Es

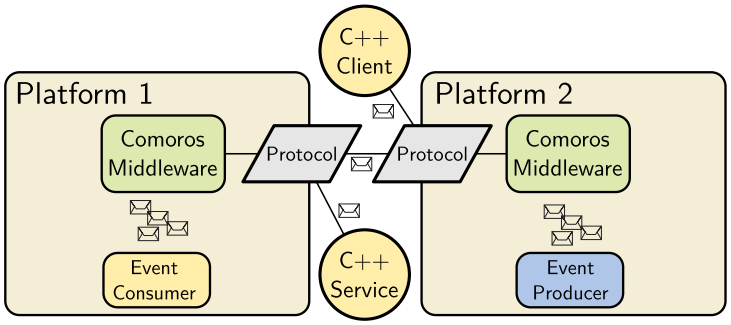


Abbildung 4.5: Event-basierte Kommunikation zwischen unterschiedlichen Kommunikationsendpunkten. Events, die in einer OSGi-Plattform veröffentlicht werden, können von entfernten OSGi-Plattformen oder OSGi-fremden Clients empfangen werden. Ebenso können auf der OSGi-Plattform Events OSGi-fremder Services empfangen werden.

ist notwendig, dass Service-Nutzer parametrisierte Service-Anfragen absenden können, um so über passende Services informiert zu werden. Solange nun eine Bindung zwischen Service und Client besteht, muss der Life-Cycle des Service überwacht werden und Zustandsänderungen entsprechend behandelt werden. Ist eine Suchanfrage erfolglos, bleibt aber offen, so muss der Client informiert werden, sobald ein passender Service im verteilten System registriert wird.

4.1.2 Geräteintegration

Die Bedeutung der Integration von Geräten, Sensoren und Aktoren in die OSGi-Plattform wurde bereits in Unterabschnitt 2.1.4 motiviert. Der Zugriff auf diese Geräte innerhalb einer SOA ist nicht immer einfach zu realisieren. Während Technologien wie UPnP oder DPWS von sich aus einen serviceorientierten Zugriff anbieten, ist dieser bei anderen Technologien, wie USB oder RS232, nicht vorhanden. Es ergeben sich folgende Anwendungsfälle:

Ein OSGi-Client nutzt Funktionen von Geräten, die keine Service-Schnittstelle anbieten

Der aktuelle Forschungsstand zur Integration von Geräten in die OSGi-Plattform wurde bereits ausführlich in Unterabschnitt 2.1.4 behandelt. Comoros setzt dabei auf den Mechanismen der *OSGi Device Access Specification* auf. In Comoros sollen aber auch serielle Geräte eine SOA-Schnittstelle erhalten. Ein Sensor, der z. B. die Pulsdaten eines Menschen misst, soll eine Operation zum Abfragen des aktuellen Pulses und eine Event-basierte Schnittstelle zum eigenständigen Ausliefern der Pulsdaten an interessierte Empfänger erhalten. Diese funktionale Ebene ist nicht Teil der *OSGi Device Access Specification* und muss innerhalb von Comoros ergänzt werden.

Nutzung von Datenströmen

Die Nutzung von Datenströmen ist eigentlich nicht Teil einer SOA, in der Teilnehmer über diskrete Nachrichten kommunizieren. Ein kontinuierlicher Datenstrom ist aber im Umfeld einer verteilten Dienste- und Geräte-Infrastruktur notwendig. Viele Geräte und Sensoren liefern kontinuierliche Daten von enormer Größe. Als Beispiel kann an dieser

Stelle ein EKG-Sensor dienen. Ein EKG-Sensor kann bis zu 12 Kanäle haben und Daten mit einer Abtastrate von 50 kHz ausliefern. Das Datenaufkommen ist somit enorm und eine Verpackung der einzelnen Messdaten in separate Nachrichten würde einen enormen Overhead verursachen, der gerade im Umfeld von Kleinstgeräten nicht vertretbar ist.

Aus diesem Grund muss Comoros Clients die Möglichkeit bieten, über eine SOA-Schnittstelle Datenströme abzufragen.

4.2 Heterogenität

Die Möglichkeit zur Integration heterogener Komponenten in verteilten Systemen im Allgemeinen, und in der durch Comoros definierten Umgebung im Speziellen, ist aus verschiedenen Gründen eine wichtige Anforderung. Neu geschriebene Komponenten müssen komplikationslos mit Altkomponenten zusammenarbeiten. Komponenten, die innerhalb einer OSGi-Plattform laufen müssen mit Komponenten die außerhalb einer solchen Plattform laufen zusammenarbeiten. All diese unterschiedlichen Komponenten können unterschiedliche Technologien bei der Implementierung von Diensten verwenden. Neben Unterschieden in den Plattformen und Programmiersprachen können je nach Eignung verschiedene Datenformate oder Kommunikationsprotokolle eingesetzt werden. Es ist klar ersichtlich, dass diese Heterogenität notwendig ist, die Komplexität des verteilten Systems dadurch aber enorm erhöht wird. Somit ergeben sich für Comoros zwei wichtige Anforderungen:

- *Unterstützung heterogener Systemstrukturen:* Comoros muss die Unterstützung von Komponenten unterschiedlicher Technologien umsetzen. Verschiedene Datenformate zur Konversation und variable Kommunikationsprotokolle müssen ebenso von Comoros unterstützt werden, wie die Integration von Altkomponenten im System.
- *Maskierung der Heterogenität:* Für den Benutzer von Comoros muss die Heterogenität verborgen sein. Die Nutzung von Komponenten unterschiedlicher Technologien darf sich nicht unterscheiden.

4.3 Transparenz

Eine verteilte Systemarchitektur, wie sie von Comoros aufgebaut werden muss, hat gegenüber lokalen Anwendungen eine erhöhte Komplexität. Für den Anwendungsentwickler ist es aber vorteilhaft, wenn ihm die Komplexität der verteilten Systemstruktur weitestgehend verborgen bleibt. Er soll sich nicht mit der durch die Verteilung hervorgerufenen Komplexität beschäftigen, sondern sich gänzlich der Anwendungsentwicklung widmen. Dieser Vorgang wird als Verteilungstransparenz definiert. Mittels dieses Konzepts werden interne Abläufe der Middleware durch einfache Transparenzfunktionen angeboten, was zu einem enormen Effizienzgewinn bei der Entwicklung und Benutzung verteilter Anwendungen führt.

Die Verteilungstransparenz ist nur ein Oberbegriff, unter dem eine Reihe von einzelnen Transparenzfunktionen zusammengefasst werden. Diese wurden ursprünglich in [ANS89] definiert und anschließend innerhalb des Standard für *Open Distributed Processing (ODP)* weiterentwickelt [ISO09a]. Die für Comoros wichtigen Arten der Transparenz sind:

4.3.1 Ortstransparenz

Bei der Ortstransparenz müssen Dienstanforderer nichts über physische Orte von Komponenten wissen. In dem durch Comoros aufgespannten verteilten Geräte- und Dienste-System müssen Clients in der Lage sein Dienste zu suchen und zu nutzen, ohne zu wissen, ob der Dienst lokal in der selben OSGi-Plattform oder in einer entfernten Plattform installiert ist, oder gar durch ein OSGi-fremdes Gerät angeboten wird.

4.3.2 Zugriffstransparenz

Innerhalb des verteilten Systems müssen Komponenten untereinander interagieren, indem Dienste angefordert und aufgerufen werden. Die Benutzung von lokalen und über Comoros entfernt angebotene Services darf sich aber nicht unterscheiden. Weder im Service noch im Client sind Änderungen im Code notwendig. Dem Entwickler muss aber die Nutzung eines entfernten Service bewusst sein und er muss mit den daraus resultierenden Eigenschaften (z. B. erhöhte Latenz) umgehen. Dazu müssen ihm Möglichkeiten geschaffen werden zwischen entfernten und lokalen Services zu unterscheiden und diese zu selektieren.

4.3.3 Fehlertransparenz

In einem verteilten System ist das Auftreten von Fehlern wahrscheinlicher als in einer lokalen Anwendung, da neue, verteilungsbezogene Fehlerarten entstehen. Im Zuge der Fehlertransparenz müssen diese neuen Fehlerarten von Comoros vor dem Benutzer verborgen werden. Alle durch die verteilte Kommunikation hervorgerufenen Fehler werden auf das vorhandene OSGi-Fehlermodell abgebildet. Somit muss der Entwickler keine gesonderte Fehlerbehandlung durchführen. Damit ein Client dennoch nachvollziehen kann, wenn ein Fehler die Verteilungsaspekte des Systems betrifft, werden neue Fehlerkategorien eingeführt.

4.4 Adaptierbarkeit der Middleware

Mit Hilfe von ws4d.Comoros ist es möglich umfangreiche und flexible Strukturen bestehend aus Geräten, Services, Clients und deren Assoziationen aufzubauen. Diese Strukturen können wechselnden Bedingungen und Anforderungen unterworfen sein, auf die adäquat reagiert werden muss. Die Komponenten der Software müssen demnach zur Laufzeit konfigurierbar sein und es muss weiterhin eine Schnittstelle für die Überwachung des Systems existieren. Es ergeben sich somit folgende Anforderungen:

- *Konfiguration* Die Entwicklung einer verteilten Applikation innerhalb einer komplexen Infrastruktur mit einer Vielzahl von Geräten und Services verlangt den Aufbau einer exakt spezifizierten und konfigurierten Kommunikation. Der Rahmen Konfiguration wird dabei durch ein Modell beschrieben, in dem die beteiligten Services, Clients, Geräte und Gateways als Komponenten bezeichnet und deren Beziehungen untereinander als Assoziationen modelliert werden. In diesem Modell wird festgelegt, welche Services auf welchen Plattformen zugreifbar sind, welche Kommunikationsprotokolle mit welchen Eigenschaften für die Kommuikations-Assoziazionen verwendet werden und zwischen welchen Plattformen eine Event-basierte Kommunikation realisiert wird. Abbildung 4.6 zeigt eine beispielhafte Konfiguration

der Kommunikation. Über eine entsprechende Konfiguration kann sichergestellt werden, dass innerhalb einer OSGi-Plattform nur die Services verteilt werden, für die eine Verteilung explizit erwünscht ist, nur die Events in entfernten Plattformen veröffentlicht werden, für die es Interessenten gibt, und nur die Services nativer Technologien integriert werden, für die aktive Clients zur Nutzung existieren.

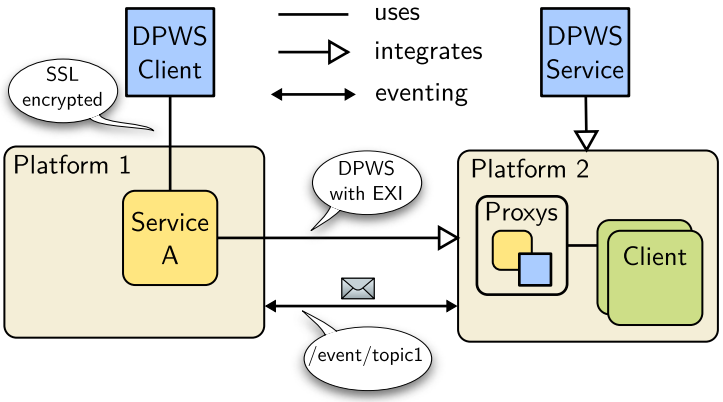


Abbildung 4.6: Aufbau und Parametrisierung einer durch Comoros aufgebauten Kommunikation

Die aufgebauten Assoziationen können zudem weiter parametrisiert werden um alle speziellen Anforderungen an das verteilte System zu erfüllen, wie z. B. die Realisierung von unterschiedlichen Effizienzgraden. So bestehen unter anderem Möglichkeiten für die Einstellungen von Sicherheitsmerkmalen, z. B. kann eine verschlüsselte Verbindung vorgeschrieben werden. Weiterhin können bei der Kommunikation zwischen Client und Service Protokollspezifika angegeben werden. Für eine möglichst effiziente Kommunikation kann bei Benutzung des DPWS-Protokolls die Verwendung von EXI vorgeschrieben werden oder auf ein völlig anderes Protokoll gesetzt werden. Assoziationen, welche die Event-basierte Kommunikation zwischen zwei Komponenten beschreiben, können auf bestimmte Event-Typen beschränkt werden.

Die resultierende Konfiguration ist flexibel und kann zur Laufzeit angepasst werden um wechselnden Anforderungen zu entsprechen.

- *Monitoring* Neben der Konfiguration des Systems ist das Überwachen interner Zustände aller Komponenten von zentraler Bedeutung. Um mögliche Probleme, Fehler und auftretende Ereignisse frühzeitig zu erkennen und darauf adäquat zu reagieren, muss es möglich sein den Zustand des Systems von außen zu überwachen. Zu diesem Zweck werden zwei unterschiedliche Typen von Statusvariablen eingeführt:

  1. Monitoring-Daten für Performanz-Analysen und Kapazitätsplanungen oder Logging-Daten, die Abläufe einzelner Prozesse dokumentieren, werden im laufenden Betrieb gesammelt und müssen bei Bedarf von interessierten Clients abgerufen werden können. Diese Daten werden nur passiv gesammelt.
  2. Kritische Daten, wie auftretende Fehler oder ein Mangel an Ressourcen müs-

sen an Monitoring-Clients aktiv übermittelt werden, sobald sie auftreten. Die überwachte Komponente muss die Übermittlung dieser Statusvariablen selbstständig anstoßen.

Sowohl Monitoring als auch Konfiguration nutzten dabei offene und akzeptierte Standards und bietet Entwicklern Zugriff auf die Schnittstellen via OSGi und DPWS. Nur so kann gewährleistet werden, dass auf einfache Weise die internen Komponenten und die externe Umgebung überwacht werden können und adaptiv auf wechselnde Bedingungen eingegriffen wird.

4.5 Softwareentwicklung

Bei der Softwareentwicklung müssen unterschiedlichste Aspekte beachtet werden um die Qualität der Software auf ein hohes Maß zu bringen. Deren Bedeutung wurde bereits in Unterabschnitt 2.1.8 unterstrichen. Qualitätsmodelle helfen Merkmale, Kriterien und Eigenschaften zur systematischen Definition und Messung von Softwarequalität zu abstrahieren. Dort werden konkrete Merkmale, wie Zuverlässigkeit, Funktionalität, Effizienz oder Wartbarkeit definiert, an denen sich folglich die Anforderungen an die Softwareentwicklung für Comoros ableiten. Dabei müssen die unterschiedlichen Blickwinkel, die der Benutzer der Software und der Entwickler von Comoros haben, beachtet werden.

- *Funktionalität:* Die funktionalen Anforderungen an Comoros sind in Abschnitt 4.1 festgelegt. Aus Sicht des Benutzers sind diese Funktionen notwendig und müssen daher vom Entwickler umgesetzt werden.
- *Effizienz:* Gerade um auf dem wachsenden Markt mobiler Anwendungen zu partizipieren wird Comoros für den Einsatz auf Geräten mit Ressourcenbeschränkungen ausgerichtet. Daher muss der Softwareentwurf einen Ressourcen schonenden und performanten Betrieb sicherstellen. Durch die Einführung von darauf ausgerichteten Metriken und den Einsatz von Messsystemen kann dieses Merkmal evaluiert werden und den Entwicklungsprozess beeinflussen.
- *Nutzerkomfort:* Eine einfache und komfortable Benutzung ist eine der wichtigsten Voraussetzungen für eine hohe Akzeptanz bei Benutzern und Entwicklern. Um dies zu erreichen müssen einerseits passende Softwarewerkzeuge für die Bedienung von Comoros existieren und andererseits der Entwicklungsprozess für die Erstellung einer verteilten Anwendung möglichst einfach sein. Diese Einfachheit kann mittels geeigneter Kennzahlen, wie z. B. *Lines of Code (LoC)* oder die von McCabe definierte *Cyclomatic Complexity* [McC76], bestimmt werden. Im Bereich der Softwarewerkzeuge für Comoros bietet sich unter anderem eine Unterstützung in der Konfiguration der Middleware an.
- *Wartbarkeit und Erweiterbarkeit:* Die Eigenschaft eines Systems, das einfach erweitert und gewartet werden kann, wird in der Literatur oft als *Offenheit* bezeichnet [Cou05; Tan06]. Um diese Eigenschaft zu erreichen, muss ein geeignetes Architekturmodell für das System verwendet werden. Comoros wird als lose gekoppelte Komponenten entwickelt, die wohl definierte und gut dokumentierte Schnittstellen besitzen, und hält sich in der gesamten Systemkonstruktion an anerkannte

Standards. So kann neue Funktionalität, wie z. B. die Unterstützung für ein neues Kommunikationsprotokoll oder ein neues Daten-Marshaling, einfach durch die Entwicklung neuer Komponenten in das Gesamtsystem integriert werden. Durch die Ausnutzung der Hot-Deployment-Eigenschaft der OSGi-Plattform kann diese Integration sogar im laufenden Betrieb erfolgen. Die Wartbarkeit des Systems wird ebenso durch die Aufteilung der Middleware in einzelne Komponenten wesentlich vereinfacht. Unterstützend wirkt dabei zusätzlich der Einsatz von Softwarewerkzeugen für die Überwachung, die Konfiguration und für das Deployment von Komponenten.

Die Umsetzung der angegeben Kriterien und Merkmale muss anhand einer systematischen Vermessung evaluiert werden.

4.6 Kompatibilität und Interoperabilität

Die Akzeptanz von neuer Software fußt im wesentlichen häufig auf einer einfachen Integrierbarkeit in ein bestehendes Umfeld. Zu diesem Zweck verfolgt ws4d.Comoros folgende Anforderungen:

- *Konsistenz zum bestehenden OSGi-Programmiermodell:* Für die Realisierung der Verteilungsfunktionalitäten sollen nur OSGi-eigene Mechanismen verwendet werden. Der zentrale Aufrufmechanismus über die OSGi Service Registry darf für den Zugriff auf entfernt liegende Services nicht verändert werden. Die Verteilungseigenschaften eines Service sollen ausschließlich über seine Properties, basierend auf dem bestehenden OSGi-Metamodell, beschrieben werden oder für Legacy-Services über eine entsprechende Konfiguration der Middleware. OSGi-fremde Erweiterungen oder Implementierungen spezieller Schnittstellen dürfen nicht verlangt werden.
- *Unterstützung und Verwendung bestehender Standards:* Neben der Einhaltung der OSGi-Spezifikation ist die *WS-DD*-Spezifikation ein weiterer elementarer Baustein, da DPWS als Basis-Kommunikationsprotokoll verwendet wird. Auch für die Realisierung der aufgestellten Anforderungen dürfen nur bestehende Standards genutzt werden oder Neuentwicklungen auf solchen Standards aufbauen. Zum Beispiel sind dies im beschriebenen Management-Bereich im Detail der OSGi-Config-Admin für die Konfiguration der Middleware und der OSGi-Monitor-Admin für die Überwachung. Die Plattform übergreifende Event-basierte Kommunikation wird mit Hilfe des OSGi-Event-Admin und WS-Eventing realisiert.
- *Integration von Altkomponenten:* Beim Aufbau einer verteilten OSGi-basierten SOA ergibt sich das Problem, dass für viele Anwendungsfälle bereits Services existieren, die seinerzeit nicht für eine verteilte Nutzung entwickelt wurden. Diese entfernte Nutzung der so genannten Legacy-Services muss von Comoros ohne die Anpassung ihrer Implementierungen erfolgen. So wird sichergestellt, dass bestehende Systeme und Anwendungen ohne Anpassungen in eine verteilte Umgebung integriert werden können.
- *Unabhängigkeit von der OSGi-Implementierung:* Es gibt eine Reihe verschiedener Implementierungen der OSGi-Spezifikation, die zum Teil eigene, zusätzliche Funktionen und Eigenschaften bieten. Um mit allen verfügbaren Implementierungen

kompatibel zu sein, darf die zu entwickelnde Software ausschließlich Funktionen der OSGi-Spezifikation verwenden.

4.7 Gemeinsame Ressourcennutzung

Ressourcen im Kontext von Comoros bezeichnen Daten und Hardware. Oftmals müssen Ressourcen von mehreren Benutzern gemeinsam genutzt werden. Als Beispiel kann hier die Überwachung der Vitalparameter eines Patienten durch mehrere Supervisoren dienen. Der Patient trägt Sensoren am Körper, die Vitaldaten aufzeichnen und diese über die Comoros-Middleware anbieten. Die Supervisoren haben nun sowohl gemeinsamen lesenden Zugriff auf die Daten der Sensoren, als auch gemeinsamen schreibenden Zugriff, für die Konfiguration der Geräte.

Wenn Ressourcen nicht exklusiv von einem Benutzer, sondern von mehreren Benutzern auf unterschiedlichen Rechnern genutzt werden, so muss eine geeignete Zugriffsverwaltung eingeführt werden. In Comoros muss zwischen exklusiv nutzbaren Ressourcen und gemeinsam nutzbaren Ressourcen unterschieden werden. Bei einer exklusiven Nutzung muss jeder weitere Benutzer über die belegte Ressource benachrichtigt werden. Der gemeinsame lesende Zugriff auf eine Ressource muss auch verwaltet werden. Ein Datenstrom kann z. B. nicht beliebig oft repliziert werden, da ansonsten die Performanz stark leidet.

4.8 Datensicherheit

Im Bereich der Sicherheit in der Kommunikation müssen von Comoros ein Sicherheitsmodell umgesetzt werden, das im Sinner der Schutzziele in der Informationssicherheit die *Integrität*, die *Verbindlichkeit*, die *Vertraulichkeit* und eine *Authentifizierung* realisiert. Weiterhin soll ein Authorisierungsmechanismus eingeführt werden.

Bei der Umsetzung der Schutzziele wird auf Standard-Bibliotheken zurückgegriffen, wie z. B. JSSE, eine Java-Bibliothek die unter anderem den Aufbau von auf SSL basierende HTTPS-Verbindungen ermöglicht. Die Sicherheitsfunktionen werden nicht direkt durch Comoros implementiert sondern sind Bestandteil der verwendeten JMEDS-Bibliothek.

4.9 Zwischenfazit

Die Comoros-Software realisiert eine Middleware für den Aufbau einer verteilten Dienste- und Geräte-Infrastruktur und ermöglicht dem Softwareentwickler komfortabel verteilte Anwendungen in dieser Systemlandschaft zu entwickeln. Die bestehenden Lösungen, vorgestellt in Kapitel 3, bieten allesamt nur Speziallösungen zu Teilproblemen im Bereich einer umfassenden Anwendungsentwicklung für ambiente Systeme, lassen aber jeweils wichtige Gebiete unerschlossen. Die hier aufgestellten Anforderungen definieren Eigenschaften, die Comoros diese Lücke schließen lassen. Im einzelnen werden Anforderungen für den Aufbau einer verteilten OSGi-Umgebung definiert, für die Integration von Geräten und OSGi-fremden Services, für die Heterogenität und die Transparenz der Benutzung, für die Softwareentwicklung, für die Kompatiblität zu besehenden Standards und für die Datensicherheit. Die Anforderungen sind im Detail in Tabelle 4.1 zusammengestellt.

Der Softwareentwurf für Comoros muss diese Anforderungen nun realisieren. Dazu wird eine passende Architektur entwickelt, die sowohl die Umsetzung der funktionalen und der nicht-funktionalen Anforderungen sicherstellt. Weiterhin bleibt der Fokus von

Comoros auf eine komfortable Softwareentwicklung der verteilten Anwendungen und eine effiziente Umsetzung.

5 Kernarchitektur

Zur Umsetzung der in Kapitel 4 definierten Anforderungen wurde die Software ws4d.Comoros entwickelt. Diese Software basiert auf wohl bekannten Standards, insbesondere der Remote-Services-Spezifikation, die durch die OSGi-Allianz als Teil der Version R4.2 der Compendium-Spezifikation eingeführt wurde. Diese Spezifikation erfüllt einen Teil der funktionalen Anforderungen und dient als Basis für Comoros. Betrachtet man die RS-Spezifikation kann Comoros in zwei Teile unterteilt werden: in den Kern, der eine vollständige Implementierung der OSGi-Remote-Services-Spezifikation realisiert, und einer Erweiterung, die notwendig war um alle genannten Anforderungen umzusetzen. Dieses Kapitel beschreibt die Kernarchitektur und beantwortet insbesondere folgende Fragestellungen:

- Wie ordnet sich der Kern in die Gesamtarchitektur ein?
- Wie sieht der Architekturentwurf für die Kernarchitektur aus?
- Wie sieht der Entwurf auf der Client- und auf der Server-Seite aus?
- Wie verläuft die Interaktion zwischen der Client- und der Server-Seite?

Abbildung 5.1 gibt eine Übersicht über die Gesamtarchitektur. Die OSGi-Service-Plattform bietet die technische Grundlage von Comoros. Durch ein SOA-basiertes Programmiermodell und der damit verbundenen Modularisierung ist Comoros in verschiedene Komponenten aufgeteilt. Auf der Kommunikationsschicht befinden sich Komponenten, die verschiedene Kommunikationsprotokolle implementieren. Für den Kern, also die Umsetzung der OSGi-Remote-Services, wird das *Devices Profile for Webservice (DPWS)* verwendet. Durch die Komponentenorientierung können weitere Protokolle durch die Entwicklung zusätzlicher Komponenten leicht ergänzt werden, diese sind dann Teil der erweiterten Architektur. Auf dem DPWS aufsetzend realisiert der Comoros Kern

die OSGi-Remote-Services-Spezifikation. Die Gesamtsoftware setzt auf den Kern auf, erweitert diesen um weitergehende Funktionen, und kann die zusätzlichen Protokolle verwenden.

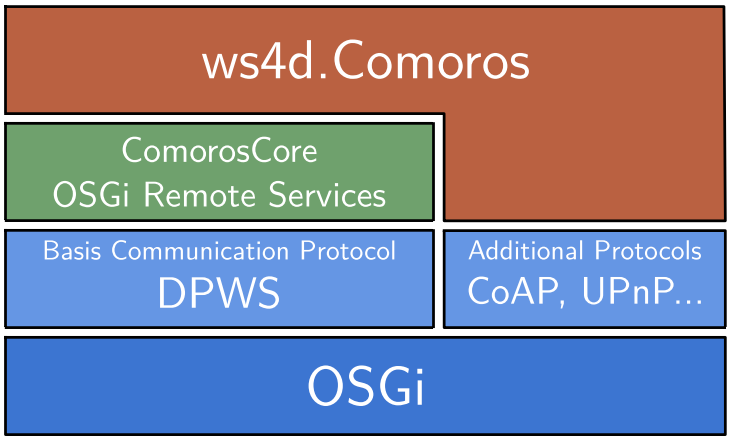


Abbildung 5.1: Überblick über die Comoros-Architektur

Für die Realisierung der Kernarchitektur wurden die in Tabelle 5.1 angegebenen Mechanismen eingesetzt. Einige der Mechanismen beruhen auf bekannten Prinzipien, andere wurden neu eingeführt.

Tabelle 5.1: Eingesetzte Mechanismen zur Realisierung der Kernarchitektur

Mechanismus	*Fremd*	*Erläuterung*
Proxy/Skeleton	X	Auf RMI basierendes Prinzip für die Kommunikation in verteilten Systemen
Umgebungsüberwachung	X	Überwachung der OSGi- und DPWS-Umgebung mit den dort realisierten Technologien
Proxy/Skeleton-Generierung		Dynamische Generierung von Proxy und Skeleton in Folge der Umgebungsüberwachung
Kompatibilitätscheck		Automatische Überprüfung, ob ein Service für die entfernte Nutzung kompatibel ist
Strukturabbildungen		Abbildungen der OSGi-Strukturen in die DPWS-Welt und umgekehrt
Service-Suche	X	Effiziente entfernte Service-Suche in der verteilten OSGi- und DPWS-Umgebung aufgrund einer Client-Nachfrage
Interface-Bereitstellung		Automatische Generierung und Bereitstellung von Service-Interfaces
Marshaling	X	Effizientes Marshaling der Daten zur Kommunikation zwischen OSGi und DPWS

5.1 OSGi Remote Services auf Basis des Devices Profile for Webservices

Für die Realisierung der plattformübergreifenden Kommunikation zwischen OSGi-Komponenten wird die OSGi-basierte, lokale serviceorientierte Architektur mittels des DPWS

erweitert. Mit Bezug auf das Modell der verteilten Objektsysteme [Tan06] wird zwischen einem Client, der einen Service nutzt, und einem Server, der einen Service anbietet unterschieden. Im Prozess der entfernten OSGi-basierten Kommunikation werden drei Phasen unterschieden (siehe Abbildung 5.2). Begonnen wird mit der Bereitstellungsphase, in der die Middleware auf der Client- und auf der Server-Seite alle benötigten Komponenten für die konkrete Kommunikation zwischen einem lokalen OSGi-Client `A` und einem entfernten OSGi-Service `B` erstellt und im Framework registriert. Dabei handelt es sich im wesentlichen um die Registrierung eines Skeletons auf der Server-Seite und eines Proxys auf der Client-Seite, sowie um die Verteilung der Service-Interfaces und die Bereitstellung der benötigten Services für das Daten-Marshaling. In der zweiten Phase läuft die eigentliche Kommunikation ab. Diese ist nicht auf einen einzelnen Serviceaufruf beschränkt, sondern wird erst beendet, wenn der Client kein weiteres Interesse an dem Service hat. Die dritte Phase umfasst schließlich die Deinstallation aller nicht mehr benötigten Komponenten.

Abbildung 5.2: Die drei Phasen für den Ablauf einer entfernten OSGi-Servicenutzung

5.1.1 Bereitstellungsphase

Abbildung 5.3 gibt einen Überblick über die Architektur des Comoros-Kerns, in welchem der Bereitstellungsprozess aller für die entfernte Servicenutzung benötigter Komponenten illustriert ist. Das OSGi-Framework auf der rechten Seite agiert in diesem Beispiel in der Rolle des Servers und bietet einen Service innerhalb des Bundles `A` an. Das Bundle `B` innerhalb des Client-OSGi-Frameworks will diesen Service nutzen. Die Bereitstellungsphase startet mit einer lokalen Suchanfrage des Bundles `B` an die OSGi-Service-Registry für einen Service `A`, der die Interfaces `X` und `Y` implementiert (1). Diese Suchanfrage wird durch die *Distribution Software (DSW)* von Comoros abgefangen (2). Für die Suche nach entfernten Services wird nun das DPWS-Protokoll verwendet. Hierfür ist das JMEDS-Framework, die verwendete DPWS-Implementierung, als OSGi-Bundle verpackt und in Comoros integriert worden. JMEDS sendet nun eine DPWS-`Probe`-Nachricht, welche die Informationen der lokalen Suchanfrage enthält, mittels einer UDP-Multicast-Nachricht an das Netzwerk (3). Die JMEDS-Instanz auf dem Server-OSGi-Framework empfängt anschließend diese `Probe`-Nachricht. Comoros bildet sie daraufhin auf eine OSGi-Suchanfrage ab, mittels derer die lokale OSGi-Service-Registry befragt wird, und findet so den passenden Service, angeboten von Bundle `A` (4). Um den gefundenen lokalen Service auch entfernt nutzen zu können, müssen passende Marshaling-Services für die verwendeten Datentypen der Service-Operationen zur Verfügung stehen. Diese notwendige Voraussetzung wird von Comoros überprüft (5) und im Erfolgsfall ein DPWS-Device `A` erstellt, der den entsprechenden Skeleton-Service `A` enthält (6). Anschließend sendet JMEDS eine `ProbeMatch`-Nachricht zurück an die anfragende Instanz (7). Ähnlich dem Prozess auf der Server-Seite wird auch auf dem Client-Framework die Verfügbarkeit der notwendigen Marshaling-Services überprüft (8) und Comoros erstellt daraufhin einen

DPWS-Proxy `A` (9), der als OSGi-Service `A` mit den implementierenden Interfaces `X` und `Y` im Framework registriert wird. Da das Bundle `B` aufgrund seiner Suchanfrage weiterhin auf die Verfügbarkeit dieses Service lauscht, wird eine Bindung erstellt und der Service kann verwendet werden (10).

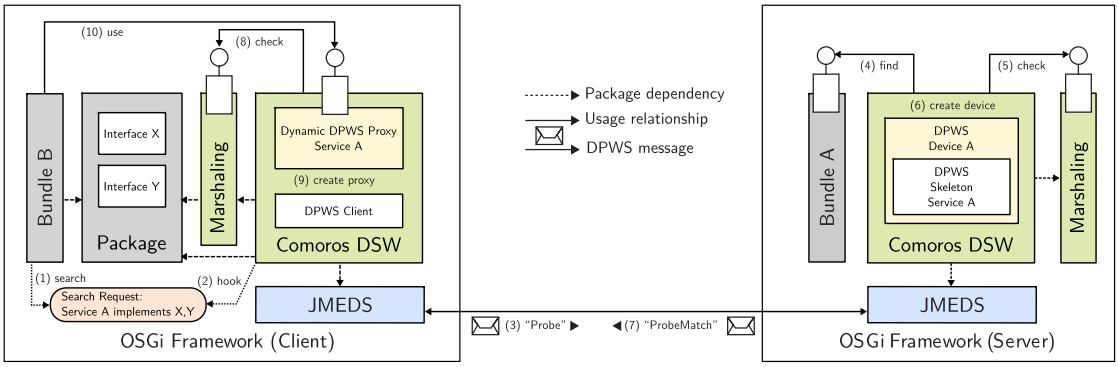


Abbildung 5.3: Vorgang der Skeleton- und Proxy-Erzeugung in der Comoros-Kernarchitektur

5.1.2 Kommunikationsphase

Wie in Unterabschnitt 2.1.3 beschrieben, sind OSGi-Services einfache Java-Objekte, die in der OSGi-Service-Registry registriert wurden. Für einen entfernten Zugriff auf diese Objekte bietet sich daher der Ansatz der *Remote Method Invocation* an (siehe Unterabschnitt 2.1.5). Wie bei der Bereitstellung bereits beschrieben, wird dem RMI-Modell folgend, auf der Client-Plattform über einen Client-Stub ein Proxy-Service registriert, der die gleichen Interfaces implementiert, wie der lokale Service auf der Server-Seite.

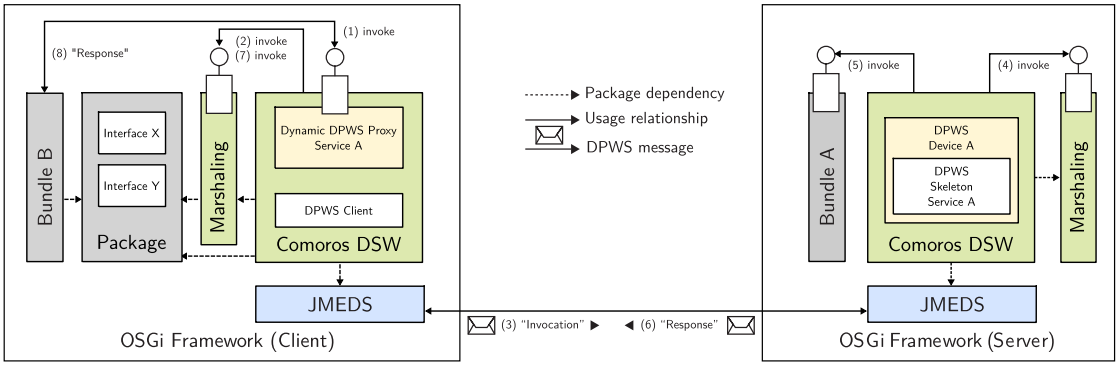


Abbildung 5.4: Vorgang eines entfernten Serviceaufrufs in der Comoros-Kernarchitektur

Abbildung 5.4 gibt einen Überblick über den entfernten OSGi-Service-Aufruf mittels der Comoros-Kernsoftware. Dieser startet mit dem Aufruf des DPWS-Proxy-Service `A` durch den Client in Bundle `B` (1). Um den Serviceaufruf mittels des DPWS-Protokolls an die Server-Seite verschicken zu können, müssen die Nutzdaten in eine XML-Struktur überführt werden. Diese Aufgabe übernimmt der Marshaling-Service (2). Anschließend kann ein DPWS-Serviceaufruf erstellt werden, der über die JMEDS-Instanz in Form einer `Invocation`-Nachricht an die Server-Seite geschickt wird (3). Auf der Server-Plattform

muss dieser entfernte Aufruf des Clients bzw. Proxys entgegengenommen werden. Diese Aufgabe übernimmt ein Objektadapter, der eine Schnittstelle für entfernte Aufrufe nach außen bereitstellt. Die Rolle des Objektadapters übernimmt das JMEDS-Bundle. Der Objektadapter leitet den externen Aufruf an das Skeleton-Objekt `A` weiter, das an den konkreten OSGi-Service `A` angepasst ist. An dieser Stelle müssen die XML-Nutzdaten wieder mittels des Marshaling-Service in Java-Objekte überführt werden (4), bis das Skeleton-Objekt schließlich den lokalen OSGi-Service `B` mit diesen Daten aufruft (5). Die Rückgabewerte werden über eine DPWS-`Response`-Nachricht zurück an die Client-Seite versendet (6) und auf beiden Seiten erneut in das jeweils benötigte Datenformat überführt (7). Abschließend erhält der Client `B` die Rückgabe über den Proxy-Service.

5.1.3 Deinstallationsphase

Die Kommunikationsphase kann, abgesehen von Fehlerfällen, durch zwei Ereignisse beendet werden. Im ersten Fall wird der entfernte Service `A` auf der Service-Seite deinstalliert. Der DPWS-Skeleton-Service `A` ist an den Lebenszyklus des entsprechenden OSGi-Service gebunden und wird dementsprechend auch beendet. Daraufhin versendet das JMEDS-Framework eine `Bye`-Nachricht per UDP-Multicast in das Netzwerk, die von der JMEDS-Instanz auf der Client-Seite empfangen wird. Diese Information wird an die Comoros-DSW weitergeleitet, welche die Deregistrierung des OSGi-Proxy-Service `A` vollzieht. Über OSGi-Mechanismen wird der Client `B` über den Wegfall benachrichtigt. Im zweiten Fall benötigt der Client `B` den entfernten Service nicht mehr, gibt ihn frei und schließt den entsprechenden Service-Tracker. Dieses Ereignis wird über den OSGi-Hook-Mechanismus durch die DSW abgefangen. Existiert im gesamten lokalen Framework auf der Client-Seite kein weiterer Client, der Interesse an dem Service `A` hat, so wird der Proxy-Service `A` deregistriert. Dieser Vorgang wird der Comoros-DSW auf dem Server-Framework mitgeteilt. Nach einer Überprüfung, ob keine weiteren entfernten Client-Frameworks den Service `A` nutzen, wird der DPWS-Skeleton-Service `A` beendet.

5.2 Skeleton

Der Hauptanwendungsfall der Comoros-Kernsoftware, die Nutzung eines entfernten OSGi-Service durch einen lokalen OSGi-Client, kann in zwei Seiten unterteilt werden, in die Client- und in die Server-Seite, die in diesem Abschnitt behandelt wird. Um entfernt zugreifbar zu sein, muss ein OSGi-Service durch ein Skeleton-Objekt repräsentiert werden. Innerhalb der Comoros-Software übernimmt die Verwaltung von Skeletons der Skeleton-Generator, der innerhalb der Distribution Software von Comoros integriert ist. Die Aufgaben des Skeleton-Generators sind in Abbildung 5.5 dargestellt. Die Distribution Software überwacht die lokale OSGi-Umgebung nach speziell ausgezeichneten Services (1). Wird ein solcher Service registriert, wird dieser an den Skeleton-Generator übergeben und analysiert, ob er alle Anforderungen für eine verteilte Nutzung erfüllt (2). Der Skeleton-Generator bildet nun alle Strukturinformationen des lokalen OSGi-Service im Skeleton ab (3) und erstellt in diesem Prozess ein DPWS-Device inklusive des DPWS-Skeleton-Service (4). Diese Schritte werden im Folgendem näher betrachtet.

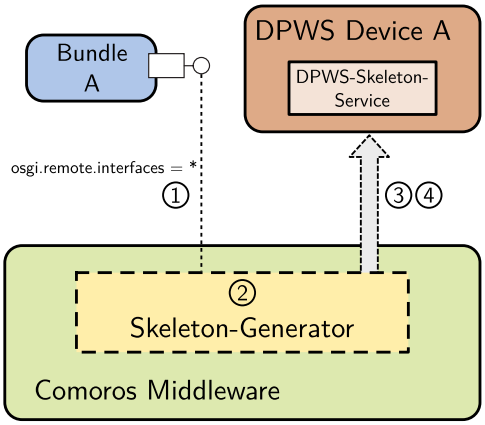


Abbildung 5.5: Prozess der Erstellung eines Skeleton durch den Skeleton-Generator

5.2.1 Überwachung der Umgebung

Wie in Abschnitt 4.1 formuliert, muss die dynamische Struktur der OSGi-Plattform bei der Entwicklung einer Remote-Services-konformen Middleware berücksichtigt und darf nicht eingeschränkt werden. Auf der Server-Seite bedeutet dies, dass DPWS-Skeletons dynamisch erzeugt werden und an den Lebenszyklus der repräsentierenden OSGi-Services gebunden sind.

Für die Umsetzung dieser Anforderung überwacht der Skeleton-Generator im ersten Schritt die lokale OSGi-Plattform und identifiziert die Services, die für eine entfernte Nutzung bereitgestellt werden sollen. Diese Services wurden, der RS-Spezifikation folgend, mit der speziellen Property `osgi.remote.interfaces` registriert. Diese Property spezifiziert genau die Service-Interfaces, über die der OSGi-Service entfernt zugreifbar sein soll. Somit können Interfaces eines OSGi-Service in die Menge der Interfaces unterteilt werden, die lokal implementiert werden *(ImplIf)*, die Menge der Interfaces die lokal registriert werden *(RegIf)* – dargestellt über die objectClass-Property – und schlussendlich in die Menge der Interfaces, die für eine entfernte Nutzung vorgesehen sind *(RemoteIf)*.

$$RemoteIf \subseteq RegIf \subseteq ImplIf \tag{5.1}$$

Diese spezielle Remote-Property kann, genau wie jede normale OSGi-Property, zur Laufzeit den Inhalt verändern oder ganz entfallen. Der Skeleton-Generator erkennt dies und passt den vorhandenen Skeleton entsprechend an oder entfernt ihn gegebenenfalls.

5.2.2 Analyse des Remote-Service

Wird ein OSGi-Service registriert, der die genannten Voraussetzungen erfüllt, muss der Service von Comoros im nächsten Schritt überprüft werden, ob er alle Anforderungen an einen Remote-Service erfüllt. Hierbei handelt es sich um Anforderungen bezüglich seiner *Intents*, seiner *ConfigurationTypes* und Kompatibilität bezüglich der verwendeten Datentypen.

Kompatibilität der ConfigurationTypes

ConfigurationTypes sind innerhalb der RS-Spezifikation definiert und beschreiben zum einen die Technologie, über die ein Service für entfernte Zugriffe veröffentlicht wird, und zum anderen dessen konkreten Zugriffspunkt. Ein OSGi-Service registriert dafür die Property `service.exported.configs`, deren Inhalt unterschiedliche Kommunikations-Technologien über eindeutige Namen auszeichnen. Diese Namen sind Basis für weitergehende Properties, die den exakten Kommunikations-Endpunkt für die jeweilige Technologie und den jeweiligen Service beschreiben. Ein konkretes Beispiel für Comoros kann folgende Struktur besitzen:

```
service.exported.configs = org.ws4d.dpws
org.ws4d.dpws.address    = 129.217.16.20
org.ws4d.dpws.port       = 8090
org.ws4d.dpws.path       = service-id/24
```

Comoros, in der Rolle als Distribution Provider, definiert sein unterstütztes Vokabular (hier: `org.ws4d.dpws`) und vergleicht es mit dem Vokabular des zu exportierenden Service. Gibt es keine Übereinstimmung, wird der Service vom Skeleton-Generator als inkompatibel vermerkt und verworfen. Gibt es, wie in diesem Beispiel, eine Übereinstimmung, so liest Comoros die weiteren Properties aus und verwertet die abgelegten Informationen für die Erstellung des Skeleton-Service.

Kompatibilität der Intents

Intents bilden ebenso wie ConfigurationTypes eine besondere Kategorie von Properties. Sie bezeichnen abstrakte Eigenschaften bezüglich der Verteilung, so können z. B. QoS-Anforderungen und -Zusagen oder generelle Beschränkungen für die Verteilung von Services formuliert werden. Beispielsweise kann ein Intent `confidentiality.message` die geforderte Vertraulichkeit auf Nachrichtenebene spezifizieren. Für den Intent-Mechanismus definiert die RS-Spezifikation die Properties `service.intents`, `service.exported.intents` und `remote.intents.supported`. Innerhalb dieser Properties spezifizieren jeweils der Service Anbieter und der Distribution-Provider, hier Comoros, eine Menge von Vokabeln. Ein Vergleich dieser Vokabeln steuert, ob ein OSGi-Service von Comoros verteilt werden kann oder nicht. Das Intent-Konzept ist aus der *SCA Policy Framework specification (SCA)* [Bei07] abgeleitet, in der ein Vokabular vorgeschlagen wird. Comoros unterstützt folgende von dem SCA-Policy-Framework definierte Intents:

`SOAP` mit den Qualifier `v1_1, v1_2`
`confidentiality` mit dem Qualifier `message`
`integrity` mit dem Qualifier `message`
`serverAuthentication` mit dem Qualifier `message`
`clientAuthentication` mit dem Qualifier `message`

Kompatibilität der Datentypen

Sind die Voraussetzungen bezüglich der Intents und der ConfigurationTypes erfüllt, analysiert der Skeleton-Generator innerhalb von Comoros im letzten Schritt die zu

exportierenden Interfaces auf Inkompatibilitäten der verwendeten Datentypen. Generell wird in dem durch Comoros erstellten verteilten System eine Call-by-Value-Semantik verfolgt. Eine Call-by-Reference-Semantik, die bei einigen Datentypen, wie z. B. bei einem `java.io.File`-Objekt, notwendig ist, kann im Allgemeinen nicht unterstützt werden. Enthält eine Methode eines Service-Interfaces einen solchen Datentyp, wird sofort das ganze Interface als inkompatibel markiert. Im Rahmen der Call-by-Value-Semantik legt die RS-Spezifikation innerhalb der als *Data Fencing* bezeichneten Restriktionen bezüglich erlaubter Datentypen in einem Service-Interface eine minimale Menge von Datentypen fest, die zwingend unterstützt werden müssen. Diese sind alle primitiven Datentypen, deren Wrapper, sowie eindimensionale Arrays und Collections bestehend aus den Wrapper-Klassen. Innerhalb des Comoros-Kerns wird diese Menge um Streams und komplexe Arrays und Collections erweitert, so dass folgende Datentypen direkt unterstützt werden:

```
1 type       ::= scalar | collection | array
2 scalar     ::= String | Integer | Long | Float | Double | Byte |
3                Short | Character | Boolean | Calendar | Date |
4                InputStream | ObjectInputStream
5 primitive  ::= int | long | float | double | byte |
6                short | char | boolean
7 array      ::= <Array of primitive> | <Array of type>
8 collection ::= <Collection of type>
```

Menge der kompatiblen Services

Schlussendlich ergibt sich aus der Analyse eine Menge von Services, die exportiert werden. Die Menge der Service-Interfaces, die für einen entfernten Zugriff markiert wurden, ist bereits in Gleichung 5.1 definiert worden. In der Service-Sicht gelten weiterhin folgende Aussagen:

$$RemoteServ \subseteq AllServ \tag{5.2}$$

$$AllServ := RemotableServ \cup NonRemotableServ \tag{5.3}$$

$$ExportedServ := RemoteServ \cap RemotableServ \tag{5.4}$$

RemoteServ definiert alle Services, die für eine entfernte Nutzung markiert wurden, *RemotableServ* umfasst die Services, die nach der Analyse als kompatibel gelten, und *ExportedServ* sind letztendlich die Services, die durch entfernte OSGi-Clients genutzt werden können.

5.2.3 Integration des DPWS in OSGi

Bei der Umsetzung einer RS-konformen Implementierung mittels DPWS als Kommunikationsprotokoll wird das DPWS in das OSGi-Umfeld integriert. Obwohl beide Technologien eine serviceorientierte Architektur umsetzen, stellt diese Integration eine Herausforderung dar, da DPWS und OSGi konzeptionelle Unterschiede aufweisen.

- **Konzept:** DPWS ist für verteilte, sprachunabhängige Umgebungen ausgelegt. Die Kommunikation erfolgt nachrichtenbasiert über das XML-basierte SOAP-

Protokoll. OSGi hingegen ist eine Java-basierte Komponentenplattform, die von allen spezifischen Merkmalen dieser Sprache Gebrauch macht.

- **Datentypen:** Die OSGi-Service-Plattform ist in Java spezifiziert und verwendet die entsprechenden Java-Datentypen. DPWS-Services werden hingegen über die *Webservice Definition Language (WSDL)* spezifiziert, innerhalb derer die verwendeten Datentypen über XML-Schema definiert werden.
- **Strukturen:** OSGi-Services werden innerhalb von Softwarekomponenten (Bundles) implementiert und kommunizieren über normale Java-Methodenaufrufe. DPWS-*Services* werden hingegen in *Devices* gruppiert und definieren *Operations*, die von Clients angesprochen werden können.

Bedingt durch diese konzeptionellen Unterschiede entsteht ein heterogenes Gesamtsystem. Für die Integration des DPWS in das OSGi-Umfeld müssen diese Unterschiede weitestgehend überwunden und verborgen werden. Dazu beschreibt dieser Abschnitt, wie OSGi-Strukturen in das DPWS abgebildet werden können, was auch ein komplettes Serialisierungskonzept zur Übertragung von Java-Datentypen in DPWS beinhaltet.

Die grundlegende Architektur für eine Nutzung von entfernten OSGi-Services wurde bereits zu Beginn des Kapitels vorgestellt. Für die entsprechenden OSGi-Services werden auf der Server-Seite DPWS-Skeletons und auf der Client-Seite nachfolgend spezielle Proxies erstellt. Damit der lokale Client den entfernten Service auf die gleiche Weise nutzen kann wie ein Client auf der entfernten Plattform, muss der Proxy-Service exakt auf die gleiche Weise wie der originale Service registriert werden. Die notwendigen Informationen über den OSGi-Service werden dazu im DPWS-Skeleton abgebildet, der neben seiner Funktion als Stellvertreter so entworfen wird, dass er eben jene Informationen abbildet. Die zentrale Fragestellung ist nun, wie ein OSGi-Service registriert wird und welche Informationen sich daraus für eine exakte Abbildung des Service ergeben. Die OSGi-Serviceregistrierung wurde in Unterabschnitt 2.1.3 erläutert, für die Registrierung des Proxys wird zusammengefasst folgendes benötigt:

1. Die Namen der Interfaces, unter denen der Service registriert und für eine entfernte Nutzung registriert wurde.
2. Properties, mit denen der Service registriert wurde.
3. Signaturen aller Methoden der Interfaces.

Punkte 1.) und 2.) sind direkt für die Registrierung des Proxy-Service erforderlich. Punkt 3.) betrifft eine Implementierungsbesonderheit der von Comoros bereitgestellten Proxys. Proxys werden hier generisch über das seit Java 1.3 eingeführte Prinzip der Dynamischen Proxys bereitgestellt [Goe05]. Ein Dynamischer Proxy fungiert dabei als Stellvertreter für jedes beliebige Service-Interface, die Aufrufe der einzelnen Methoden werden innerhalb eines *InvocationHandler* über das Java-Reflection-API realisiert. Über die Schnittstelle des InvocationHandler werden Methoden und Parameter des jeweiligen Aufrufs übermittelt und es wird somit eine generische Implementierung des Methodenaufrufs ermöglicht. Im Falle von Comoros werden Methodenaufrufe an die entsprechende Operation des

DPWS-Skeletons weitergeleitet. Für die Identifizierung der richtigen Operation wird dabei die exakte Methodensignatur benötigt.

Bis jetzt gehen wir davon aus, dass die Service-Interfaces sowohl auf der Server-, als auch auf der Client-Seite vorhanden sind. Ist dies nicht der Fall, müssen die Service-Interfaces auf der Client-Seite generiert werden. Hierfür sind weiterführende Informationen notwendig:

4. Informationen über Vererbungshierarchien in den Service-Interfaces.

Neben diesen Punkten müssen Überlegungen über eine grundlegende Struktur der erstellten Skeletons und über eine Abbildung der Java-Datentypen in XML für die Übertragung mittels des DPWS angestellt werden.

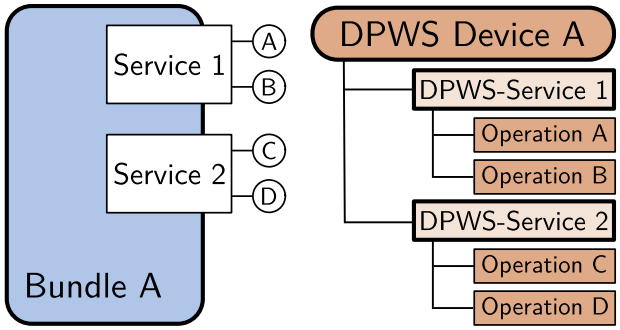


Abbildung 5.6: Abbildung von OSGi-Elementen auf DPWS-Elemente

In den Abschnitten 2.1.3 und 2.1.7 wurden die grundlegenden Strukturelemente von OSGi und DPWS, vorgegeben durch die jeweilige Spezifikation, bereits vorgestellt. Diese werden, wie in Abbildung 5.6 zu sehen, aufeinander abgebildet.

Bundle

Softwarekomponenten in OSGi werden als Bundle bezeichnet. Bundles sind JAR-Dateien, die Ressourcen und zusätzliche Metadaten enthalten. In einem Bundle können eine beliebige Anzahl von Services implementiert werden. Innerhalb des DPWS werden Services in Devices – auch *Hosting Service* genannt – gruppiert. Durch diese enge semantische Entsprechung können die Strukturen auf dieser Ebene leicht aufeinander abgebildet werden. Wird ein OSGi-Service aus Bundle `A` exportiert, so wird ein entsprechendes DPWS-Device `A` erstellt. Metadaten aus dem Bundle werden über die Scopes des Device abgebildet und mit einer eindeutigen Plattform-Identifikation ergänzt. Somit kann auf DPWS-Ebene leicht nach Devices, die bestimmte Eigenschaften eines OSGi-Bundles abbilden, wie z. B. einen bestimmten Hersteller, gesucht werden.

Service

Services in OSGi sind Objekte, die beliebig viele Java-Interfaces implementieren. In DPWS existiert für jeden Service eine WSDL, in der das Service-Element die entsprechende Abbildung für den OSGi-Service darstellt.

Interface

Für die Abbildung der Strukturen auf Ebene der Interfaces gibt es wieder eine geeignete Struktur in der WSDL. PortTypes fassen eine Menge von Operations zusammen, genau wie Java-Interfaces eine Menge von Methoden. Ein PortType wird durch einen vollqualifizierten Namen ausgezeichnet, dargestellt in XML-Schema durch den simplen Datentyp `QName`. Der Namensraum wird durch eine URI dargestellt, gefolgt von einem lokalen Namensteil. Dieses Konzept kann für die Abbildung des ebenfalls vollqualifizierten Namen des Java-Interfaces verwendet werden. Bei dieser Abbildung entsteht der PortType durch die URI `http://www.ws4d.org/<PackageName>`, wobei der Package-Name in eine URI-konforme Darstellung überführt wird, und dem Namen des Java-Interfaces als lokalen Namensraum. Beispiel:

```
1 Java:  edu.udo.lsiv.InterfaceA
2 WSDL:  http://ws4d.org/edu/udo/lsiv/InterfaceA
```

Eine exakte Abbildung der Interface-Strukturen, wie sie für eine mögliche Generierung der Interfaces auf der Client-Seite notwendig ist, ist damit allerdings noch nicht erreicht. Grund ist eine mögliche Vererbungshierarchie in den Service-Interfaces. Abbildung 5.7 zeigt das Problem.

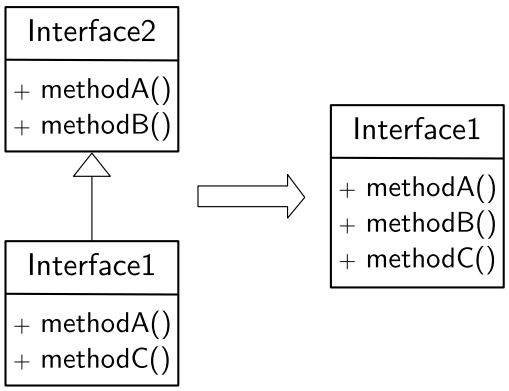


Abbildung 5.7: Problematik der Abbildung von Interfaces mit einer Vererbungshierarchie

Der ursprüngliche OSGi-Service implementiert ein `Interface 1` mit den Methoden `A` und `C`. Dieses Interface erbt von einem `Interface 2` mit den Methoden `A` und `B`. Ohne Informationen zu den Vererbungshierarchien werden alle Methoden zu Operations des PortTypes `Interface 1`, und dementsprechend wird auf der Client-Seite ein `Interface 1` mit den Methoden `A`, `B` und `C` generiert. Eine exakte Abbildung ist nicht erreicht.

Das Problem kann durch eine Erweiterung des WSDL-Dokuments gelöst werden. Unter [Chr01, Section A4.1] ist spezifiziert, wie das PortType-Element innerhalb der WSDL um eigene Attribute erweitert werden kann. Ein Elter-Interface bzw. Elter-PortType wird nun durch die Einführung des Attributs `ii:extends` ausgezeichnet, dessen Inhalt der vollqualifizierte Name des entsprechenden PortType ist. Beispiel:

```
1 ii:extends="edu.udo.lsiv.InterfaceA"
```

Der Namensraum `ii` identifiziert das optionale Attribut als Entität von Comoros und wird von anderen DPWS-Clients ignoriert. Somit besteht die Möglichkeit, jede Operation

genau dem PortType zuzuordnen, der das entsprechende Java-Interface darstellt und so die Hierarchie über Elter-Elemente zu spezifizieren.

Methoden

Java-Methoden bestehen aus einer Menge von Eingabeparametern, genau einem Ausgabeparameter und einer Menge von Fehlerparametern. Diese Struktur findet innerhalb der WSDL in den Operations ihre Entsprechung. Bei der Strukturabbildung erhält die Operation als Namen die Signatur der Java-Methode. Der alleinige Methodenname ist nicht ausreichend, da es mehrere Methoden mit dem selben Namen aber unterschiedlichen Parametern geben kann. Diese müssen folglich unterschieden werden. Die Signatur wird dabei in einer Form und einer Menge an Zeichen dargestellt, welche die XML-Spezifikation nicht verletzt [Bra08, Section 2.3]. Beispiel:

```
1 void_methodA_int_short
```

Als Kinder hat das Operation-Element genau ein Input-Element, ein Output-Element und beliebig viele Fault-Elemente. Die Abbildung von Java-Exceptions auf WSDL-Fault-Elemente scheint naheliegend, ist jedoch mit einigen Hindernissen versehen. Eine Besonderheit des Fault-Elements ist, dass es nur in Verbindung mit den *Message Exchange Pattern (MEP)* `Request-Response` und `Solicit-Response` verwendet werden kann [Chr01]. Java-Methoden können allerdings ohne Ausgabeparameter auskommen, was dem MEP `Notification` entspricht. Für die Abbildung der Java-Exception könnte in diesem Fall nicht auf das Fault-Element zurückgegriffen werden. Andere Ansätze führen aber entweder zu einer asynchronen Fehlerbehandlung – beispielsweise durch Einführung einer zusätzlichen Operation für die Fehlerbenachrichtigung und Abonnieren dieser Operation durch einen Listener auf der Client-Seite – oder einer nicht wohlgeformten Abbildung – z. B. durch die Einführung eines zusätzlichen Output-Parameters, der im Falle eines Fehlers gesetzt wird. Da das Java-Exception-Pattern aber eine synchrone Fehlerbehandlung realisiert, und innerhalb einer wohlgeformten Strukturabbildung Output-Parameter nur für die Abbildung von Java-Ausgabeparametern verwendet werden können, sind diese Ansätze nicht zielführend. Um dieses Problem zu lösen, wird in der WSDL eine leere Nachricht eingeführt.

```
1 <message name="empty"/>
```

Mit Hilfe dieser Definition kann jede Java-Methode ohne Rückgabewert in eine Operation nach dem `Request-Response`-Muster abgebildet werden. Somit kann in jedem Fall das Fault-Element für die Abbildung der Java-Exceptions verwendet werden. Die einzelnen Input-, Output- und Fault-Elemente verweisen nun in der WSDL auf Message-Elemente, in denen die Datentypen spezifiziert werden.

Datentypen

Bei der Umsetzung der verteilten OSGi-Plattform mittels DPWS entsteht ein heterogenes System. Die Übertragung der Daten ist in diesem System eine der größten Herausforderung, da die Konzepte von OSGi und DPWS in diesem Punkt grundlegend verschieden sind. In einem OSGi-Service-Interface können sämtliche Datentypen der Java-API verwendet werden, wohingegen die Datentypen innerhalb einer WSDL in XML-Schema

spezifiziert werden. Die XML-Spezifikation erlaubt zudem beliebige komplexe Datentypen – Arrangements aus simplen Datentypen – zu definieren, die keine Entsprechung in Java haben. Gleichfalls können in Java beliebige Klassen definiert werden, die durch die XML-Spezifikation nicht direkt abgedeckt werden können.

Eine einfache Lösung wäre nun die Java-Datentypen in ein externes Datenformat zu serialisieren – beispielsweise mit Hilfe von XStream [Wal05] oder dem Java Serialisierungsmechnismus [Rig96] – und als Anhang der SOAP-Nachricht zu übertragen. Diese Lösung steht aber im Gegensatz zu den in Kapitel 4 gestellten Anforderungen. Comoros-fremde DPWS-Clients könnten dieses Format nicht lesen und wären als Kommunikationspartner ausgeschlossen. Daher bleibt als Lösung nur die Serialisierung der Java-Datentypen in das vom DPWS verwendete XML-Datenformat. Die RS-Spezifikation legt im Rahmen der als Data Fencing definierten Restriktion eine Menge von Datentypen fest, die für eine entfernte Kommunikation unterstützt werden müssen [The10b]. Tabelle 5.2 listet die Java-Datentypen, die notwendigerweise unterstützt werden müssen und gleichzeitig eine Entsprechung in XML-Schema besitzen. Es gibt Übereinstimmungen bei den primitiven Datentypen und deren Wrapper-Klassen. Komplexe Datentypen, wie Collections oder Arrays, fehlen aber komplett.

Tabelle 5.2: Mapping of Java data types to XML-Schema

primitive type	*Wrapper*	*XML-Schema type*
boolean	java.lang.Boolean	xs:boolean
char	java.lang.Character	-
byte	java.lang.Byte	xs:byte
double	java.lang.Double	xs:double
float	java.lang.Float	xs:float
int	java.lang.Integer	xs:int
long	java.lang.Long	xs:long
short	java.lang.Short	xs:short
-	java.lang.String	xs:string

Im folgenden wird nun erläutert, wie die unterstützten Datentypen in Comoros durch XML-Schema repräsentiert werden.

- **Primitive Datentypen:** Für die meisten primitiven Java-Datentypen finden sich passende simple Datentypen in XML-Schema, so genannte *built-in datatypes*. Der Datentyp `int` wird so beispielsweise über folgende XML-Schema-Repräsentation abgebildet:

```
1 <xsd:element name="int" type="xsd:int"/>
```

  Die weiteren primitiven Datentypen folgen analog. Der einzige primitive Java-Datentyp, für den es keine direkte Entsprechung in XML-Schema gibt, der `char`-Datentyp, kann als Restriktion des Datentypen `String` der Länge 1 modelliert werden:

```
1 <xsd:element name="char">
```

```
 <xsd:simpleType>
  <xsd:restriction base="xsd:string">
   <xsd:lenght value="1"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

- **Null-Pointer:** In Java können Objekt-Referenzen den wert `null` annehmen. Ein solcher Null-Pointer kann in XML-Schema durch die Einführung des optionalen Attribut `xsd:nillable="true"` eingeführt werden. Ein Element mit diesem Attribut kann als leeres Element instantiiert werden, was der Semantik eines Null-Pointers entspricht.
- **Wrapper:** Die Wrapper-Klassen in Java sind Objektrepräsentationen der primitiven Datentypen. Für die Übertragung als XML-Dokument gibt es eigentlich keinen Unterschied zu den simplen Typen, da der gleiche Wert übertragen wird. Allerdings kann eine Objekt-Referenz den Wert `null` annehmen, so dass das Schema-Element das Attribut `xsi:nillable` gesetzt haben muss. Weiterhin ist es für die Rekonstruktion der exakten Signatur notwendig zwischen simplen Datentypen und ihren Wrapper-Klassen zu unterscheiden. Die Wrapper-Klasse `java.lang.Integer` wird wie folgt definiert:

```
<xsd:element name="Integer" type="xsd:int" nillable="true"/>
```

- **Arrays:** Bei der Darstellung eines Arrays bietet sich die Lösung an, einen komplexen XML-Datentyp mit einer unbegrenzten Sequenzlänge zu definieren. Hier sind aber einige Details zu beachten. Ein Array kann grundsätzlich jeden beliebigen Objekt-Typ beinhalten. Da ein Array aber Typ-konsistent ist, kann in der Schema-Beschreibung der naheliegende Typ `xs:any` nicht verwendet werden. Andernfalls wäre es möglich innerhalb einer Sequenz in jedem Element den Typ neu zu bestimmen. Die Typ-Konsistenz wäre somit nicht mehr gegeben. Für die Lösung des Problems, ist es notwendig, alle (nach der RS-Spezifikation) erlaubten Typen zu definieren, und dann in ein `choice`-Element einzubetten. So hat das Element bei der Instantiierung eine Sequenz von Elementen genau eines innerhalb der `choice` definierten Typs.

```
<xs:complexType name="arrayType">
 <xs:choice>
  <xs:sequence>
   <xs:element maxOccurs="unbounded" minOccurs="0" ref="tns:String"/>
  </xs:sequence>
  <xs:sequence>
   <xs:element maxOccurs="unbounded" minOccurs="0" ref="tns:int"/>
  </xs:sequence>
  ...
 </xs:choice>
<xs:attribute name="type" type="xs:string"/>
<xs:attribute name="size" type="xs:int"/>
```

```
13 <xs:attribute name="id" type="xs:int"/>
14 <xs:attribute name="refid" type="xs:int"/>
15 </xs:complexType>
```

Das `size`-Attribut des Wurzelelements kann zur Bestimmung der Größe verwendet werden und erleichtert so die Instantiierung des Java-Array-Objekts im Zuge des Demarshaling. Als Erweiterung gegenüber der RS-Spezifikation unterstützt Comoros direkt auch multidimensionale Arrays. Auf Ebene des Schemas wird dazu einfach innerhalb der `choice` ein weiteres Element hinzugefügt, das als Typ die umgebende Array-Definition referenziert. Dadurch ist es möglich Arrays mit beliebiger Dimension zu erzeugen.

```
1 <xs:complexType name="arrayType">
2  <xs:choice>
3   <xs:sequence>
4    <xs:element maxOccurs="unbounded" minOccurs="0" ref="com:arrayType"/>
5   </xs:sequence>
6   ...
7  </xs:choice>
8 ...
9 </xs:complexType>
```

Durch die Erweiterungen können nun innerhalb eines Arrays Referenzstrukturen auftreten, in denen ein Objekt mehrfach referenziert wird. Für die Abbildung dieser Struktur werden die `id`- und `refid`-Attribute verwendet, die verhindern, dass bei der Rekonstruktion für jedes referenzierte Objekt eine separate Kopie erstellt wird. Abbildung 5.8 verdeutlicht dieses Vorgehen: das Objekt `Array 2` wird dem Objekt `Array 1` an zwei Positionen zugewiesen, die Referenzstrukturen werden entsprechend abgebildet.

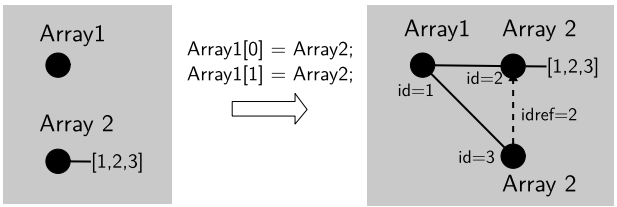


Abbildung 5.8: Abbildung von Referenzstrukturen mittels der `id`- und `refid`-Attribute

Das Einbetten der einzelnen Array-Elemente in XML-Elemente hat aber den Nachteil, dass ein großer Overhead innerhalb einer SOAP-Nachricht erzeugt wird. Für ein `byte`-Array ergibt sich folgendes Beispiel:

```
1 <byteArray>
2  <byte>7</byte>
3  <byte>2</byte>
4  ...
5 </byteArray>
```

Das schlechte Verhältnis zwischen der Anzahl der Zeichen, welche die Nutzdaten beschreiben, und der Zeichen zur Auszeichnung der Daten ist leicht erkennbar. Gerade in Ressourcen beschränkten Umgebungen kann ein solcher Overhead nicht akzeptiert werden. Für Arrays aus primitiven Datentypen und deren Wrapper, also die durch die RS-Spezifikation vorgeschriebene Menge, kann aber eine wesentlich kompaktere Darstellung durch eine Übertragung als Binärdaten über SOAP-Attachments erreicht werden. Dazu sieht die XML-Spezifikation den Datentyp `base64Binary` vor, mit dem das entsprechende Element belegt wird. Zusätzlich zu den reinen Binärdaten werden in der XML-Darstellung Informationen zum Typ des Arrays und zur Array-Länge benötigt, um die entsprechende Java-Instanz erzeugen zu können. Die Größe der einzelnen Elemente ist durch die Spezifikation der JVM vorgegeben [Lin99]. So wird beispielsweise der Datentyp `int` als 32-bit Zweierkomplement mit *big endian* Byte-Reihenfolge definiert. Mit Hilfe dieser Informationen ist es möglich das Array zu Rekonstruieren. Für diese kompakte Darstellung ergibt sich folgendes Schema:

```
1 <complexType name="compactArrayType">
2  <xsd:attribute name="type" type="xsd:string"/>
3  <xsd:attribute name="length" type="xsd:int"/>
4  <element name="data" type="xsd:base64Binary"/>
5 </complexType>
```

Diese Form der Darstellung ist nun zwar deutlich kompakter und effizienter, aber auch schwieriger zu rekonstruieren, da Hintergrundwissen über die Binärdaten benötigt wird.

- **Collections:** Im Gegensatz zu Arrays ist der Inhalt einer `Collection` nicht homogen sondern kann aus verschiedenen Elementen bestehen. Daher kann für diese Datenstruktur keine kompakte Darstellung realisiert werden, jedoch kann nun auf den `xs:any`-Typ zurückgegriffen werden. Das Schema für Collections ist folgendermaßen definiert:

```
1 <xsd:complexType name ="Collection">
2  <xsd:sequence>
3    <xsd:element name="entry" minOccurs="0" maxOccurs="-1" type="xs:any"/>
4  </xsd:sequence>
5 </xsd:complexType>
```

Für Collections wurde eine generische Repräsentation eingeführt. Der tatsächliche Typ wird, dem *Late-Binding*-Konzept folgend, erst zur Laufzeit über das `xsi:type`-Attribut gesetzt. Der Grund für dieses Vorgehen liegt daran, dass Arrays in Comoros, entgegen der RS-Spezifikation auch Collections enthalten können. Somit müsste für jede mögliche konkrete Collection-Klasse eine Schema-Repräsentation bestehen, die WSDL würde unnötig aufgebläht werden. In der DPWS-Umgebung wird die WSDL aber mit jeder `GetResponse`-Nachricht ausgetauscht, eine ineffiziente Lösung wäre die Folge. Durch das Late-Binding-Konzept wird dieses Problem umgangen.

- **Maps:** Maps werden analog zu Collections umgesetzt. Der einzige Unterschied

besteht darin, dass ein Eintrag innerhalb einer Map aus genau zwei Objekten bestehen muss, einem *key* und einem *value*. Aus diesem Grund werden die Elemente innerhalb des `Entry`-Elements um die Attribute `minOccurs=1` und `maxOccurs=1` ergänzt.

```
<xsd:complexType name="Map">
 <xsd:sequence>
  <xsd:element name="mapEntry" minOccurs="0" maxOccurs="-1">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="key" minOccurs="1" maxOccurs="1" type="xs:any"/>
     <xsd:element name="value" minOccurs="1" maxOccurs="1" type="xs:any"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
```

- **Exceptions:** Neben den Datentypen der Eingabe- und Ausgabeparameter müssen auch die Datentypen möglicher Exceptions in XML-Schema beschrieben werden. Wie in Unterabschnitt 5.2.3 erläutert, werden Java-Exceptions in WSDL-Fault-Parameter abgebildet, deren Datentypen auch innerhalb des Types-Elements definiert werden. Um die Komplexität der WSDL möglichst gering zu halten, ist es erforderlich eine möglichst generische Beschreibung der Exceptions zu erstellen. Butek [But08] definiert dabei mehrere Grade der Wiederverwendbarkeit von WSDL-Faults. Für den Fall von Comoros reicht die Wiederverwendbarkeit innerhalb einer WSDL.

  In Java existieren zwei Arten von Fehlern: Declared-Exceptions und Runtime-Exceptions (beispielsweise `NullPointerException`), die zu jeder Zeit auftreten können ohne explizit deklariert worden zu sein. Um die Signatur der zugrunde liegenden Java-Methode rekonstruieren zu können, muss der Typ für deklarierte Exceptions direkt in der WSDL spezifiziert werden, wohingegen der Typ der Runtime-Exceptions erst zur Laufzeit bekannt sein muss. Im Comoros-Core können nur simple Exceptions abgebildet werden, die keine eigenen Klassen-Felder oder zusätzliche Semantik einführen. Die Abbildung komplexer Exception-Klassen in XML-Schema geht über eine generische Darstellung hinaus und widerspricht damit der Core-Architektur. Aus diesem Grund ist die Exception-Nachricht die einzig relevante Information, die im Falle einer deklarierten Exception übertragen werden muss. Undeklarierte Exceptions benötigen noch zusätzlich Informationen über ihren Typ. Die Typinformationen werden im Namen des jeweiligen Exception-Elements durch Angabe des vollqualifizierten Java-Namens abgebildet. Insgesamt sieht eine generische Beschreibung für beide Typen folgendermaßen aus:

```
<message name="tns:NumberFormatException">
 <part name="nfeMessage" type="xsd:string"/>
</message>

```

```
<message name="UndeclaredException">
 <part name="ueMessage"
       type="xsd:undeclaredException"/>
</message>
```

Der Datentyp, der für undeklarierte Exceptions verwendet wird, hat folgende Definition:

```
<xsd:complexType name="undeclaredException">
 <xsd:sequence>
  <xsd:element name="exceptionType" type="xsd:string"/>
  <xsd:element name="message" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Properties

OSGi-Services können mit einer Menge von Properties registriert werden, die ebenfalls auf der Proxy-Seite propagiert werden müssen. Properties eines OSGi-Service sind von der Form `java.util.Dictionary`, einem Datentyp, der laut Oracle vom Datentyp `java.util.Map` abgelöst werden soll. Daher wurde entschieden die Übertragung über den `Map`-Datentyp zu realisieren und die Registrierung mit Hilfe der Adapterklasse `java.util.Hashtable` zu vollziehen, die zu beiden Oberklassen kompatibel ist.

Da OSGi-Properties sich im laufenden Betrieb ständig ändern können, wird die Komplexität der Übertragung weiter erhöht. Zu diesem Zweck wird jedem Skeleton-Service ein spezieller PortType (`PropertyInterface`) mit einer Evented-Operation hinzugefügt, die Änderungen an den Service-Properties propagiert. Dabei wird nicht bei jeder Änderung der komplette Property-Satz verschickt, sondern nur die einzelnen veränderten Properties. Auf der Client-Seite abonniert die Comoros-Distribution-Software die entsprechenden Events und kann die dadurch erhaltenen Informationen zur Registrierung des Proxy-Service nutzen. OSGi-Clients auf dieser Seite können anschließend die Properties abrufen, eine Änderung der Properties auf der Client-Seite ist aber nicht gestattet.

5.2.4 Erstellung des DPWS-Skeletons

Die Erstellung der Skeletons erfolgt in mehreren Schritten. Im ersten Teil wird überprüft ob der Service aus einem Bundle stammt, das schon einen Remote-Service anbietet. In diesem Fall existiert schon ein DPWS-Device, dem der Sekeleton-Service hinzugefügt werden kann. Ist dies nicht der Fall, wird ein neues Device erstellt. Anschließend wird der Skeleton mit den beschriebenen Informations-Abbildungen erstellt, dem Device hinzugefügt und abschließend das Device gestartet. Dieser Vorgang wird im Sinne verteilter Objektsysteme als *Activation* bezeichnet. Das JMEDS-Bundle agiert als Objektadapter, es erhält das erstellte DPWS-Device und versendet die entsprechende `Hello`-Nachricht.

Weiterhin ist es Aufgabe der Comoros-Distribution-Software die OSGi-Umgebung zu überwachen und DPWS-Services oder Devices gegebenenfalls zu entfernen oder neu hinzuzufügen.

5.3 Proxy

Um zu gewährleisten, dass ein lokaler Client einen entfernten OSGi-Service auf die gleiche Weise benutzen kann wie einen lokalen OSGi-Service, wird ein Proxy-Objekt registriert, das eine exakte Abbildung des entfernten Service-Objekts ist. Sobald das Objekt registriert wurde, kann es mittels standardisierter Mechanismen über die lokale Service-Registry gesucht werden. Der Prozess der Proxy-Generierung wird dabei durch eine initiale Benachrichtigung eines DPWS-Skeletons (`Hello`-Nachricht) oder durch eine erfolgreiche Suche (`Probe`-Nachricht mit anschließenden `ProbeMatches`) im Proxy-Generator ausgelöst, der Teil der Comoros Distribution-Software ist. Die Aufgaben des Proxy-Generators sind in Abbildung 5.9 zu sehen.

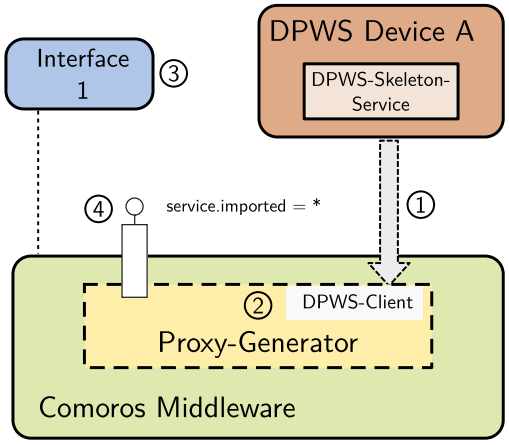


Abbildung 5.9: Prozess der Erstellung eines Proxys durch den Proxy-Generator

Der Proxy-Generator überwacht die DPWS-Umgebung auf relevante DPWS-Skeletons (1). Aus den abgebildeten Informationen rekonstruiert der Proxy-Generator den entfernten OSGi-Service um eine exakte Abbildung zu erstellen (2). Anschließend wird überprüft, ob die benötigten Interfaces im Framework vorhanden sind oder noch bereitgestellt werden müssen (3). Am Ende wird der Proxy-Service im lokalen Framework registriert (4).

5.3.1 Überwachung der Umgebung

Analog zur Überwachung der OSGi-Umgebung durch den Skeleton-Generator, muss der Proxy-Generator die DPWS-Umgebung auf neu hinzukommende, entfernte und modifizierte DPWS-Skeletons überwachen, um anschließend die entsprechenden Proxys zu erstellen oder zu verwalten. Dieser Vorgang umfasst zum einen die Steuerung innerhalb der lokalen OSGi-Umgebung, welche entfernten Services lokal eingebunden werden sollen, und zum anderen die Suche nach entsprechenden Services in der DPWS-Welt.

Suche nach entfernten Services in OSGi

Für die Bereitstellung eines entfernten Service in der lokalen Plattform existieren zwei verschiedene Varianten:

1. **Anfragengesteuerte Bereitstellung:** Bei der anfragengesteuerten Bereitstellung von Proxys werden diese erst erzeugt, wenn eine Anfrage eines lokalen OSGi-Clients

für einen entfernten Service existiert, z. B. durch das Öffnen eines `ServiceTrackers`. Für die Umsetzung dieses Szenarios bietet sich die Benutzung des in Version R4.2 der OSGi-Spezifikation eingeführten Hook-Prinzips [The10b] an. Der an dieser Stelle relevante `ListenerHook` wird von der Comoros-Distribution-Software im Framework registriert, und wird fortan über alle Suchanfragen lokaler Clients informiert. In diesem Prozess empfängt der `ListenerHook` den kompletten `LDAP`-Filterausdruck der lokalen Service-Suche eines OSGi-Clients. Dieser Filterausdruck wird vom Proxy-Generator analysiert, und in eine DPWS-Suchanfrage für entfernte DPWS-Skeletons überführt, die dann vom integrierten DPWS-Client abgeschickt wird. Bei einer erfolgreichen Anfrage wird schlussendlich ein Proxy erzeugt und registriert. Wird direkt kein passender Skeleton gefunden, so wird die Umgebung weiterhin auf neu erscheinende, passende Skeletons überwacht.

Der Proxy ist fortan an den Lebenszyklus der Suchanfrage gebunden. Wird der ServiceTracker, der die Bereitstellung ausgelöst hat, geschlossen, so wird auch diese Information vom ListenerHook abgefangen und die Überwachung der Umgebung auf diesen Service beendet. Bereits registrierte Proxys werden, falls keine Bindungen zu lokalen Clients mehr bestehen oder sich andere Suchanfragen auf den Proxy beziehen, anschließend deregistriert.

2. **Konfigurationsgesteuerte Bereitstellung:** Die anfragengesteuerte Bereitstellung der Proxys hat den Nachteil, dass der initiale Aufruf eines entfernten Service durch den zeit- und ressourcenintensiven Erzeugungsvorgang (DPWS-Suchanfrage, Proxy-Generierung etc.) einer überdurchschnittlichen Verzögerung unterworfen ist. Dies kann gerade in Ressourcen beschränkten Umgebungen problematisch sein. Aus diesem Grund wird eine konfigurationsgesteuerte Bereitstellung eingeführt, bei der, durch eine passende Konfiguration, Proxys schon vor dem ersten Aufruf erzeugt werden. Diese, in der Distribution-Software hinterlegte Konfiguration, steuert auf der Client-Seite, welche entfernten Services für lokale Zugriffe in die lokale Plattform eingebunden werden. Die Konfigurationen umfassen im wesentlichen drei Informationsteile:

   - **platformId:** Systemweit eindeutige Kennzeichnung einer OSGi-Platform.
   - **bundle:** Symbolischer Name des Bundles, das den Remote Service anbietet.
   - **interface:** Vollqualifizierter Klassenname des Service-Interfaces, das entfernt zugreifbar sein soll, beziehungsweise für das ein Proxy generiert werden soll.

   Die drei Positionen ergeben eine logische `UND`-Verknüpfung dessen Interpretation die Erstellung von Proxys auf der Client-Seite anstösst. Dabei kann jede der drei Positionen mit einer Wildcard (`*`) belegt werden, oder innerhalb einzelner Positionen die Wildcard als Ergänzung im Sinne eines `LDAP`-Filterausdrucks verwendet werden. Beispiel:

```
1 platform1/edu.udo.lsiv.MathBundle/Math*
```

   Diese Konfiguration bedingt das Importieren jedes Service der OSGi-Plattform

`platform1`, aus dem Bundle mit dem Namen `edu.udo.lsiv.MathBundle`, deren Interface-Namen mit `Math` beginnen.

Die konfigurationsgesteuerte Bereitstellung bietet somit die Möglichkeit ein Kommunikationslayout für das verteilte System zu erstellen, in dem die Zugriffsmuster im wesentlichen festgelegt sind.

Suche nach entfernten Services in DPWS

Sowohl bei der anfragen- als auch bei der konfigurationsgesteuerten Bereitstellung müssen passende entfernte Services gesucht werden. Im Sinne der Nutzertranzparenz (siehe Unterabschnitt 4.3.2) werden bei der anfragengesteuerten Variante die durch den Nutzer abeschickten lokalen OSGi-Suchanfragen transparent auf DPWS-Mechanismen abgebildet. Die gleichen Mechanismen werden für die Abbildung von hinterlegten Konfigurationen auf entsprechende Suchanfragen verwendet. Dabei ergeben sich die folgenden Möglichkeiten zur DPWS-basierten Service-Suche:

1. **Allgemeine Servicesuche:** Die DPWS-Spezifikation sieht eine direkte Suche nach Services nicht vor [Nix09b], stattdessen sind Suchanfragen immer an Devices gerichtet. Aus dieser Einschränkung resultiert ein komplexerer Vorgang um solche Skeletons zu finden, die zur OSGi-Servicesuche passende Services anbieten. Dieser Vorgang ist in Abbildung 5.10 zu sehen.

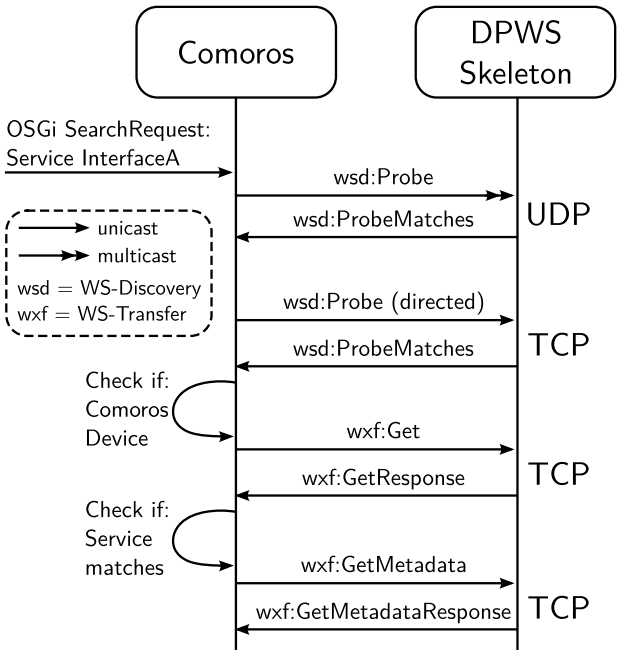


Abbildung 5.10: Prozess einer allgemeinen Servicesuche

Ausgelöst durch die OSGi-Suchanfrage wird eine leere `Probe`-Nachricht mittels UDP-Multicast im LAN versendet. `Probe`-Nachrichten können durch Angabe von Suchparametern spezialisiert werden, allerdings betreffen diese Parameter nur die Devices, und nicht die gehosteten Services. Skeletons, die durch Comoros erzeugt

werden, haben allerdings einen gesetzten Scope, der die Devices als Comoros Skeletons auszeichnet (`ComorosSkeletonDevice`). Diese Information darf aber nicht als Suchparameter benutzt werden um die spätere Integration von Comoros-fremden DPWS-Devices nicht zu verhindern. Auf eine allgemeine `Probe`-Nachricht antworten schließlich alle aktiven Devices im LAN mit einer `ProbeMatches`-Nachricht. Die WS-Discovery-Spezifikation schreibt hierbei vor, dass diese Nachricht eine einzelne UDP-Nachricht ist [Bea09]. UDP-Nachrichten sind in ihrer Größe insgesamt auf 65.535 Bytes beschränkt [Pos80], so dass nicht gewährleistet werden kann, dass sämtliche Scopes eines Devices in dieser Nachricht enthalten sind. Deshalb wird im Anschluss an ein `ProbeMatches` eine gerichtete `Probe`-Nachricht über TCP an das Device versendet, die ebenfalls über TCP empfangene `ProbeMatches`-Nachricht enthält nun alle Scopes des Devices. Nun können alle Devices, die keine durch Comoros generierten Skeletons darstellen, ausgefiltert werden. Um nun an die Informationen über die gehosteten Services zu gelangen wird das Device mittels einer `Get`-Anfrage und anschließend empfangener `GetResponse`-Antwort aufgelöst. Jetzt kann Comoros durch eine Analyse der PortTypes eines jeden Services entscheiden, ob dieser Service zur ursprünglichen OSGi-Suchanfrage passt. Falls ja, kann die komplette Servicebeschreibung (inklusive WSDL-Datei) über eine `GetMetadata`-Nachricht angefordert und der Proxy erstellt werden.

Bei der konfigurationsgesteuerten Bereitstellung der Proxys kann, falls der `bundle`-Teil des Filterausdrucks gesetzt ist, die `Probe`-Nachricht um einen Suchparameter, der den Inhalt dieses Teils enthält, ergänzt werden. Nun antworten nur darauf passende Devices mit einem `ProbeMatches`, auf die eigentlich anschließende gerichtete `Probe`-Nachricht kann nun verzichtet werden.

2. **Scope-basierte Servicesuche:** Eine einfache Erweiterung der allgemeinen Service-Suche kann die Anzahl der ausgetauschten Nachrichten deutlich reduzieren. Hierzu wird eine zielgerichtete Suche über die kategorisierenden Scopes eines Devices eingeführt. Ein Skeleton erhält nun jedes registrierte Interface des abgebildeten OSGi-Services als Scope. Als Folge kann nun bei einer eingehenden OSGi-Suchanfrage die DPWS-`Probe`-Nachricht mit dem Interface dieser Anfrage spezialisiert werden. Daraufhin antworten nur die Devices, die einen passenden Service gehostet haben.

   Eine Analyse der versendeten Nachrichten für beide Ansätze zeigt den Vorteil der Scope-basierten Suche. Abbildung 5.11 zeigt ein beispielhaftes Netzwerk mit dem Comoros-Client und fünf DPWS-Devices.

   Die Gesamtanzahl von Devices (n) kann in drei Kategorien unterteilt werden: Comoros-Skeletons mit einem zur Suchanfrage passenden Service, Comoros-Skeletons mit keinem passenden Service (o) und Comoros-fremde DPWS-Devices (m). Alle Devices haben eine unterschiedliche Menge Services ($x \in \mathbb{N}_0$).

   Die initiale Probe-Nachricht ist nicht Teil der Analyse, da an dieser Stelle nur Unicast-Nachrichten analysiert werden, und die Anzahl der `Probe`-Nachrichten in allen Ansätzen identisch ist. Die versendeten Nachrichten bis zur Generierung des Proxys setzen sich aus folgenden Teilen zusammen:

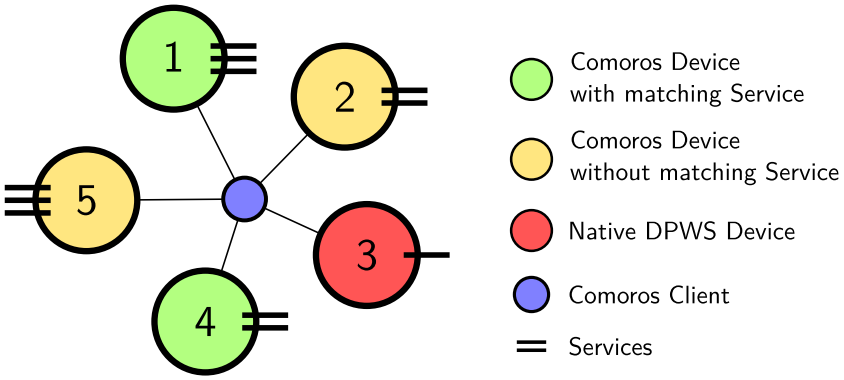


Abbildung 5.11: Beispielhaftes Netzwerk mit einem Comoros-Client als Wurzel und verschiedenen DPWS-Devices, die jeweils eine unterschiedliche Anzahl an Services hosten

a) Allgemeine Servicesuche:

ProbeMatches + DirectedProbe + ProbeMatches
für alle Knoten

$$3n \tag{5.5}$$

Get + GetResponse
für alle Comoros Knoten

$$2(n-m) \tag{5.6}$$

GetMetadata + GetMetadataResponse
für alle Knoten mit passendem Service für jeden passenden Service

$$2 * \sum_{i=1}^{n-m-o} x_i \tag{5.7}$$

Invoke + InvokeResponse (retrieve Properties)
für alle kompatiblen und passenden Services

$$2 * \sum_{i=1}^{n-m-o} x_i \tag{5.8}$$

Insgesamt: $3n + 2(n-m) + 4 * \sum_{i=1}^{n-m-o} x_i$

b) Scope-basierte Servicesuche:

ProbeMatches

für alle Knoten, die einen passenden Service enthalten

$$n - m - o \tag{5.9}$$

Get + GetResponse
für alle Knoten, die einen passenden Service enthalten

$$2(n - m - o) \tag{5.10}$$

Nachrichten definiert in 5.7 und 5.8

Insgesamt: $3(n - m - o) + 4 * \sum_{i=1}^{n-m-o} x_i$

Beim Vergleich der beiden Varianten müssen die Teile 5.7 und 5.8 nicht betrachtet werden, da sie für beide Varianten identisch sind. Also ergibt sich für das Beispiel, das in Abbildung 5.11 definiert wurde, für die allgemeine Servicesuche eine Nachrichtenanzahl von $3 * 5 + 2 * 4 = 23$ und für die Scope-basierte Suche lediglich eine Nachrichtenanzahl von $3 * 2 = 6$. Im Worst-Case werden für die erste Variante (alle Knoten sind Comoros-Skeletons) $5n$ Nachrichten ausgetauscht und für die Zweite (alle Knoten sind Comoros-Skeletons mit passenden Services) $3n$.

Für ein eindeutiges Fazit muss zusätzlich zur Anzahl der ausgetauschten Nachrichten noch die Nachrichtengröße betrachtet werden. Die bisher betrachteten Nachrichten, die in beiden Varianten versendet werden (`ProbeMatches`, `Get`, `GetResponse`), unterscheiden sich nicht in ihrer Größe. Einzig die bisher noch nicht betrachteten initialen `Probe`-Multicast-Nachrichten haben durch die Angabe von Suchparametern unterschiedliche Größen. Im Detail nimmt die Nachrichtengröße pro Zeichen für das in der OSGi-Suchanfrage angegebene Interface um 1 Byte zu. Dieser geringe Zuwachs steht in keinem Verhältnis zu den eingesparten Nachrichten, so dass die zweite Variante nachweislich effizienter ist.

Für die konfigurationsgesteuerte Bereitstellung kann dieser Vorteil nicht vollständig umgesetzt werden. Der vorliegende Filterausdruck steuert die Spezialisierung der `Probe`-Nachricht. Ist der `bundle`-Teil gesetzt, können `Probe`-Nachrichten entsprechend angepasst werden und die Anzahl der antwortenden Geräte wird eingeschränkt. Um eine weitere Einschränkung auf Service-Ebene zu erreichen, muss die beschriebene Scope-basierte Suche umgesetzt werden. Allerdings ist das nur möglich, wenn der Service-Teil der Suchanfrage vollständig, d. h. ohne Angabe von Wildcards, gesetzt ist. In einem Netzwerk, in dem alle DPWS-Devices von Comoros erstellt wurden, d. h. auf das Versenden von gerichteten `Probe`-Nachrichten zum vollständigen Abruf der Scopes kann verzichtet werden, müssen nun folgende vier Fälle betrachtet werden:

a) **Bundle-Teil nicht gesetzt und Service-Teil nicht gesetzt:**
Da der Service-Teil der Konfiguration nicht gesetzt ist, kann der Vorteil der

Scope-basierten Suche nicht ausgespielt werden. Spezialisierte Suchanfragen sind nicht möglich, so dass sich die Scope-basierte Suche von der allgemeinen Suche nicht unterscheidet. In beiden Fällen werden n (`ProbeMatches`) + $2n$ (`Get + GetResponse`) = $3n$ Nachrichten bis zum Abruf der Metadaten versendet.

b) **Bundle-Teil gesetzt und Service-Teil gesetzt:**
Für die Analyse dieses Falles wird eine neue Menge q eingeführt, welche der Menge der Knoten im Netzwerk entspricht, die den Filterausdruck im Bundle-Teil nicht erfüllen. Bei der allgemeinen Servicesuche werden nun $(n-q)$ `ProbeMatches` und $2(n-q)$ `Get-` und `GetResponse`-Nachrichten versendet. Insgesamt also $3(n-q)$ Nachrichten. Die Scope-basierte Suche kann den gesetzten Service-Teil ausnutzen und kommt so mit insgesamt $3(n-o-q)$ Nachrichten aus (siehe Gleichung 5.9 und Gleichung 5.10).

Grundsätzlich gilt die Aussage:

$$(n-o-q) \leq (n-q) \tag{5.11}$$

Für den Fall, dass jedes Bundle, welches dem Filterausdruck im Bundle-Teil entspricht, auch einen Service enthält, welcher dem Filterausdruck im Service-Teil entspricht, gilt:

$$(n-o-q) = (n-q) \tag{5.12}$$

Dann unterscheiden sich die beiden Suchanfragen nicht. Andernfalls entsteht für jedes betroffene Bundle im der allgemeinen Servicesuche gegenüber der Scope-basierten Suche ein Overhead von 3 Nachrichten (`ProbeMatches + Get + GetResponse`).

c) **Bundle-Teil gesetzt und Service-Teil nicht gesetzt:** In beiden Varianten ergeben sich $3(n-q)$ Nachrichten, da der Service-Teil nicht gesetzt ist.

d) **Bundle-Teil nicht gesetzt und Service-Teil gesetzt:** Ist der Service-Teil gesetzt und der Bundle-Teil nicht, so unterscheidet sich die Suche innerhalb der konfigurationsgesteuerten Bereitstellung nicht von der in der anfragengesteuerten Bereitstellung. Für die allgemeine Suche werden also $5n$ Nachrichten und in der Scope-basierten Suche nur $3n$ Nachrichten versendet.

Es ist erkennbar, dass sich durch die Informationen im Bundle-Teil in der konfigurationsgesteuerten Bereitstellung die allgemeine Suche und Scope-basierte Suche annähern. Allerdings ist die Scope-basierte Suche noch immer effizienter, da sie bei gesetztem Service-Teil weniger Nachrichten benötigt.

3. **LDAP-basierte Servicesuche:** Eine Service-Suche innerhalb von OSGi wird zumeist nicht über eine einfache Suche nach Interfaces angestoßen sondern über den wesentlich mächtigeren LDAP-Filter-Mechanismus. Über LDAP-Filter können beliebig komplexe Suchen definiert werden, die sich auf die Properties eines OSGi-

Service beziehen. Bei einer solchen Suche wird mit den vorher vorgestellten DPWS-Suchmechanismen die Anzahl der ausgetauschten Nachrichten extrem erhöht, da für jeden Service die registrierten Properties abgefragt werden müssen. Bis zur Entscheidung, ob ein Skeleton passende Services enthält, ergibt sich die folgende Anzahl:

ProbeMatches
für alle nicht nativen Knoten

$$n - m \tag{5.13}$$

Get + GetResponse
für alle nicht nativen Knoten

$$2(n - m) \tag{5.14}$$

GetMetadata + GetMetadataResponse
für jeden Service auf jedem nicht nativen Knoten

$$2 * \sum_{i=1}^{n-m} x_i \tag{5.15}$$

Invoke + InvokeResponse (retrieve Properties)
für jeden Service auf jedem nicht nativen Knoten

$$2 * \sum_{i=1}^{n-m} x_i \tag{5.16}$$

Overall: $3(n - m) + 4 * \sum_{i=1}^{n-m} x_i$

Im Beispiel 5.11 ergibt sich also eine Anzahl von $3 * 4 + 4 * (2 + 2 + 3 + 3) = 52$ Nachrichten bis zur Entscheidung ob ein passender Service existiert (zum Vergleich: $3 * 5 + 2 * 4 + 4 * (3 + 2) = 43$ beziehungsweise $3 * 2 + 4 * (3 + 2) = 26$ Nachrichten bei einfacher Suche über Interfaces mit den vorgestellten Varianten).

Um dieses Verfahren zu optimieren wurde ein Hook-Mechanismus für DPWS-Suchanfragen konzipiert und als Feature in die DPWS-Implementierung JMEDS eingefügt. Über diesen Mechanismus können nun in den Devices Suchanfragen abgefangen und lokal bearbeitet werden. Abbildung 5.12 zeigt, wie mit Hilfe dieses Mechanismus die Nachrichtenanzahl deutlich reduziert werden kann.

Aus der in Comoros abgefangenen LDAP-basierten OSGi-Suchanfrage wird der LDAP-Filter extrahiert und dieser der DPWS-`Probe`-Nachricht angehängt. Der LDAP-Filter ist dabei nur eine Nutzlast, die `Probe`-Nachricht wurde nicht spezialisiert, so dass weiterhin alle Devices in einem LAN angesprochen werden. In

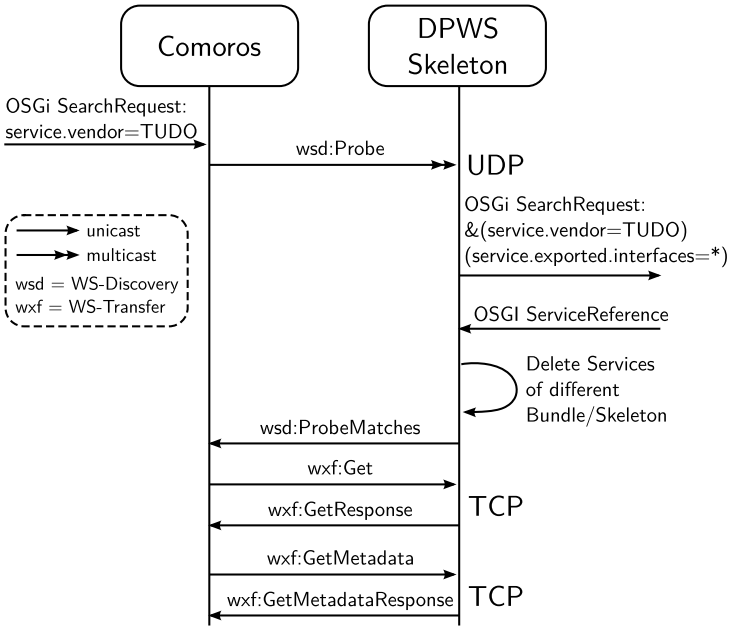


Abbildung 5.12: Prozess einer LDAP-basierten Servicesuche

jedem Device, das diese `Probe`-Nachricht empfängt wird nun der LDAP-Filter aus der Nachricht innerhalb des entwickelten JMEDS-Hook-Mechanismus entnommen und in eine lokale OSGi-Suchanfrage verpackt. Der Filter wird zudem um die Bedingung `service.exported.interfaces=*` ergänzt, um sicherzustellen, dass nur für eine entfernte Nutzung vorgesehene Services antworten. Werden im lokalen Framework passende Services gefunden, so werden im nächsten Schritt alle Services aus den Bundles, die nicht durch den Skeleton, der die OSGi-Suchanfrage initiiert hat, repräsentiert werden, aus der Ergebnismenge ausgefiltert. Ist die Ergebnismenge nicht leer, antwortet das Device mit einem entsprechenden `ProbeMatches`. Mittels `Get` und `GetResponse` werden vom Device anschließend die gehosteten Services abgefragt und es wird mit dem normalen Vorgang zur Proxy-Erzeugung fortgefahren.

Die Nachrichtenanzahl beträgt nun genau wie bei der Scope-basierten Suche $3(n-m-o)$, also für das Beispiel 5.11: $3*2=6$ Nachrichten (`ProbeMatches, Get, GetResponse` von allen Skeletons mit passenden Services). Allerdings kann nun mit der gleichen Anzahl Nachrichten ein wesentlich mächtigerer Suchmechanismus verwendet werden. An dieser Stelle stellt sich die Frage, warum nicht jede Suchanfrage mittels dieses Mechanismus umgesetzt wird. Schließlich ist eine einfache Suche nach Interfaces auch nur eine spezielle LDAP-Suchanfrage (`objectClass=InterfaceA`). Der Grund liegt in der lokalen OSGi-Suchanfrage innerhalb des Hook-Mechanismus der Devices. Hierdurch entsteht eine Verzögerung, die den Ablauf gegenüber der Scope-basierten Suche ineffizienter werden lässt.

Für die konfigurationsgesteuerte Bereitstellung der Services ist diese Art der Suche

aber zu bevorzugen. Durch die mögliche Angabe von Wildcards in den einzelnen Teilen der Konfiguration können `Probe`-Nachrichten nur in wenigen Fällen spezialisiert werden. Der Hook-Mechanismus in den Devices ermöglicht aber die einfache Analyse der in der Suchanfrage übermittelten Konfiguration, genau wie die Analyse eines LDAP-Filters.

4. **Benachrichtigung über passende Services:** Nicht jede Suchanfrage liefert direkt ein passendes Ergebnis. In einer dynamischen Umgebung kann es passieren, dass zunächst kein passender Remote Service zu einer Suche existiert, dieser aber nach einiger Zeit im System aktiviert wird. Dieser Service wird dann der Suchanfrage zugeordnet und ein Proxy wird erstellt. Abbildung 5.13 zeigt den Vorgang für diesen Anwendungsfall.

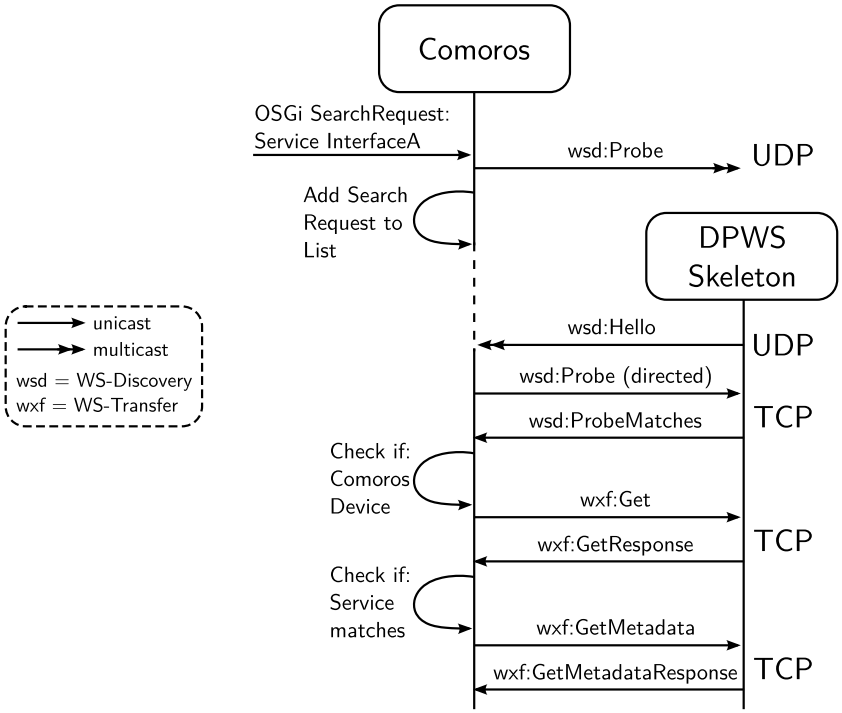


Abbildung 5.13: Prozess der Notifizierung über neu installierte Services, die zu einer zuvor abgesendeten Suchanfrage passen

Zu der abgeschickten `Probe`-Nachricht erhält Comoros kein `ProbeMatches`. Damit Services nachträglich der Suche zugeordnet werden können, und somit das Verhalten einer lokalen OSGi-Suche exakt abgebildet wird, müssen Suchanfragen intern verwaltet werden. Neu installierte oder veränderte DPWS-Devices senden eine `Hello`-Nachricht per Multicast an alle Knoten in einem LAN (die zu der selben Multicast-Gruppe gehören), um die Änderungen bekannt zu machen. Diese `Hello`-Nachricht wird von Comoros empfangen und das betreffende Device als potentieller Kandidat für die intern verwalteten, bereits abgeschlossenen OSGi-Suchanfragen behandelt. Von hier an gleicht der Vorgang der allgemeinen Service-Suche. Es wird eine gerichtet `Probe`-Nachricht an das Device versendet um alle Scopes abzurufen.

Bei einem passenden Scope wird das Device anschließend mittels einer `Get`-Anfrage aufgelöst. Nach dem Abfrage der Servicebeschreibungen (`GetMetadata`-Nachricht) kann der Proxy erstellt werden. Betrifft die `Hello`-Nachricht nur eine Änderung der Metadaten eines bereits in Comoros integrierten Device, müssen die Änderungen überprüft werden und gegebenenfalls Proxys neu installiert oder entfernt werden.

Betrachtet man die Menge an neu hinzugefügten oder veränderten Geräte p, so ergibt sich für die Anzahl an ausgetauschten Nachrichten:

Hello + directed Probe + ProbeMatches
für alle neuen oder modifizierten Knoten

$$3p \tag{5.17}$$

Get + GetResponse
für alle modifizierten oder neuen Comoros-Knoten

$$2(p - (m \in p)) \tag{5.18}$$

GetMetadata + GetMetadataResponse
für jeden passenden Service eines jeden Knoten

$$2 * \sum_{i=1}^{p-(m\in p)-(o\in p)} x_i \tag{5.19}$$

Invoke + InvokeResponse (retrieve Properties)
für alle kompatiblen und passenden Services

$$2 * \sum_{i=1}^{p-(m\in p)-(o\in p)} x_i \tag{5.20}$$

Overall: $3p + 2(p - m) + 4 * \sum_{i=1}^{p-(m\in p)-(o\in p)} x_i$

Für diesen Anwendungsfall muss sich das Beispiel 5.11 zur Laufzeit verändern. In den Knoten 2 und 3 werden Services installiert, die zu einer zuvor erfolglosen Suchanfrage passen (siehe Abbildung 5.14). So ergibt sich ein Austausch von insgesamt $3 * 2 + 2 * 1 + 4 * (3 + 2) = 28$ Nachrichten. Falls, wie bei der Scope-basierten Suche vorgeschrieben, die Services jeweils innerhalb der Scopes der Devices ausgezeichnet sind, so reduziert sich die Nachrichtenanzahl in Gleichung 5.18 zu $2(p - (m \in p) - (o \in p))$ Nachrichten.

Abschließend bleibt anzumerken, dass jede OSGi-Suchanfrage in Comoros intern verwaltet wird um vier Anwendungsfälle abzudecken. Der erste Anwendungsfall wurde im vorherigen Abschnitt beschrieben und betrifft zunächst erfolglose Suchanfragen die durch neu hinzukommende Geräte erfüllt werden können. Der zweite Anwendungsfall betrifft

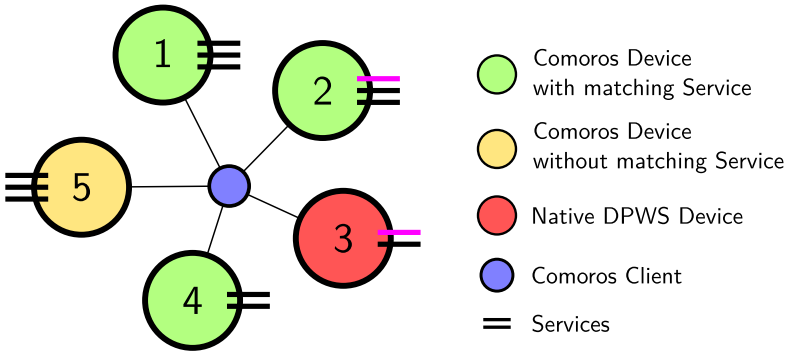


Abbildung 5.14: Zur Laufzeit verändertes Netzwerk im Vergleich zu dem Netzwerk aus Abbildung 5.11

deinstallierte DPWS-Geräte. Diese senden eine `Bye`-Nachricht an alle Knoten des LAN und können in Comoros das Deinstallieren von Proxys auslösen. Aus veränderten DPWS-Geräten (z. B. Scopes verändert, Services hinzugefügt oder entfernt) kann wiederum das Erstellen oder Deinstallieren von Proxys resultieren. Der letzte Anwendungsfall betrifft das Schließen eines OSGi-Service-Tracker, also das Beenden der Suchanfrage. In diesem Fall müssen alle Proxys, die keine Bindung zu anderen Suchanfragen haben, deinstalliert werden.

5.3.2 Proxy-Erzeugung

Resultiert aus der Überwachung der Umgebung die Bereitstellung eines OSGi-Proxy, so muss dieser zuerst in passender Form erstellt werden. Wie in Unterabschnitt 5.2.3 beschrieben sind in den Comoros-Skeletons alle Informationen kodiert um eine exakte Abbildung des ursprünglichen Service im Proxy-Service zu realisieren. Die Informationen werden ausgelesen, ausgewertet und anschließend für die Erstellung des Proxys verwendet. Je nach Anwendungsfall variieren dabei die Anforderungen an den Erstellungsprozess und an die bereitgestellten Proxys. In hoch dynamischen Umgebungen ist eine schnelle und dynamische Erzeugung der Proxys notwendig, wohingegen in Umgebungen mit hohen Kommunikationsaufkommen und wiederkehrenden Zugriffsmustern effiziente Methodenaufrufe priorisiert werden. Aus diesem Grund werden innerhalb von Comoros zwei Varianten für die Erstellung der Proxys umgesetzt.

Proxy-Bundle

Dieses Konzept sieht vor, dass zur Laufzeit statische Proxys mittels Java-Bytecode-Generierung erzeugt und registriert werden. Der Ansatz zur Bytecode-Generierung wurde erstmals in [Rel07c] vorgestellt und für Comoros in [Doh09] initial eingeführt. In Comoros wird so die Registrierung der Proxy-Services aus der Distribution-Software in die eigens für diesen Zweck generierten Proxy-Bundles ausgelagert.

Für diese Proxy-Bundles gibt es aber eine wichtige Einschränkung. Es ist offensichtlich, dass die Proxy-Bundles eine Abbildung der Bundles auf der Service-Seite darstellen. Diese Abbildung kann aber aus unterschiedlichen Gründen nicht realisiert werden. Services innerhalb eines Bundles können in OSGi zur Laufzeit registriert und deregistriert werden.

Existiert nun ein Bundle `B_A` mit den registrierten Services `S_1` und `S_2`, so würde auf der Client-Seite ein Proxy-Bundle `B_A'` mit den Services `S_1'` und `S_2'` erzeugt werden. Wird nun im original Bundle Service `S_2` entfernt und Service `S_3` registriert, so muss diese Änderung auf das Proxy-Bundle übertragen werden.

Die statische Struktur des generierten Bundles erlaubt eine solche Änderung aber nicht. Als Folge müsste das Bundle entfernt werden und als Bundle `B_A'` mit den Services `S_1'` und `S_3'` neu generiert werden. Diese Änderung würde aber die Nutzer von Service `S_1'` beeinträchtigen, da der Service kurzzeitig nicht zur Verfügung steht. Die Seiteneffekte können nicht abgeschätzt werden, so dass diese Lösung nicht realisierbar ist. Abbildung 5.15 zeigt, wie das Problem in Comoros gelöst ist.

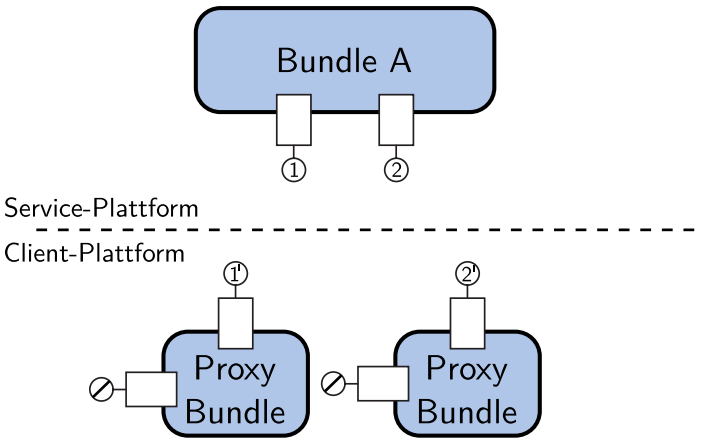


Abbildung 5.15: Aufteilung der Proxy-Bundles um unnötige Neuerzeugungen im Falle von Laufzeit-Modifikationen des originalen Bundles zu vermeiden

Jeder Service ist in einem eigenen Proxy-Bundle verpackt. Diese Aufteilung ist möglich, da OSGi eine Service-orientierte Architektur umsetzt und lediglich eine Suche nach Services sinnvoll ist. Die Service-Sicht ist durch die Aufteilung aber nicht beeinflusst. Wird nun die oben beschriebene Änderung am originalen Service durchgeführt, so wird auf der Client-Seite lediglich das Bundle mit Service `S_2'` entfernt und ein neues Bundle mit Service `S_3'` erzeugt.

Neben dem Registieren und dem Deregistrieren von Services im Zuge der Laufzeit-Modifikationen eines OSGi-Services können die Modifikationen zusätzlich auch die Properties des OSGi-Service betreffen. Eine solche Änderung ist allerdings unkritisch, da die Distribution-Software über eine definierte Service-Schnittstelle in jedem Proxy-Bundle die Properties der registrierten Services neu setzen kann. Damit fremde Clients die Properties nicht modifizieren, ist diese Schnittstelle über den Java-Security-Mechanismus [Oak98] abgeschirmt und kann nur von der DSW aufgerufen werden.

Der Aufbau eines Proxy-Bundles ist durch die Integration eines DPWS-Clients in jedem einzelnen Bundle gekennzeichnet. Dieser Client ist fest an den zugehörigen DPWS-Skeleton gebunden. Über den Client werden OSGi-Serviceaufrufe an den Skeleton-Service weitergegeben und die Rückgabe an den aufrufenden OSGi-Client zurückgegeben. Weiterhin sind im Bundle alle notwendigen Funktionalitäten des Marshaling integriert. Insgesamt beste-

hen für jedes Proxy-Bundle Package-Abhängigkeiten zu Comoros (Distribution-Software, Marshaling-Bundle, DPWS-Bundle) und zu den Service-Interfaces. Die Proxy-Bundles sind an den Lebenszyklus des DPWS-Skeleton-Service und damit transitiv an den Lebenszyklus des originalen OSGi-Service gebunden.

Wie einleitend erwähnt, eignet sich diese Variante vor allem für Systeme mit hohem Kommunikationsaufkommen und festen Zugriffsmustern. Jeder Service-Aufruf an einen Proxy-Service kann durch den individuell erstellten statischen Code im Proxy-Bundle effizient bearbeitet werden. Die Bytecode-Generierung führt allerdings zu einem erhöhten Zeit- und Ressourcenbedarf. Dieser Bedarf verstärkt sich bei zunehmender Dynamik der Plattform. Neu hinzukommende, deinstallierte und veränderte Services führen direkt zu einer Deinstallation der entsprechenden Proxy-Bundles und zu einer erneuten Bytecode-Generierung. Gerade bei der anfragengesteuerten Bereitstellung von Proxy-Services kann es zu einer merklichen Verzögerung bis zur ersten Nutzung kommen. Für die konfigurationsgesteuerte Bereitstellung ist diese Verzögerung von geringerer Relevanz, da Proxys schon vor der ersten Anfrage erstellt werden. Das Problem sich ändernder Services ist für beide Bereitstellungsarten gleichermaßen relevant. Insgesamt kann dieser Ansatz bei eingebetteten Systemen mit hoher Dynamik also zu ineffizienten Ergebnissen führen.

Dynamische Proxys

Beim Konzept der dynamischen Proxys wird die Registrierung der Proxy-Services, im Gegensatz zum Ansatz der Proxy-Bundles, nicht in separate Bundles ausgelagert, sondern verbleibt in der Distribution-Software. Wird für die Proxy-Bundles statischer Bytecode erzeugt, der für jeden Service speziell abgestimmt ist, so verfolgen die Dynamischen-Proxys einen generischen Ansatz. Dynamische-Proxys sind Teil der Standard-Java-Bibliothek und bieten dem Entwickler die Möglichkeit, generische Module mit Hilfe eines *InvocationHandlers* zu erzeugen. Abbildung 5.16 demonstriert dessen Arbeitsweise.

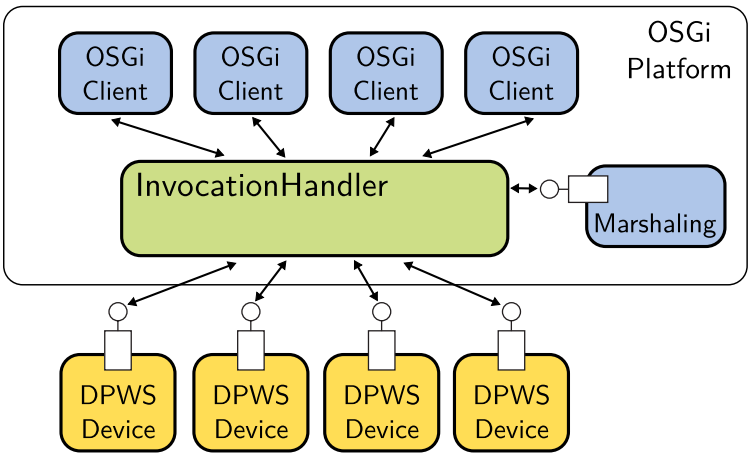


Abbildung 5.16: Arbeitsweise des InvocationHandler innerhalb der durch Comoros erstellten Dynamischen Proxys

Dieser *InvocationHandler* wird bei der Instantiierung des Dynamischen-Proxys re-

ferenziert und verwaltet alle eingehenden Aufrufe des Proxy-Service. Dabei wird der *InvocationHandler* nicht individuell für jeden Proxy angepasst, sondern deckt generisch alle Eventualitäten der eingehenden Aufrufe ab. Innerhalb wird sowohl das Marshalling über den Aufruf der entsprechenden Marshaling-Services realisiert, als auch werden die konkreten DPWS-Services aufgerufen. Dieses erfolgt mittels der Java-Reflection-API [For04]. Der Dynamische-Proxy wird nun bei Bedarf von der Distribution-Software instantiiert und als Service registriert. Für die Registrierung der Dynamische-Proxys werden neben dem *InvocationHandler* das Service-Interface und der Classloader, mit dem die Interfaces geladen wurden, benötigt. Erst dann kann das Proxy-Objekt auf das Service-Interface gecastet und der Service-Registry übergeben werden. Dynamische-Proxys wurden bereits in der Version 1.3 der Java SE eingeführt und sind daher auch mit älteren JVM, wie sie häufig auf eingebetteten Systemen eingesetzt werden, verwendbar.

Im OSGi-Umfeld müssen für das Konzept der Dynamischen-Proxys einige Herausforderungen gelöst werden:

- Um an das Service-Interface zu gelangen, das für die Registrierung des Dynamischen-Proxys benötigt wird, muss dieses, dem OSGi-Konzept folgend, in der Manifest-Datei importiert werden. Für die Proxy-Bundles ist dies kein Problem, da die Bundles, und damit auch das Manifest, erst zur Laufzeit erzeugt werden. Die Dynamischen-Proxys werden allerdings von der Distribution-Software registriert, die folglich alle Packages mit potentiellen Service-Interfaces importieren muss. Da das Manifest zur Laufzeit nicht verändert werden kann, ergibt sich auf dem herkömmlichen Weg keine Lösungsmöglichkeit. Die OSGi-Spezifikation sieht aber für den Fall, dass zur Design-Zeit nicht alle Imports bekannt sind, das dynamische Importieren von Packages vor. Der Key-Eintrag `DynamicImport` veranlasst das Importieren aller im Value angegeben Packages, sobald versucht wird eine Klasse aus einem der angegeben Packages zu laden. Das Format der Values erlaubt dabei die Angabe von Wildcards, so dass der Manifest-Eintrag `DynamicImport = *` das beschriebene Problem für die Distribution-Software löst.
- Wie erwähnt, werden für das Instantiieren der Dynamischen-Proxys der *Invocation-Handler*, die Implementierung zur generischen Verarbeitung der Methodenaufrufe, die Service-Interfaces und genau ein Classloader verlangt. Im OSGi-Umfeld kann es aber vorkommen, dass ein Service mehrere Interfaces implementiert, und diese Interfaces aus verschiedenen Bundles importiert werden. In diesem Fall werden die Service-Interfaces von unterschiedlichen Classloadern geladen. Es ist nun unmöglich den Dynamischen Proxy korrekt zu instantiieren, da mehrere Classloader beteiligt sind, aber nur ein einzelner Classloader bei der Instantiierung übergeben werden kann. Um dennoch Dynamische-Proxys verwenden zu können muss auf der Client-Seite die Service-Struktur aufgebrochen werden. Der Proxy ist nicht länger Stellvertreter für den gesamten Service sondern nur noch für genau ein implementiertes Interface auf unterster Ebene. Interface-Hierarchien sind von dieser Änderung unberührt. Abbildung 5.17 illustriert das Konzept.

  Auf der Server-Seite importiert das Bundle `B_A` Package `P_1` aus Bundle `B_B` und Package `P_2` aus Bundle `B_C`. Es wird ein Service `S_A` registriert, der Interface

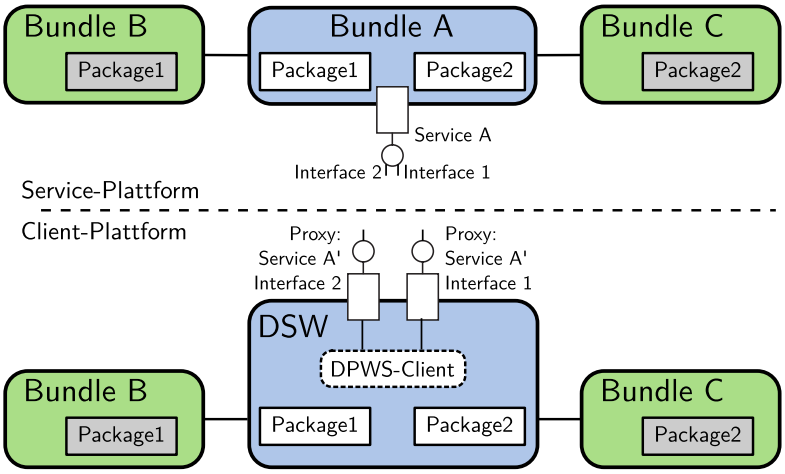


Abbildung 5.17: Aufbrechen der Service-Struktur um die Verwendung von Dynamischen Proxys zu ermöglichen wenn die Service-Interfaces mit unterschiedlichen Classloadern geladen wurden

`I_1` aus Package `P_1` und Interface `I_2` aus Package `P_2` implementiert, und für den entfernten Zugriff ausgezeichnet ist. Auf der Client Seite ist das Importieren der Packages identisch, die Interfaces `I_1` und `I_2` werden dementsprechend mit unterschiedlichen Classloadern geladen. Um Dynamische-Proxys verwenden zu können wird die Struktur des Service aufgespalten, und ein Service `S_A'`, der Interface `I_1` implementiert, und ein Service `S_A'` der Interface `I_2` implementiert im Framework registriert.

Wurde der original Service mit Properties registriert, müssen diese auch auf der Client Seite abrufbar sein. In diesem Fall werden die Properties in beiden Services `S_A'` hinterlegt, um die Nutzungstransparenz für den aufrufenden Client zu gewährleisten.

Da die OSGi-Plattform eine SOA umsetzt, sind alle Komponenten lose gekoppelt und ein Client hat keine Kenntnis über die Struktur oder die Implementierung eines Service. Weiterhin werden in OSGi Services über ihre Interfaces genutzt, so dass die vorgestellte Aufteilung unproblematisch ist. Allerdings gibt es einen problematischen Anwendungsfall. Sucht ein Client nach einem Service, der sowohl Interface `I_1` als auch Interface `I_2` implementiert, so bleibt die Suchanfrage von den Proxys unbeantwortet. Dieser Fall ist allerdings nicht Best-Practise und kann als seltener Sonderfall betrachtet werden. Soll dieses Verhalten dennoch unterstützt werden, ist die Verwendung von Proxy-Bundles obligatorisch.

Insgesamt bietet das Konzept der Dynamischen Proxys eine flexible Lösung, insbesondere in hoch dynamischen Umgebungen. Die Bereitstellung der Proxy-Services kann effizienter als beim Proxy-Bundle-Konzept durchgeführt werden, da die Erstellung des Proxys nur eine Objektinstantiierung darstellt und kein Bytecode generiert werden muss. Service-Aufrufe sind allerdings weniger effizient, da der generische Aufruf über die

Java-Reflection-API mehr Ressourcen verbraucht als der Aufruf über exakt angepassten Code.

5.3.3 Service-Interfaces

OSGi-Services sind Java-Objekte, die ein bestimmtes Interface implementieren. Will ein Client einen solchen Service nutzen, so bekommt er nach erfolgreicher Suche ein untypisiertes Objekt von der Service-Registry übergeben (siehe Unterabschnitt 2.1.3). Um auf die Methoden des Services zugreifen zu können, muss der Client dieses Objekt anschließend auf das Service-Interface casten. Hierbei muss das Service-Interface, das zum Cast verwendet wird, vom selben Classloader geladen werden wie das Interface, das vom Service implementiert wurde. Sowohl der Client, als auch der Service, müssen also das Interface aus dem gleichen Bundle importieren.

Im Falle von Comoros muss das Interface für den Proxy-Service aus dem gleichen Bundle importiert werden wie das für den Cast verwendete Interface im OSGi-Client. Dies hat zur Folge, dass ein Bundle existieren muss, welches das Interface für Comoros und den Client exportiert, ein so genanntes API-Bundle. Dieses API-Bundle muss im Zuge der Proxy-Erstellung im Framework bereitgestellt werden. Dazu existieren zwei Lösungsmöglichkeiten, die manuelle Bereitstellung und die automatisierte Generierung der Interfaces.

Manuelle Bereitstellung

Im Sinne einer SOA ist die Schnittstellen-Beschreibung ein Vertrag zwischen Service-Anbieter und Service-Nutzer. Diese Beschreibung muss dem Service-Nutzer bereits zur Entwicklungszeit zur Verfügung stehen. Im OSGi-Umfeld führt das zur gängigen Praxis der bereits erwähnten API-Bundles, die lose gekoppelt zum Service-implementierenden Bundle existieren und dem Client-Entwickler so auf einfache Weise zur Verfügung gestellt werden können. Diese Bundles können direkt im Framework installiert werden, so dass für die manuelle Bereitstellung keine besonderen Vorkehrungen getroffen werden müssen. Diese Aufgabe wird im Rahmen der Erstellung eines verteilten Enterprise-Systems vom Solution-Deployer übernommen.

Generierte Bereitstellung

Eigentlich müsste mit der obigen Argumentation die manuelle Bereitstellung der Service-Interfaces ausreichend für die Umsetzung einer verteilten OSGi-SOA sein. Es existieren allerdings Szenarien, die dieses Vorgehen erschweren und eine weitere Untersuchung dieser Thematik notwendig machen. Die gängige Praxis der Erstellung von API-Bundles kann nicht vorausgesetzt werden. Sowohl in mobilen Umgebungen und insbesondere in Systemen die ursprünglich zur lokalen Verwendung entwickelt wurden, befinden sich Interface und Service-Implementierung häufig im selben Bundle. Dies macht ein Aufbrechen der lokalen Strukturen in ein verteiltes System schwierig. Abbildung 5.18 verdeutlicht das Problem.

In der lokalen Plattform befinden sich die Service-Implementierung und das Service-Interface in dem selben Bundle. Das Package mit dem entsprechenden Interface wird vom Service-Provider-Bundle exportiert und vom Client importiert. Auf diese Weise ist eine lokale Service-Nutzung möglich. Comoros unterstützt aber explizit die Verwendung

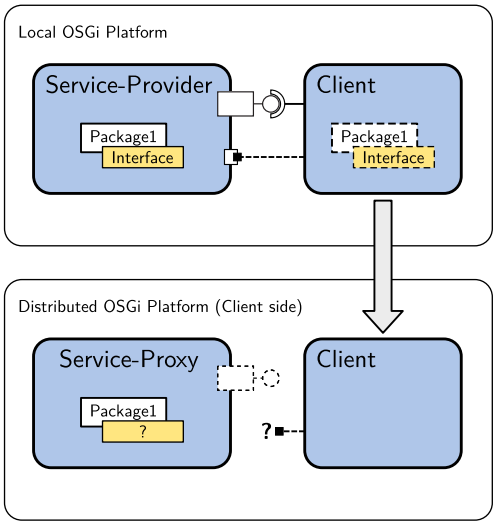


Abbildung 5.18: Fehlen von Service-Interfaces bei der Verwendung von Komponenten in verteilten Umgebungen, die ursprünglich für den Einsatz in lokalen OSGi-Plattformen erstellt wurden

von Legacy-Services und Clients. Also muss es möglich sein, den Client aus der lokalen Plattform zu entfernen und in einer entfernten Plattform zu installieren. Hier gibt es aber zwei Probleme: Zum einen kann der Client das benötigte Interface nicht importieren, das Bundle bleibt also im Zustand *Resolved*, zum anderen kann der Proxy aufgrund des fehlenden Interfaces erst gar nicht erzeugt werden.

Zwar ist die Unterstützung von Legacy-Systemen nicht Bestandteil der Comoros-Core-Architektur, allerdings muss dieses Szenario schon jetzt berücksichtigt werden, um eine spätere Unterstützung nicht auszuschließen. Der Systementwurf sieht also vor, die Interfaces der entfernten Services dynamisch zur Laufzeit mittels Bytecode-Generierung auf der Client-Plattform zu installieren, eine manuelle Bereitstellung ist somit nicht mehr notwendig. Dies gilt vornehmlich für die konfigurationsgesteuerte Bereitstellung der Proxys (siehe Unterabschnitt 5.3.1). Da die anfragengesteuerte Bereitstellung der Proxys vorschreibt, dass bestimmte RS-Properties bei der Suche gesetzt sind, muss der Client bei der Entwicklung für ein verteiltes Umfeld ausgelegt worden sein. Dem OSGi-Best-Practise-Konzept folgend kann somit davon ausgegangen werden, dass hier die Verwendung von API-Bundles bereits umgesetzt ist.

Bei der Konzeptionierung eines geeigneten Systems zur Interface-Generierung ergeben sich, bedingt durch die Funktionsweise der OSGi-Plattform, einige Schwierigkeiten auf der Client-Plattform, die bedacht werden müssen. Es wurde bereits festgelegt, dass die Interfaces in eigenständige API-Bundles verpackt und von dort exportiert werden. Somit ergeben sich folgende Ansätze:

- **Generierung eines API-Bundles für jeden Proxy**

  Wird ein Proxy erstellt, der das Interface `package_1.Interface_A` implementiert, so wird ein API-Bundle generiert, das genau dieses Interface enthält und `package_1`

für andere Bundles exportiert.

Dieser Ansatz funktioniert nur so lange, bis ein Proxy erstellt wird, der das Interface `package_1.Interface_B` implementiert. In diesem Fall würde ein zweites API-Bundle generiert werden, und zwei Bundles würden nun das selbe Package exportieren. An dieser Stelle wird es problematisch, da das OSGi-Framework zu einem Zeitpunkt das Exportieren eines Packages – in seiner Version – nur aus einem Bundle erlaubt. Ein Client-Bundle, das `package_1` importiert, hätte also entweder `Interface_A` oder `Interface_B` zur Verfügung, jedoch nie beide gleichzeitig. Dabei wird immer das Package importiert, das von dem Bundle mit der niedrigsten `BundleId` exportiert wird. Das gesamte Verhalten wird als *Split Packages Problem* bezeichnet.

Um Split-Packages zu vermeiden, muss ein Package also von genau einem Bundle exportiert werden. In dem vorgestellten Fall muss das erste API-Bundle also deinstalliert werden und ein neues API-Bundles das `Interface_A` und `Interface_B` enthält generiert werden. Diese Neugenerierung erzeugt aber einen Overhead, der gerade im Umgang mit eingebetteten Systemen unerwünscht ist und kann weiterhin zu unerwarteten Nebeneffekten führen, da ein Interface kurzfristig nicht verfügbar ist.

- **Generierung eines API-Bundles mit Fragmenten für jedes Package**

  Für den selben Anwendungsfall, wie im vorherigen Abschnitt, wird bei dieser Lösung ein Bundle erzeugt, welches das erforderliche `package_1` exportiert. Dieses Bundle enthält hier nur die Package-Definition aber keine Interfaces. Für die Interfaces werden nun jeweils separate Fragment-Bundles [The12b] erzeugt, die an das Package-Bundle angehängt werden und so dessen Klassenpfad erweitern. Das Package-Bundle exportiert nun alle Interfaces, die im Package enthalten sind und durch Fragment-Bundles angehängt wurden. Abbildung 5.19 illustriert das Vorgehen.

  Im ersten Schritt wird das Package-Bundle für `package_1` und ein Fragment-Bundle, das `Interface_A` enthält generiert. Wird nun `Interface_B` für die Erstellung eines Proxys benötigt, so wird ein weiteres Fragment-Bundle erzeugt, das dieses Interface enthält, und an das Package-Bundle angehängt. Eine Besonderheit tritt auf, wenn ein Interface ein anderes Interface erweitert, das wiederum in einem anderen Package liegt. Die durch diese Vererbungshierarchie entstehende Abhängigkeit wird aufgelöst, indem das Fragment-Bundle, welches das erbende Interface enthält, zusätzlich das Fragment-Bundle mit dem zu erweiternden Interface importiert.

  Ist der Vorteil gegenüber des ersten Ansatzes offensichtlich, gibt es aber auch bei dieser Lösung ein Problem. Die Erweiterung des Package-Bundle-Klassenpfades um zusätzliche Klassen/Interfaces durch Fragmente wird erst durch eine Aktualisierung des Hosts, also des Package-Bundle, wirksam. Diese Aktualisierung übernimmt der *Package Admin Service*. Der Package-Admin-Service ist ein System-Service, Bestandteil des OSGi-Kerns und in jeder Plattform-Instanz verfügbar. Innerhalb von OSGi übernimmt er das Management von Package-Abhängigkeiten. Über

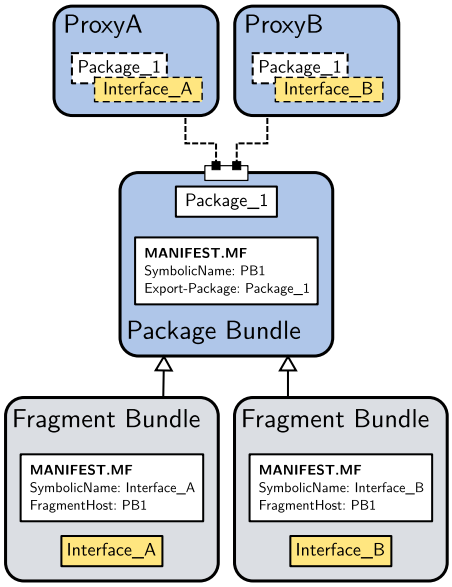


Abbildung 5.19: Exportieren verschiedener Interfaces über Package-Bundles und die mittels Fragment-Bundles angehängten Interfaces

die Methode `refreshPackages()` werden die Abhängigkeiten eines Bundles neu ermittelt und aktualisiert. Innerhalb dieses Vorgangs werden alle Bundles ermittelt, die direkt oder indirekt vom betroffenen Package-Bundle abhängig sind. Es wird ein Abhängigkeitsbaum erstellt und alle Knoten (Bundles) des Baums werden gestoppt und neu gestartet. Wurden die Proxys als dynamische Proxys realisiert, würde folglich die gesamte Comoros-Distribution-Software, die eine Vater-Beziehung zu allen Proxys unterhält, neu gestartet. Ein Verhalten, das nicht akzeptabel ist. Wurden die Proxys allerdings als separate Proxy-Bundles realisiert, so würden nur einzelne Proxy-Bundles neu gestartet werden. In Systemen mit statischem Kommunikationsmuster tritt der Anwendungsfall sehr selten auf. Da Proxy-Services zudem zustandslos und lose gekoppelt sind, ist das Verhalten in der Praxis zumeist akzeptabel. Schlussendlich obliegt die Verantwortung in der Rolle des Solution Deployers, der entscheiden muss, ob eine dynamische Generierung der Service-Interfaces angewendet werden kann.

5.3.4 Registrierung des Proxy-Service

Die abschließende Registrierung des Proxy-Service verhält sich für alle vorgestellten Varianten – dynamische Proxys oder Proxy-Bundles – gleich. Nach Instantiierung des Service-Objekts wird dieses der OSGi-Service-Registry übergeben.

Um ein exaktes Abbild des originalen Service zu erreichen, müssen bei der Registrierung auch alle Service-Properties des ursprünglichen Service angegeben werden. Dabei ist allerdings zu beachten, dass einige Remote-Service-spezifische Properties ausgefiltert und nicht registriert werden. Dabei handelt es sich um alle Properties, die nur auf der Server-Seite einen Sinn ergeben, wie z. B. die Properties `service.exported.*`. Dahingegen müssen

auf der Client-Seite andere RS-spezifische Properties eingeführt werden. Die Property `service.imported` markiert den Service als Proxy für einen entfernten OSGi-Service und ist somit die Voraussetzung für die Regelung der Zugriffstransparenz. Clients können nun die Nutzung von Remote-Services explizit verbieten oder vorschreiben. Weiterhin gibt die Property `service.imported.*` an, über welches Transportprotokoll der entfernte Service angeboten wird. Auch hier kann der Client nun bestimmte Protokolle vorschreiben. Alle weiteren Properties sind in der RS-Spezifikation angegeben (siehe Unterabschnitt 3.2.2 beziehungsweise [The12a]).

Bei der Deregistrierung eines Proxy-Service ergibt sich für die Variante der Proxy-Bundles ein weiterer Schritt. In diesem Fall wird das gesamte Proxy-Bundle zur Deinstallation vorgemerkt. Sobald alle Bindungen von den entsprechenden Clients zum Service aufgelöst sind, wird dieser Schritt vollzogen.

5.3.5 Konfigurationsvorgaben

Die unterschiedlichen Verfahren und Konfigurationen von ws4d.Comoros bieten eine Vielzahl von Nutzungsmöglichkeiten. Dieser Abschnitt gibt eine Übersicht, welche Konfigurationen bei unterschiedlichsten Anwendungsfällen sinnvoll ist.

Die verschiedenen Anwendungsfälle setzen sich aus grundlegenden Bausteinen zusammen. Diese Bausteine beziehen sich auf Systemeigenschaften, die der Solution Deployer im Vorfeld bestimmen muss, und sind folgendermaßen definiert:

- **Statisches Kommunikationsmuster (statCom):** Die Bindungen zwischen Clients und Services sind relativ fest. Sie werden zu Beginn festgelegt und aufgebaut und verändern sich danach nicht mehr.
- **Dynamisches Kommunikationsmuster (dynCom):** Die Bindungen zwischen Clients und Services sind einem ständigen Wechsel unterworfen. Services verschwinden, Clients gehen Bindungen zu besser geeigneten Services ein, neue Services oder ganze Plattformen erscheinen. All das führt zu einem hoch dynamischen System, bei dem Bindungen eine kurze Lebensdauer haben.
- **Legacy Services (ls):** Das System enthält Elemente, zumeist Altkomponenten, die nicht für verteilte Anwendungen entwickelt wurden.
- **Hoher Kommunikationsaufwand (highCom):** Über die bestehenden Bindungen fragen Clients die Services häufig an. Sowohl Anfrage- als auch Antwortdaten sind dabei verhältnismäßig groß.
- **Geringer Kommunikationsaufwand (lowCom):** Es gibt wenige Serviceaufrufe, die ausgetauschten Daten sind verhältnismäßig klein.

Um diese Anwendungsfälle effizient zu unterstützen bietet Comoros unterschiedliche Konfigurationen in den Bereichen Proxy (ProxyBundles oder dynamische Proxys), Proxy-Bereitstellung (konfigurationsgesteuert oder anfragengesteuert) und Interface-Bereitstellung (manuell oder generiert) an. Einige Konfigurationen schließen sich dabei gegenseitig aus. Sowohl eine anfragengensteuerte Bereitstellung, als auch generell dynamische Proxys sind nicht mit der Bereitstellung generierter Interfaces kombinierbar

(siehe Unterabschnitt 5.3.3). Betrachtet man zusätzlich die drei folgenden Grundsätze – `statCom` empfiehlt eine konfigurationsgesteuerte Bereitstellung der Proxys, `highCom` empfiehlt die Verwendung von Proxy-Bundles, `ls` empfiehlt die Generierung von Interfaces – ergibt sich die Tabelle 5.3.

Tabelle 5.3: Empfehlungen für die Konfiguration von ws4d.Comoros

Use Case	*Proxy type*	*Deployment*	*Interfaces*	*Bemerkung*
statCom + highCom + ls	Proxy-Bundles	Konfigurationsgesteuerte Bereitgestellung	Generierte Interfaces	
statCom + highCom	Proxy-Bundles	Konfigurationsgesteuerte Bereitgestellung	Manuelle Bereitstellung	
dynCom + lowCom + ls	Dynamische Proxies	Anfragengesteuerte Bereitgestellung	Manuelle Bereitstellung	Nur möglich, wenn API-Bundle vorliegt
dynCom + lowCom	Dynamische Proxies	Anfragengesteuerte Bereitgestellung	Manuelle Bereitstellung	
statCom + lowCom + ls	1) Proxy-Bundles 2) Dynamische Proxys	Konfigurationsgesteuerte Bereitgestellung	Generierte Interfaces	
statCom + lowCom	1) Proxy-Bundles 2) Dynamische Proxys	Konfigurationsgesteuerte Bereitgestellung	Manuelle Bereitstellung	
dynCom + highCom + ls	1) Dynamische Proxys 2) Proxy-Bundles	Anfragengesteuerte Bereitgestellung	Manuelle Bereitstellung	Nur möglich, wenn API-Bundle vorliegt
dynCom + highCom	1) Dynamische Proxys 2) Proxy-Bundles	Anfragengesteuerte Bereitgestellung	Manuelle Bereitstellung	

5.4 Methodenaufruf

Eine wichtige Anforderung ist, dass es für einen Client keinen Unterschied zwischen der Benutzung eines entfernten oder eines lokalen Service gibt (siehe Unterabschnitt 4.3.2). Im Zuge dieser Nutzung sucht ein Client über die Schnittstellen zur OSGi-Plattform nach einem Service und bekommt entweder ein lokales Service-Objekt oder ein Proxy-Objekt zurück, das auf das entsprechende Service-Interface gecastet werden muss.

Im Falle eines Proxy-Objekts, kommuniziert der Proxy bei einem Methodenaufruf mittels SOAP-Nachrichten mit dem entsprechenden Skeleton. Die SOAP-Messages enthalten die gemarshalten Daten, die auf der Server-Seite wieder in Java-Objekte zurückgeführt werden, und der Methodenaufruf wird schließlich vom Skeleton an den OSGi-Service weitergeleitet. Eventuelle Rückgabewerte des OSGi-Service werden entsprechend in die andere Richtung versendet.

In diesem Prozess gibt es drei Aspekte, die etwas genauer betrachtet werden müssen: Das Konzept des *Late Binding*, das Behandeln von Fehlerfällen und der Prozess des Daten-Marshaling.

5.4.1 Late Binding

Late-Binding definiert ein Konzept, bei dem die versendeten Datentypen erst zur Laufzeit festgelegt werden. So kann einer Methode mit einem Aufrufparameter `A`, ein Objekt `B` übergeben werden, falls `B` die Klasse `A` erweitert. Ähnlich verhält es sich mit Methoden, deren Aufrufparameter eine abstrakte Klasse oder ein Interface ist. Auch hier muss die übergebene Objektinstanz das Interface implementieren oder die abstrakte Klasse erweitern, ist ansonsten in ihrer Facette aber nicht weiter eingeschränkt.

Comoros setzt dieses Konzept um, indem der konkrete Datentyp im serialisierten Objekt angegeben wird, um so im Prozess des Demarshaling die korrekte Instanz zu erzeugen. Das serialisierte Objekt kann aber weiterhin auch vom Marshaller der Oberklasse verarbeitet werden, in dem Fall gehen aber Informationen verloren.

Eine Besonderheit ergibt sich bei der Verarbeitung von `Collections` und `Maps`. Hier muss das LateBinding-Konzept aufgrund der Berücksichtigung der vorgegebenen Call-By-Value-Semantik eingeschränkt werden. Über die Hilfsklasse `java.util.Collection` können beispielsweise Collections um bestimmte Facetten angereichert werden. So können spezielle Derivate erzeugt werden, die als Singleton fungieren oder nicht modifizierbar sind. Da die Semantik dieser Facetten nur beim lokalen Service-Aufruf Geltung hat, werden diese Derivate nicht unterstützt. In Comoros werden nur Duplikate der Daten versendet, den Facetten wird so in einer verteilten Umgebung ihr Sinn entzogen.

5.4.2 Exception Handling

Im Betriebsablauf können grundsätzlich zwei verschiedene Fehlerkategorien auftreten: Fehler, die lokal beim Aufruf des OSGi-Service durch den Skeleton auftreten, und Fehler, die durch die Infrastruktur des verteilten Systems verursacht werden.

- Wenn ein lokaler OSGi Client einen Proxy-Service aufruft, so wird ein SOAP-Request verschickt und von JMEDS in der Rolle des Objektadapters empfangen. Der Request wird an den Skeleton-Service weitergeleitet, der schließlich den OSGi-Service aufruft. Dieser Aufruf kann fehlschlagen und eine Exception wird ausgelöst.

Mittels des in Unterabschnitt 5.2.3 vorgestellten Mappings wird diese Exception gemarshalt und über WS-Fault-Parameter des SOAP-Response zur Client-Seite zurückgeschickt. Dort wird die Exception innerhalb des Proxy-Service geworfen, so dass der Client ein Fehlerbehandlung vornehmen kann.

- Im zweiten Fehlerfall geht der SOAP-Request oder der SOAP-Response aufgrund von Netzwerkproblemen verloren. Abbildung 5.20 zeigt das Vorgehen in diesem Fall.

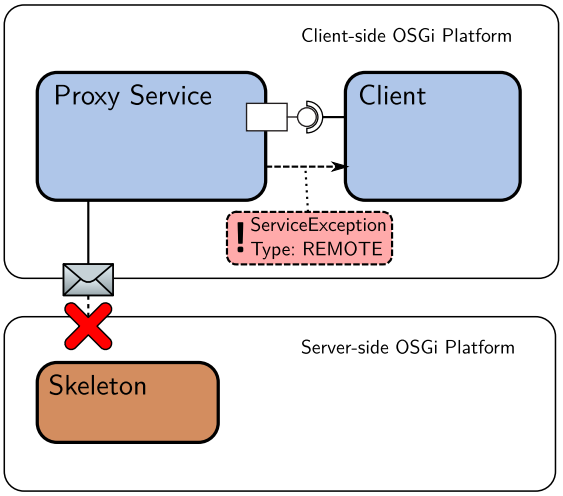


Abbildung 5.20: Auftreten einer `ServiceException` vom Typ `REMOTE` aufgrund von Netzwerkproblemen

Nach einem Timeout, resultierend aus dem Verlust der SOAP-Nachricht, und damit der Nichterreichbarkeit des Skeletons, wird im Proxy-Service eine von OSGi spezifizierte `ServiceException` vom Typ `REMOTE` geworfen. Dieses Fehlermodell kann auch für Legacy-Services verwendet werden, da die `ServiceException` eine Runtime-Exception ist, und daher zu jeder Zeit ohne Spezifizierung im Service-Interface auftreten kann. Es wurde also kein neues Fehlermodell eingeführt, und die Benutzung zwischen entfernten und lokalem Service unterscheidet sich nicht.

Der erste Fehlerfall liegt außerhalb von Comoros und bedarf daher keiner weitergehenden Betrachtung. Im zweiten Fehlerfall allerdings muss Comoros weitere Maßnahmen ergreifen die der Vermeidung weiterer Fehler dienen. Es muss die Entscheidung getroffen werden, ob der Proxy-Service, dessen zugehöriger Skeleton nicht erreichbar ist, deregistriert werden soll oder nicht. Dazu muss der Lebensstatus des Skeleton überprüft werden.

Der Lebenszyklus eines DPWS-Device wird durch die `Hello`- und `Bye`-Nachricht begrenzt. Im Falle von Netzwerkproblemen sendet das Device aber keine `Bye`-Nachricht, und so kann es zu den oben beschriebenen Problemen kommen. Gerade in mobilen Umgebungen kann es aber sein, dass die Verbindung zu dem Device nur kurzzeitig gestört ist und nach dieser Zeit wieder normal verwendet werden kann. Um den, gerade im Falle von Proxy-Bundles, relativen teuren Prozess der Deregistrierung und späteren

Neuerstellung zu vermeiden, lohnt es sich die Verbindung kurzzeitig zu überwachen und so gegebenenfalls den Proxy zu erhalten. Dazu sendet die Distribution-Software in Intervallen `DirectedProbe`-Nachrichten an das Device. Wird kein `ProbeMatches` empfangen, ist das Device verloren, und der Proxy wird deregistriert. Für diesen Prozess wird ein Algorithmus verwendet, der an den *Retransmission Algorithm* für UDP-Nachrichten aus der SOAP-over-UDP-Spezifikation angelehnt ist [Nix09a].

```
1. if PROBE_REPEAT <= 0 go to Step 10;
2. else PROBE_REPEAT−−;
3. Generate a random number T in
     [PROBE_MIN_DELAY .. PROBE_MAX_DELAY];
4. Wait T milliseconds;
5. Transmit;
6. if PROBE_REPEAT <= 0 OR
     ProbeMatches received goto Step 10;
7. else PROBE_REPEAT−−;
8. T = T ∗ 2;If T > PROBE_UPPER_DELAY then
                    T = PROBE_UPPER_DELAY;
9. go to 4;
10. Done.
```

Über die Konstanten (siehe Tabelle 5.4) kann der Algorithmus konfiguriert werden. So kann der Solution Deployer je nach Umgebung (mobil, statisch) entscheiden, ob, und in welchem Umfang, diese Überprüfung Sinn ergibt. Wird die Konstante `PROBE_REPEAT` beispielsweise auf `0` gesetzt, so werden Proxys im Fehlerfall direkt deregistriert.

Tabelle 5.4: Konstanten und deren Default-Belegung für den Algorithmus zur Überprüfung des Lebensstatus eines Skeletons

Constant	*Default Value*
PROBE_REPEAT	5
PROBE_MIN_DELAY	2000
PROBE_MAX_DELAY	20000
PROBE_UPPER_DELAY	30000

5.4.3 Marshaling

Das Marshaling bezeichnet den Prozess der Transformation der Daten in ein übertragungsfähiges Format. Die Wiederherstellung der Daten nach dem Empfang wird dann als Unmarshaling bezeichnet. So wird sichergestellt, dass die an einer Kommunikation beteiligten Knoten auf einer einheitlichen Datenrepräsentation arbeiten können. Aufgrund der Verwendung des DPWS als Kommunikationsprotokoll wurde in Comoros XML als Standard-Datenformat gewählt. Das dazu gehörende Mapping der einzelnen zu unterstützenden Datentypen wurde bereits in Unterabschnitt 5.2.3 vorgestellt.

Für die Durchführung des Marshaling-Prozesses wurde die Comoros-Marshaling-Komponente entwickelt, die innerhalb von OSGi einen Marshaling-Service anbietet. Die Verwendung dieses Services kann in drei Funktionsblöcke unterteilt werden:

1. Erstellung eines XML-Schemas im Prozess der Skeleton-Generierung
2. Transformation von Java-Instanzen in XML-Instanzen auf Basis des XML-Schemas
3. Transformation von XML-Instanzen in Java-Instanzen

Für die Bearbeitung aller Aufgaben wurde in der Marshaling-Komponente der *ParameterConverter* entwickelt. Der Parameter-Converter bietet, wie in Abbildung 5.21 dargestellt, eine Schnittstelle zur Umwandlung der Datentyp-Definitionen und ihrer konkreten Instanzen zwischen OSGi (Java) und DPWS (XML) in beide Richtungen.

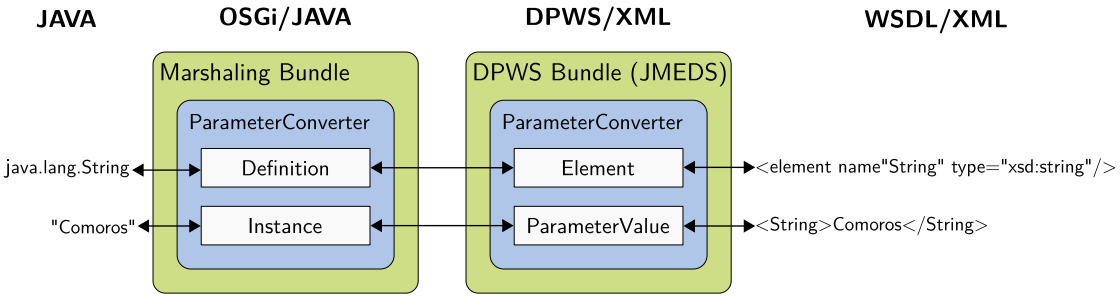


Abbildung 5.21: Prozess des Marshaling/Unmarshaling mittels des Parameter-Converter

Die Erstellung der XML-Schema-Definitionen für die WSDL des Skeletons erfolgt dabei innerhalb der *Bereitstellungsphase* (siehe Unterabschnitt 5.1.1). Hier wird dem Marshaling-Service die Signatur jeder Methode des Java-Interfaces übergeben, zu der dann passende XML-Schema-Definitionen erstellt werden. Dabei ist keine direkte Verarbeitung von XML-Dokumenten notwendig. Die verwendete JMEDS-Bibliothek stellt dem Benutzer eine Schnittstelle zur Verfügung, in der von einer dokumentbasierten Beschreibung und Verarbeitung von XML-Daten abstrahiert wird. Über diese Schnittstelle lassen sich beliebige XML-Elemente (*Parameter/Element*) definieren und weiterhin auf die konkreten XML-Instanzen (*ParameterValue*) zugreifen. Der Parameter-Converter bemächtigt sich dieser Abstraktionsschicht und erstellt XML-Schema-Elemente für die Klassen der unterstützten Java-Klassen. Enthält die Methodensignatur eine nicht unterstützte Java-Klasse, so wird im Parameter-Converter eine `TypeNotSupported`-Exception erzeugt und über den Marshaling-Service an die aufrufende Komponente übergeben.

Während der *Kommunikationsphase* erfolgt nun die Transformation der Instanzen. Hierzu übergibt die aufrufende Komponente dem Marshaling-Service entweder eine Java-Instanz oder eine XML-Instanz. Diese Instanz wird an den Parameter-Converter weitergeleitet, der dann, mit Hilfe der JMEDS-Abstraktionsschicht, eine Instanz der jeweils anderen Technologie erzeugt. Bei der Erzeugung der XML-Instanz wird dabei auf das zuvor nach den in Unterabschnitt 5.2.3 definierten Abbildungen erstellte XML-Schema zurückgegriffen und eine dazu gültige Instanz erzeugt. Die aufrufende Komponente bekommt im Anschluss als Antwort des Service-Aufrufs die jeweilige Instanz übergeben. Diese kann dann an den DPWS-Skeleton/Client oder an den OSGi-Service/Client übermittelt werden.

5.5 Zwischenfazit

Die Kernarchitektur von ws4d.Comoros bietet eine vollständige Implementierung der OSGi-Remote-Services-Spezifikation unter Verwendung der Prinzipien aus dem Bereich der verteilten Objektsysteme. Durch den Einsatz des DPWS als Basis-Kommunikationsprotokoll wird bereits durch die Kernarchitektur ein verteiltes und flexibles Dienste-System auf Basis offener Standards realisiert, das die einfache Entwicklung verteilter OSGi-Anwendungen ermöglicht.

Bei der Integration des DPWS in das OSGi-Umfeld mussten, aufgrund der konzeptionellen Unterschiede der beiden Technologien, verschiedene Herausforderungen gelöst werden. Dazu gehörte vor allem die Abbildung der OSGi-Strukturen auf DPWS-Strukturen, um ohne Informationsverlust eine vollständige Rekonstruktion eines OSGi-Service auf einer entfernten Plattform zu ermöglichen. Dieser Punkt enthält auch die Abbildung von Java-Datentypen auf XML-basierte Datentypen zur Unterstützung eines homogenen Daten-Übertragungsformats. An dieser Stelle wurde innerhalb von Comoros ein vollständiges Konzept zur Serialisierung und Deserialisierung der unterschiedlichen Datentypen entwickelt.

In Kapitel 4 wurde eine ausführliche Analyse aller für Comoros relevanten Anwendungsfälle durchgeführt. Diese sind in Tabelle Tabelle 4.1 gelistet. Durch die Kernarchitektur werden die folgenden Anforderungen abgedeckt:

- **OSGi-Client nutzt entfernten OSGi-Service**
  Hierbei handelt es sich um die funktionale Kernanforderung der RS-Spezifikation. Comoros realisiert diese Anforderung mittels des DPWS und ermöglicht somit die Entwicklung verteilter OSGi-Anwendungen nach den Vorgaben der Spezifikation.
- **Unterstützung des Life-Cycle von OSGi und OSGi-fremder Technologien**
  Ein Proxy auf einer entfernten Plattform ist an den Lebenszyklus des originalen Service gebunden. Wird dieser deregistriert, so muss auch der Proxy aus der Plattform entfernt werden. Comoros setzt diese Anforderung um, indem der DPWS-Skeleton an den Lebenszyklus des originalen Service gebunden wird und der Proxy an den des Skeletons. So wird transitiv ein einheitlicher Lebenszyklus umgesetzt. Weiterhin überwacht Comoros nicht nur die Services, die bereits eine bestehende Bindung zu einem entfernten Client halten, sondern auch die gesamte OSGi- und DPWS-Umgebung nach neu erscheinenden Services. So können auch Bindungen erstellt werden, nachdem eine zuvor erfolglose entfernte Servicesuche nachträglich durch das Registrieren eines weiteren Service erfüllt wird.
- **Transparenz**
  Comoros setzt eine verteilte serviceorientierte Architektur um. OSGi-Clients können nach Services suchen, ohne zu wissen, ob der angefragte Service lokal oder entfernt zur Verfügung steht. Somit muss der Client nichts über den physischen Ort des Service wissen. Es ist ihm jedoch möglich diesen zu erfahren, um so verteilungsbezogenes Verhalten (z. B. höhere Latenz) zu berücksichtigen. Somit ist die Ortstranzparenz sichergestellt. Weiterhin unterscheidet sich beim Einsatz von

Comoros für den OSGi-Client der Zugriff auf lokale und entfernte Services nicht. Die Einhaltung dieser Anforderung (Zugriffstransparenz) ist auch gerade für die Einbindung von Altkomponenten wichtig und darf somit auch vom Comoros-Kern nicht verletzt werden. Im Sinne der Fehlertransparenz war es zudem wichtig kein neues Fehlermodell einzuführen. In Comoros werden daher verteilungsbezogene Fehler auf eine von OSGi definierte Laufzeit-Exception abgebildet.

- **Effizienz**
  Um den Einsatz in mobilen und ressourcenbeschränkten Umgebungen zu ermöglichen, wurde als Kommunikationsprotokoll das DPWS gewählt. DPWS ist eine Untermenge von WS-Standards um Webservice-basierte Kommunikation auf eingebetteten Systemen zu ermöglichen. Im Bereich der entfernten Servicesuche wurden umfangreiche Analysen vorgenommen um die Anzahl der versendeten Nachrichten zu minimieren. Mit dem DPWS-Hook-Mechanismus wurde schlussendlich ein mächtiger Mechanismus mit minimaler Nachrichtenanzahl entwickelt und in Comoros integriert. Auch im Bereich der Kommunikation wurden für unterschiedliche Anwendungsfälle (z. B. dynamische und statische Kommunikationsmuster) jeweils spezielle Konzepte entwickelt (Proxy-Bundles und Dynamische Proxys). So kann für jeden Anwendungsfall die effizienteste Lösung realisiert werden. Schlussendlich ist auch für die Abbildung der Java-Datentypen auf XML-Datentypen ein Mechanismus entwickelt worden, der möglichst kleine XML-Instanzen erzeugt, um den XML-Overhead zu minimieren.

- **Programmierkomfort**
  Der Komfort für die Entwickler von verteilten OSGi-Anwendungen wird insbesondere bei der Implementierung für die Nutzung entfernter Services festgelegt. Wird Comoros als Middleware verwendet, so muss der Anwendungsentwickler kein neues Programmiermodell erlernen. Die Unterschiede in der Verwendung lokaler Services liegen einzig in der Verwendung spezieller Properties. Bei der konfigurationsgesteuerten Bereitstellung kann auch darauf verzichtet werden, so dass sich die Implementierung für die Nutzung lokaler und entfernter Services nicht mehr unterscheidet.

- **Wartbarkeit und Erweiterbarkeit**
  Die Wart- und Erweiterbarkeit wird in Comoros durch die Komponentenorientierung erreicht. Der Comoros-Kern ist in einzelne Teile unterteilt, die jeweils Funktionseinheiten bilden (Marshaling, DPWS, DSW). So wird zum einen die Wartung durch eine Strukturierung des Programm-Codes vereinfacht, und zum anderen die Erweiterbarkeit gewährleistet. Neue Funktionen, wie z. B. neue Kommunikationsprotokolle können durch die Entwicklung und die Installation neuer Komponenten zur Laufzeit ergänzt werden.

- **Konsistenz zum OSGi-Programmiermodell**
  Wie im Abschnitt zum Nutzerkomfort bereits beschrieben, wurde das OSGi-Programmiermodell nicht verändert.

- **Verwendung bestehender, offener Standards**

  Comoros implementiert die OSGi-Remote-Services Spezifikation unter Verwendung des *OASIS WS-DD* Standards, der das DPWS enthält. Insgesamt basiert die gesamte Entwicklung auf eine Reihe von Standards aus dem OSGi- und Webservice-Umfeld.

- **Unabhängigkeit von der OSGi-Implementierung**
  Comoros verwendet nur Funktionen aus der OSGi-Spezifikation. Somit ist Comoros unter allen vorhandenen OSGi-Implementierungen (Felix, Equinox, Knopflerfish, mBS) lauffähig.

Damit setzt der Comoros-Kern bereits eine große Menge der definierten Anforderungen um. Wie in den Kapiteln Kapitel 3 und Kapitel 4 bereits analysiert, reicht das aber nicht aus, um das Ziel einer komfortablen Anwendungsentwicklung innerhalb einer flexiblen Dienste- und Geräte-Umgebung zu realisieren. Insbesondere im Funktionellen fehlt die Unterstützung von OSGi-fremden Services und Clients, die Event-basierte Kommunikation, die serviceorientierte Verwendung von Geräten und die Integration von Altkomponenten. Dazu kommt die Adaptierbarkeit des Systems (Überwachung und Konfiguration) um wechselnde Anforderungen und Umgebungen zu unterstützen, die Maskierung der heterogenen Systemstruktur und Anforderungen zur Datensicherheit.

Verglichen mit den in Kapitel 3 vorgestellten bestehenden Lösungen bietet die Kernarchitektur von Comoros bereits einige Vorteile. Alle bisherigen Lösungen bieten lediglich eine Nachrichten orientierte Kommunikation an. Die Unterstützung kontinuierlicher Datenströme ist eine wesentliche Innovation in Comoros. Realisiert wird diese Unterstützung über den WS-Attachment-Mechanismus, so dass der OSGi-Benutzer direkt die Datentypen `InputStream` und `ObjectInputStream` verwenden kann. Viele vorgestellte Lösungen halten sich zudem nicht an bestehende Standards oder führen ein neues Programmiermodell ein. Das ECF ist z. B. an das Equinox-Framework gebunden, das ClusteredOSGi-Projekt setzt eine spezielle Laufzeitumgebung voraus. Das RBI-Projekt bietet zwar eine effiziente Kommunikation an, führt dazu aber ein stark verändertes Programmiermodell ein. Comoros hingegen setzt auf offene Standards und die Beibehaltung des OSGi-Programmiermodells. Dennoch ist Comoros hinreichend effizient. Durch die Verwendung des DPWS, einer leichtgewichtigen Programmierung und einer effizienten Datenserialisierung kann Comoros Anwendung in ressourcenbeschränkten Umgebungen finden. Projekte wie das Apache CXF, Newton oder ClusteredOSGi verwenden hingegen schwergewichtige Bibliotheken, die den Einsatz in diesen Umgebungen erschweren. Die Effizienz von Comoros wird auch durch eine flexible Anpassung an unterschiedliche Anwendungsfälle erreicht. Kein bestehendes Projekt bietet derartige Möglichkeiten. Diese Projekte erreichen einen bestimmten Grad der Effizienz lediglich durch das Angebot verschiedener Kommunikationsprotokolle. Der Comoros-Kern setzt allerdings mit dem DPWS ebenfalls auf ein effizientes Protokoll, später werden sogar noch weitere effizientere Protokolle hinzugefügt.

6 Erweiterte Architektur

Die in Kapitel 5 definierte Kernarchitektur implementiert die OSGi-Remote-Services-Spezifikation und setzt so nur einen Teil der aufgestellten Anforderungen um. Um zusätzliche Funktionen zu realisieren wurden im Rahmen einer erweiterten Architektur weitere Softwarekomponenten entwickelt und wie in Abbildung 5.1 illustriert in eine Gesamtarchitektur eingegliedert. Dieses Kapitel beleuchtet die zusätzlichen Komponenten und beantwortet dabei insbesondere die folgenden Fragestellungen:

- Wie ordnen sich die zusätzlichen Komponenten in die Gesamtarchitektur ein?
- Wie sieht der Architekturentwurf für die einzelnen Komponenten aus?
- Wie wird der Entwickler einer verteilten Anwendung durch die Comoros-Software unterstützt?
- Wie kann auf unterschiedliche und wechselnde Anforderungen an das System adäquat reagiert werden?

Die erweiterte Architektur schließt die Lücke der in Kapitel 4 aufgestellten und durch die Kernarchitektur nicht erfüllten Anforderungen. Dazu wurden, im Sinne der Modularisierung von Comoros, einzelne Komponenten mit in sich abgeschlossenen Funktionseinheiten entwickelt. Diese Teile und ihre Funktionen sind in Tabelle 6.1 definiert.

Durch die Autonomie der einzelnen Komponenten sind in den einzelnen Abschnitten jeweils eigenständige Anforderungen aufgestellt und eine entsprechende Teil-Architektur entwickelt worden. Um zusätzlich die generelle Anforderung zum Einsatz in Ressourcen beschränkten Umgebungen weiter zu unterstützen, können einzelnen Komponenten teilweise nach Bedarf ins Laufzeitsystem geladen werden. So kann eine möglichst leichtgewichtige Middleware realisiert werden. Bei diesen Teilen handelt es sich um das *Erweiterte Marshaling*, die *Event-basierte Kommunikation*, die *Geräteintegration* und die zusätzlichen

Tabelle 6.1: Komponenten der erweiterten Architektur

Name	*Funktion*
Erweitertes Marshaling	Erweiterung der innerhalb der RS-Spezifikation definierten minimal zu unterstützenden Datentypen durch ein generisches Marshaling. Effizienzsteigerung durch Transformationspattern.
Event-basierte Kommunikation	Event-basierte Kommunikation auch für Legacy und OSGi-fremde Komponenten. Minimierung des Datenverkehrs.
Legacy-Systeme	Integration von Altkomponenten in ein verteiltes System ohne Modifikationen. Ausschluss nicht kompatibler Komponenten.
Geräteintegration	Integration von SOA-fähigen Geräten unterschiedlicher Technologien. Werkzeugunterstützung für die entstehenden Datenformat-Abbildungen und Parametrisierung dieser. Integration serieller Geräte in eine SOA-Umgebung.
Kommunikationsprotokolle	Unterstützung weiterer Kommunikationsprotokolle neben dem DPWS, wie z. B. Bluetooth, ZigBee oder UPnP.
Management	Konfiguration der Middleware um unterschiedliche Anwendungsfälle effizient umzusetzen und Überwachung der Systemumgebung. Integration in bestehende Management-Systeme um adaptive Systeme zu realisieren.
Sicherheit	Realisierung gängiger Sicherheitsmechanismen.

Kommunikationsprotokolle. Die Unterstützung von Altkomponenten, das Management der Comoros-Middelware und die Sicherheitsmechanismen sind dagegen generell im System verfügbar.

6.1 Erweitertes Marshaling

Die Kernarchitektur von Comoros unterstützt bereits eine erweiterte Menge der durch die RS-Spezifikation, im Rahmen des *Data Fencing*, vorgeschriebenen Datentypen (siehe Listing 5.2.2). Diese Menge ist für eine umfangreiche OSGi-Anwendungsentwicklung aber zu klein. Der Entwickler ist eingeschränkt und die Integration von Altkomponenten, in denen keine Limitierungen in der Verwendung von Datentypen bestehen, ist a priori ausgeschlossen. Aus diesem Grund darf die Menge der unterstützten Datentypen nicht eingeschränkt werden, mit Ausnahme der allgemeinen Beschränkung auf Datentypen die über das Call-by-Value-Verfahren übergeben werden können.

In einem ersten Entwurf wurde die Erweiterung der Menge der unterstützten Datentypen in Comoros durch die Entwicklung und Installation von Plug-Ins realisiert. Für jede Java-Klasse konnte ein so genannter `CustomMarshaler` geschrieben werden, der einerseits die XML-Schema-Definition der Klasse lieferte und andererseits die Serialisierung und Deserialisierung implementierte. Dadurch konnten beliebige Datentypen unterstützt werden. In der Praxis zeigte sich aber schnell, dass der Entwicklungsaufwand für eine durchschnittliche Menge von Marshalern sehr hoch und zeitintensiv war, und gerade dem Anwendungsentwickler eine hohe Einarbeitung abverlangte. Diese Erkenntnis stand im Gegensatz zu der in Kapitel 4 definierten Anforderung nach einem hohen Komfort für den Benutzer. Somit wurde ein erweiterter, generischer Ansatz entwickelt. Dieser generiert automatisch zu jeder Java-Klasse ein passendes XML-Schema und führt ebenso automatisch das Marshaling und Unmarshaling für diese Klassen durch. Die erste

Implementierung dieser Ideen wurde in einer studentischen Arbeit [Wue11] umgesetzt.

6.1.1 Anforderungen

Die Anwendungsfälle, die für das generische Marshaling zu betrachten sind, umfassen das Bereitstellen eines OSGi-Service für die Nutzung mit beliebigen Webservice-Clients, und die Kommunikation zwischen OSGi-Service und einem entfernten OSGi-Client. Im ersten Fall erstellt die Distribution-Software aus den Methodensignaturen eines OSGi-Service ein XML-Schema-Dokument, das die verwendeten Datentypen beschreibt. Die so definierten XML-Typen werden innerhalb der WSDL-Beschreibung des DPWS-Skeleton angegeben. Bei der Nutzung des Webservice werden die Aufrufe nun durch die Comoros-Distribution-Software in Aufrufe für den lokal registrierten OSGi-Service umgewandelt, das Rückgabeobjekt wird im Anschluss wieder in eine XML-Instanz transformiert und an den Aufrufer zurückgeschickt. Der zweite Anwendungsfall ist nur eine Erweiterung des ersten Anwendungsfalls. Der Aufrufer ist hier ein OSGi-Client, der einen Proxy-Service verwendet. Dieser Proxy enthält wiederum einen DPWS-Client, der anschließend den DPWS-Skeleton aufruft. Es wird also nur ein weiterer Marshaling-Schritt hinzugefügt, die Prinzipien und Funktionsweisen bleiben aber gleich. Insgesamt ergeben sich folgende Anforderungen für die Realisierung einer generischen Marshaling-Komponente zur Unterstützung der beschriebenen Anwendungsfälle:

Generierung des XML-Schemas

OSGi-Services werden anhand ihrer Java-Interfaces definiert, die im Zuge der Struktur-Abbildung auf einen DPWS-Skeleton in einer WSDL repräsentiert werden müssen. Zu diesem Zweck muss es möglich sein alle Java-Datentypen und Klassen in XML-Schemata zu repräsentieren. Dazu sollen, wenn möglich, vordefinierte Datentypen aus der XML-Schema-Spezifikation verwendet werden. Nicht unterstützte Java-Datentypen (z. B. Datentypen, deren Instanzen nur mittels Call-by-Reference ausgetauscht werden können) müssen identifiziert und der Bereitstellungsprozess des DPWS-Skeletons daraufhin abgebrochen werden.

(Un-)Marshaling

Das Basis-Marshaling umfasst die in Listing 5.2.2 definierten Datentypen. Das in der Kernarchitektur entwickelte Marshaling-Konzept wird in die erweiterte Marshaling-Komponente übertragen. Dadurch erfolgt für diese Datentypen kein generisches Marshalling, sondern ein Marshaling anhand der in Unterabschnitt 5.2.3 definierten Abbildungen. Dadurch ist ein effizientes Marshaling der gängigsten Datentypen sichergestellt.

Erweitertes, generisches (Un-)Marshaling

Im erweiterten, generischen Marshaling werden alle Datentypen behandelt, die nicht durch das einfache Marshaling abgedeckt werden. Ohne jeglichen Programmieraufwand soll jede Instanz eines Objekts in eine Instanz des entsprechenden XML-Schema transformiert werden können. Die so erstellten XML-Instanzen können ihrerseits wieder in Java-Objekte umgewandelt werden.

Minimierung der Datenübertragung

Das generische Marshaling erzeugt unter Umständen einen höheren Laufzeit-Aufwand als notwendig, um Instanzen zwischen Java und XML zu transformieren. Das liegt daran, dass zur Laufzeit nicht entschieden werden kann, welche Felder einer Klasse tatsächlich für eine Rekonstruktion ohne Informationsverlust benötigt werden. So wird beispielsweise bei der Transformation einer `HashMap` der Hash-Wert der Einträge abgebildet, obwohl dieser bei der Rekonstruktion neu erzeugt wird. Auf diese Weise kann ein sehr großer Overhead entstehen. Der Entwickler soll die Möglichkeit bekommen, Transformationspattern zu definieren und im System abzulegen. Diese Pattern steuern die Abbildung der dort definierten Datentypen und sorgen somit für eine Minimierung der Daten.

Unterstützung des Late-Binding-Konzept

In Java ist es möglich durch Vererbungshierarchien Objekte entsprechend dieser Hierarchie verschieden zu interpretieren und zu instantiieren. Jedes Java-Objekt ist z. B. vom Objekt `java.lang.Object` abgeleitet. Dadurch ist es möglich, in einem Array des Typs `Object[]` jedes beliebige Objekt zur Laufzeit hinzuzufügen. Somit lässt sich die tatsächliche Klasse eines Objekts innerhalb dieses Arrays zum Zeitpunkt der Generierung des XML-Schemas nicht bestimmen. Daher muss es dem Entwickler möglich sein, im Erstellungsprozess dem erzeugten XML-Schema manuell weitere Klassen hinzuzufügen.

Java-Laufzeitumgebung

Comoros ist für den Einsatz in Ressourcen beschränkten Umgebungen konzipiert worden. Um eine möglichst große Menge an unterstützten Systemen zu erreichen, wird als Laufzeitumgebung Java 1.4 ME im Profil CDC eingesetzt. Somit muss auch die Marshaling-Komponente dieses Profil umsetzten. Eine Abwärtskompatibilität zu vorhergehenden Standards wird nicht verlangt.

6.1.2 Architektur

Die erweiterte, generische Marshaling-Komponente wird Teil der Comoros-Software und erweitert die für die Kernarchitektur entwickelte Komponente. Dabei implementiert die neue Marshaling-Komponente die bisherige Marshaling-API und kann so die alte Komponente ohne Anpassungen anderer Teile der Comoros-Software ersetzen. Abbildung 6.1 gibt eine Übersicht über die Marshaling-Architektur.

Der *Marshaling Service*, registriert von der Marshaling-Komponente, stellt dabei folgende Funktionen zur Verfügung, die von der Comoros-Distribution-Software verwendet werden:

- Erzeugung eines XML-Schema-Dokuments anhand der Signaturen der Methoden des Service-Interface
- Erzeugung einer Java-Instanz für eine übermittelte XML-Instanz
- Erzeugung einer XML-Instanz für eine übermittelte Java-Instanz

Für die Bearbeitung aller Aufgaben bedient sich der Marshaling-Service, ebenso wie der Parameter-Converter der Kernarchitektur, der Abstraktionsschicht des JMEDS-

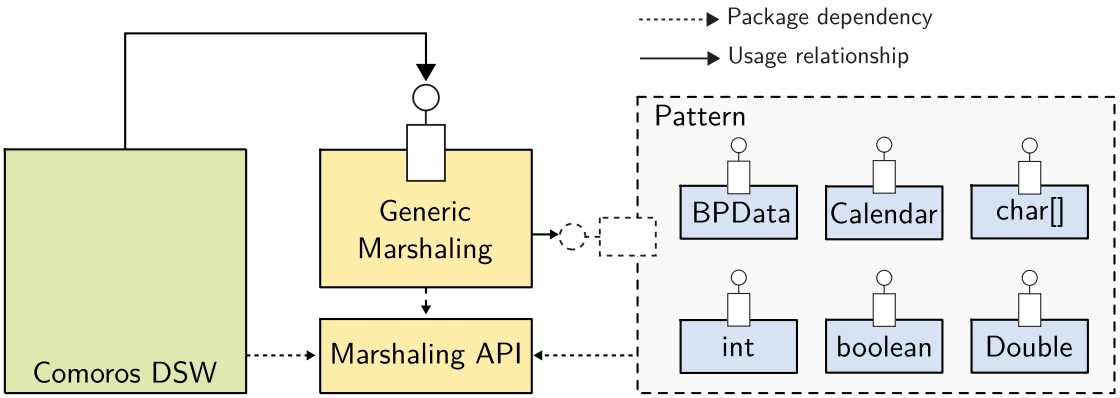


Abbildung 6.1: Architektur der erweiterten Marshaling-Komponente

Frameworks. Diese Schicht erlaubt die transparente Erstellung von XML-Dokumenten, sowohl von Instanz-Dokumenten, als auch von Schema-Dokumenten.

Die Anforderungen an die Marshaling-Komponente sehen vor, dass die Datenübertragung über die Einführung von Transformationspattern minimiert wird. Innerhalb eines solchen Patterns können für die Rekonstruktion eines Objekts unnötige Instanz-Variablen von der Abbildung auf das Schema-Dokument ausgeschlossen werden. Dieser Ausschluss kann nicht automatisch von der generischen Marshaling-Komponente vorgenommen werden, da die Komponente keine Aussage über die Relevanz einer Variable stellen kann. Der Anwendungsentwickler hat diese Informationen aber gegebenenfalls und kann so dafür Sorge tragen, dass redundante oder nicht benötigte Informationen ausgefiltert werden. Dadurch kann die Datenübertragung signifikant minimiert werden.

Transformationspattern implementieren die Marshaling-API und werden als OSGi-Services in der lokalen Plattform registriert. Somit ergibt sich ein erweiterbares Marshaling-System, das vom Entwickler zur Laufzeit an den jeweiligen Anwendungsfall angepasst werden kann. Die Comoros-Marshaling-Komponente stellt initial die Menge von Transformationspattern zur Verfügung, welche die in Unterabschnitt 5.2.3 definierten Abbildungen realisiert. Somit ist eine effiziente Verwendung der durch die RS-Spezifikation vorgegeben Datentypen gewährleistet.

Insgesamt ergibt sich damit der in Abbildung 6.2 abgebildete interne Aufbau der Marshaling-Komponente. Für die Erstellung eines XML-Schema wird der Aufruf der DSW an den `SchemaGenerator` weitergeleitet. Dieser erwartet eine Methodensignatur als Argument um dazu das passende Schema zu erstellen. In diesem Prozess befragt der Schema-Generator die OSGi-Service-Registry nach passenden Transfomationspattern-Services. Diese liefern zu dem jeweiligen repräsentierten Datentyp die passende Schemabeschreibung.

Das (Un-)Marshaling wird an den generischen Marshaler delegiert. Dieser erzeugt aus übergebenen XML-Instanzen Java-Objekte und umgekehrt. Auch hier wird in der lokalen Plattform nach installierten Pattern-Services gesucht und diese gegebenenfalls verwendet.

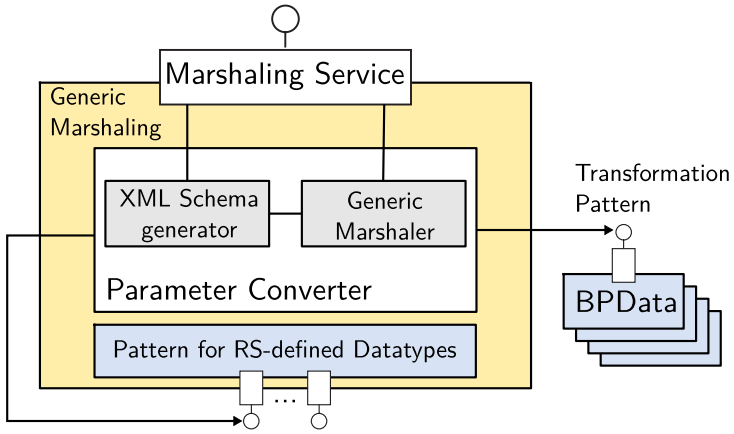


Abbildung 6.2: Aufbau und Funktion der Marshaling-Komponente

6.1.3 Schema-Generierung

Wie beschrieben, ist für die dynamische Erstellung von XML-Schema-Typen zur Repräsentation von Java-Klassen der `SchemaGenerator` verantwortlich. Dabei erhält der Schema-Generator als Eingabe die Methodensignatur einer Methode des Service-Interface. Innerhalb der Signatur müssen drei verschiedene Objekt-Typen unterschieden werden:

- **Eingabeparameter:** Für die Argumente einer Methode wird im XML-Schema ein Element mit komplexen Typ erstellt, der wiederum eine Sequenz von Elementen beschreibt. Als Name erhält das Element die Methodensignatur mit einem Präfix, der das Element als Eingabeparameter identifiziert. Die eigentlichen Parameter werden dann innerhalb der definierten Sequenz eingeordnet. Der Typ des Parameters ist abhängig vom Java-Datentyp oder der Klasse des Arguments. So ergibt sich beispielhaft folgendes Schema:

```
1 <xs:complexType name="in_int_addNumbers_intint">
2  <xs:sequence>
3   <xs:element name="arg_0" type="xs:int"/>
4   <xs:element name="arg_1" type="xs:int"/>
5  </xs:sequence>
6 </xs:complexType>
```

  Enthält die Methode keine Eingabeparameter, wird eine leere Nachricht versendet. In diesem Fall muss kein Schema-Element definiert werden.

- **Rückgabeparameter:** Jede Methode besitzt genau einen oder gar keinen Rückgabeparameter. Ist kein Rückgabeparameter vorhanden, wird eine leere Nachricht versendet, um dem `Request-Response` MEP zu entsprechen. Ansonsten wird für den Parameter ein Element definiert, das dem Java-Typ entspricht. Das Element bekommt keinen eigenen, sondern behält seinen allgemeinen Namen. Somit können mehrere Methoden das selbe Element referenzieren und die Größe des Schema-Dokuments so reduziert werden.

- **Fehlerparameter:** Bei dem Aufruf einer Methode können in Java Exceptions auftreten, die zuvor definiert wurden. Eine Ausmahme bilden die `RuntimeException`, die nicht explizit definiert wurden, aber jederzeit auftreten können. In der WSDL werden solche Ausnahmen im `Fault`-Element zusammengefasst. Die Abbildung von Java-Exceptions auf XML-Schema wurde dabei schon in Unterabschnitt 5.2.3 vorgestellt. Der Schema-Generator setzt diese Abbildung um und fügt dem Schema-Dokument die entsprechenden Typen hinzu.

XML-Repräsentation von Java-Klassen

Bei der in Unterabschnitt 5.2.3 vorgestellten Transformation von Java-Klassen zu XML-Schema werden nur Datentypen (primitive, Wrapper, Collections, Arrays) betrachtet. In Java werden aber zumeist Objekte, die aus eigenen Klassendefinitionen entstanden sind, ausgetauscht. Für die entfernte Kommunikation muss also der Zustand eines jeden Objekts abgebildet werden können. Der Zustand wird dabei durch die Werte aller Instanzvariablen definiert, Methoden und statische Variablen beeinflussen den Objektzustand nicht.

Die Verarbeitung eines Objekts nach diesen Überlegungen im Schema-Generator erfolgt nach einem fest definierten Muster. Zur Darstellung einer Java-Klasse wird ein komplexer Typ definiert, der eine Sequenz von Elementen enthält. In diesen Elementen werden, beim Durchlaufen der Variablen des Java-Objekts, alle Instanzvariablen abgebildet. Sie erhalten den Variablennamen als Elementnamen und als Typ die XML-Repräsentation des Java-Datentyps, beziehungsweise der Java-Klasse. Als Beispiel dient die Klasse `BloodPressureData`:

```
1 public class BloodPressureData {
2
3   public short systolic;
4   public short diastolic;
5   public CommonMedicalData commonData;
6
7   public BloodPressureData(){...}
8
9 }
```

Die Klasse stellt das Datenpaket einer abgeschlossenen Blutdruckmessung dar. Sie enthält zwei Variablen des Typs `short`, für den systolischen und den diastolischen Druck, und weitere Informationen innerhalb einer Klasse `CommonMedicalData`. Die Methoden der Klasse sind für den Objektzustand irrelevant und daher nur angedeutet. Für diese Klasse wird das folgende XML-Schema erstellt:

```
1 <xsd:complexType name ="BloodPressureDataType">
2  <xsd:sequence>
3    <xsd:element name="systolic" minOccurs="1" maxOccurs="1" type="xsd:short"/>
4    <xsd:element name="diastolic" minOccurs="1" maxOccurs="1" type="xsd:short"/>
5    <xsd:element name="commonData" minOccurs="1" maxOccurs="1"
6                                   type="com:CommonMedicalDataType"/>
7  </xsd:sequence>
8 </xsd:complexType>
```

Für die Klasse wurde ein komplexer Typ mit einer Sequenz der Instanzvariablen erstellt. Die Instanzvariablen repräsentierenden Elemente sind jeweils mit `minOccurs="1"` und `maxOccurs="1"` belegt. Als Typ wird der XML-Datentyp `short` verwendet, der dem Java-Datentyp `short` entspricht. Die Variable `commonData` enthält dagegen als Typ einen selbst definierten Typ (Namensraum `com`), der ebenfalls eine Java-Klasse repräsentiert. Im Zuge der Schema-Erstellung wird diese Klasse vom Schema-Generator ebenfalls analysiert. Erst wenn alle beteiligten Klassen abgearbeitet sind, ist auch die Schema-Erstellung der Klasse `BloodPressureData` abgeschlossen.

Abbildung von Vererbungshierarchien

In Java können Klassen von anderen Klassen erben, implizit erbt jede Klasse sogar von der Klasse `java.lang.Object`. Dabei werden auch die Instanzvariablen der Oberklasse an die erbende Klasse weitergegeben. Aus diesem Grund ist es notwendig, das Konzept der Vererbung bei der XML-Schemagenerierung zu berücksichtigen.

XML-Schema selbst unterstützt ein Vererbungskonzept, das über das Java-Konzept hinausgeht. Um eine Java-Vererbungshierarchie darzustellen, ist es ohne weiteres möglich den XML-Schema-Typ einer Oberklasse als Basis der Typdefinition einer Unterklasse anzugeben, wie das folgende Beispiel zeigt:

```
1 <xs:complexType name="SpO2Data">
2   <xs:complexContent>
3     <xs:extension base="CommonMedicalData">
4       <xs:sequence>
5         <xs:element name="spo2" type="xs:integer"/>
6       </xs:sequence>
7     </xs:extension>
8   </xs:complexContent>
9 </xs:complexType>
```

Die Klasse `SpO2Data` erbt von der Klasse `CommonMedicalData` und erweitert diese um die Instanzvariable `spo2`. So können beliebig komplexe und tiefe Vererbungshierarchien abgebildet werden, bis schlussendlich die Klasse `java.lang.Object` erreicht ist. Diese Klasse wird dem XML-Typ `xsd:anyType` zugeordnet. Da in XML-Schema jeder Typ eine Erweiterung von `xsd:anyType` ist, muss diese Erweiterung nicht explizit angegeben werden.

Diese Lösung wird in der erweiterten Marshaling-Komponente von Comoros aber nicht umgesetzt. Ziel des Comoros-Marshaling ist es, den Zustand eines Java-Objekts in XML abzubilden, so dass aus dem XML-Dokument die entsprechende Klasse wieder korrekt instantiiert werden kann. Für diese Anwendungsfall ist es eigentlich nicht notwendig die komplette Hierarchie abzubilden, da diese für den Objektzustand nicht relevant ist. Die Abbildung erzeugt lediglich – in Abhängigkeit von der Komplexität der Hierarchie – einen enormen Overhead in der entstehenden WSDL. Daher werden in Comoros nur die Oberklassen in XML-Schema definiert. Falls die Definition der vollständigen Hierarchie dennoch erwünscht ist, beispielsweise weil zu erwarten ist, dass zur Laufzeit auch viele Instanzen einer der vererbenden Klassen verschickt werden, so muss dies mit der Übermittlung der entsprechenden Konfiguration an Comoros explizit aktiviert werden.

Falls nur für einzelne Klassen die Hierarchie abgebildet werden soll, so muss dies über *Transformationspattern* erfolgen.

Verwendung von Transformationspattern

Transformationspattern dienen dazu, die Abbildung von Java-Klassen auf XML-Schema zu steuern, um die Datenübertragung zu minimieren. Einige Java-Klassen enthalten Instanzvariablen, die bei jeder Instantiierung neu gesetzt werden und für den Objektzustand, der die minimale Menge der für die Rekonstruktion notwendigen Informationen umfasst, keine Relevanz haben. Diese Situation kann dadurch begründet sein, dass sich Variablen und deren Belegung aus Abhängigkeiten zu anderen Variablen ergeben. So kann eine Klasse neben einer Menge von Objekten, auch deren Anzahl speichern. Die Anzahl ergibt sich aber automatisch durch das Hinzufügen der Objekte und ist für die Wiederherstellung des Objektzustands keine relevante Information. Als konkretes Beispiel kann die Klasse `java.util.HashMap` dienen. Innerhalb einer `HashMap` werden die Werte in einem Array aus `Entry`-Objekten abgelegt. Dabei ist die Klasse `Entry` folgendermaßen definiert.

```
1 class Entry<K,V> implements Map.Entry<K,V> {
2
3   final K key;
4         V value;
5         Entry<K,V> next;
6   final int hash;
7    ...
8
9 }
```

Ein `Entry`-Objekt enthält einen `key`, einen `value`, das nächste Element der Liste und einen Hashwert. Somit wird im generischen Marshaling-Prozess folgendes Schema erzeugt:

```
1 <xs:complexType name="java.util.HashMap_Entry">
2  <xs:sequence>
3   <xs:element name="key" nillable="true" type="xs:anyType"/>
4   <xs:element name="value" nillable="true" type="xs:anyType"/>
5   <xs:element name="next" nillable="true" type="tns:java.util.HashMap_Entry"/>
6   <xs:element name="hash" type="xs:int"/>
7  </xs:sequence>
8 </xs:complexType>
```

Gerade der Zeiger auf das nächste Element verursacht bei der Instantiierung einen riesigen Overhead in der XML-Datei. Durch die Rekursion enthält jedes Element gleichzeitig alle noch folgenden Elemente:

```
1 <EntryElement>
2  <key>Element1</key>
3  <value>Value1</value>
4  <hash>587354687</hash>
5  <next>
6   <key>Element2</key>
7   <value>Value2</value>
```

```
8    <hash>3244534587</hash>
9    <next>
10   ...
11   </next>
12  </next>
13 </EntryElement>
```

Die einzigen Informationen, die aber für die Übertragung relevant sind, sind der `key` und der `value`. Sowohl der Verweis auf das nächste Element, als auch der Hashwert werden bei der Instantiierung der `HashMap` neu erzeugt. Ein Transformationspattern kann nun die unnötigen Informationen ausfiltern und z. B. das in Unterabschnitt 5.2.3 definierte Schema für eine `Map` verwenden.

Ein Transformationspattern für eine Java-Klasse umfasst zum einen die Darstellung der Klasse in XML-Schema, und zum anderen die Logik zur Realisierung der Abbildung und somit für die Erzeugung von XML- und Java-Instanzen aus der jeweiligen anderen Technologie. Die Transformationspattern werden als OSGi-Services registriert, über deren Schnittstellen dann der XML-Schema-Typ der repräsentierenden Klasse abgefragt werden kann, und zur Laufzeit das (Un-)Marshaling für diese Objekte angestoßen wird. Der Service wird mit einer Property registriert, die alle unterstützten Klassen angibt. So kann die Marshaling-Komponente einfach in der Plattform nach einem Transformationspattern-Service für eine bestimmte Klasse suchen. Ist ein solches Pattern vorhanden, wird der Typ dieses Pattern vom Schema-Generator verwendet, andernfalls wird ein Schema durch die generische Generatormethode erzeugt.

Ist das Schema erstellt und der Skeleton-Service bereitgestellt, können nachträglich installierte Pattern nicht mehr verwendet werden. Der Skeleton müsste ansonsten komplett neu erstellt werden, da die WSDL bereits erzeugt ist und verwendet wird. Für zukünftige Skeleton-Bereitstellungen kann das neu installierte Pattern aber verwendet werden.

Bis zu diesem Punkt wurde lediglich beschrieben, wie Java-Klassen durch XML-Schema repräsentiert werden können und wie dieser Vorgang durch den Pattern-Mechanismus effizienter gestaltet werden kann. Um die Effizienz größtmöglich zu steigern, ist es mittels Transformationspattern aber auch möglich, für die Darstellung einer Java-Klasse ein binäres Datenformat zu verwenden. Dazu wird die MTOM-Spezifikation angewandt, die es erlaubt, innerhalb eines XML-Dokuments binäre Daten zu verschicken. So kann ein Transformationspattern für die Klasse `de.med.BPData` folgendes Schema erzeugen:

```
1    <xsd:element name="de.med.BPData" type="xsd:base64Binary"/>
```

Gerade bei komplexen Klassen, die ein aufwendiges Schema erzeugen, kann diese Darstellung für einen erheblichen Effizienzgewinn sorgen. Das binäre Datenformat, das innerhalb des `base64Binary`-Typ verwendet wird, ist nicht festgelegt. Eine einfache Umsetzung bietet der Java-Serialisierungs-Mechanismus. Dieser Mechanismus kann alle Objekte der Klassen, die das `Serializable`-Interface implementieren, in Binärdaten umwandeln und später wieder in die Java-Objekte zurückwandeln. Wird diese Art der Darstellung verwendet, gibt es allerdings eine erhebliche Einschränkung in der Kompatibilität für die Kommunikation zwischen verschiedenen Knoten im Netz. Nur OSGi-Knoten oder in Java implementierte Knoten OSGi-fremder Technologien können

das Format verarbeiten. Dies steht im Gegensatz zu den Anforderungen von Comoros.

Der WS-Attachment-Mechanismus kann neben abgeschlossenen binären Daten auch kontinuierliche Daten verarbeiten. Diese Fähigkeit wird in Comoros dazu verwendet kontinuierliche Datenströme, z. B. `InputStream` und `ObjectInputStream`, in Java zu unterstützen. Während der `InputStream` so genannte Rohdaten verschickt, die potenziell von allen Knoten im Netz verarbeitet werden können, unterliegt der `ObjectInputStream` den gleichen Beschränkungen wie der Java-Serialisierungs-Mechanismus. Ein Überblick ist in Abbildung 6.3 illustriert. Lediglich die Verwendung des XML-Mechanismus ohne binäre Daten kann eine vollständige Kompatibilität zu allen Knoten im Netz gewährleisten.

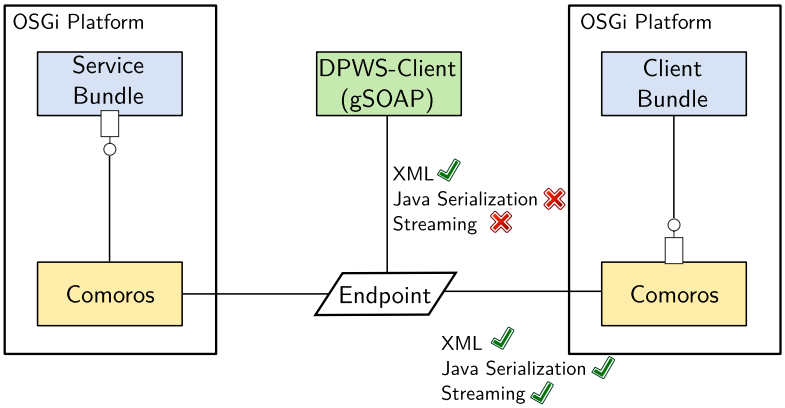


Abbildung 6.3: Einschränkungen der Kompatibilität bei der Verwendung unterschiedlicher Datenformate

6.1.4 (Un-)Marshaling

Der *MarshalingService* bietet Schnittstellen zur Umwandlung zwischen Java- und XML-Instanzen. Dazu werden die Eingaben des Marshaling-Service an den generischen Marshaler innerhalb der Marshaling-Komponente weitergeleitet (siehe Abbildung 6.2), in dem dem die Transformation implementiert ist. Der Ablauf des generischen Marshaling-Prozess, beispielhaft für den Fall der Generierung eines XML-Dokuments für ein Java-Objekt, ist in Abbildung 6.4 abgebildet.

Im ersten Schritt wird innerhalb des generischen Marshalers für das übergebene Java-Objekt in der OSGi-Plattform überprüft, ob ein passendes Transformationspattern existiert. Falls das der Fall ist, wird mittels des Pattern das XML-Dokument erzeugt und der Prozess ist beendet. Falls kein Pattern existiert, muss das Objekt analysiert werden. Dazu erstellt der generische Marshaller als erstes ein leeres XML-Element, das dem übergebenen Objekttyp entspricht. Nun wird der Zustand des Objekts analysiert und die Liste der Instanzvariablen abgefragt. Ist keine Variable vorhanden, ist der Prozess beendet, ansonsten wird für den Inhalt der gefundenen Variable erneut nach einem Pattern gesucht. Falls vorhanden wird der Inhalt der Variable nach den Regeln des Pattern transformiert, dem übergeordneten Element hinzugefügt, und mit dem nächsten Eintrag in der Liste der Instanzvariablen fortgefahren. Steht allerdings kein Pattern zur Verfügung, muss an dieser Stelle für den Variableninhalt der Prozess des generischen Marshalling erneut angestoßen werden. Wenn der Prozess beendet wurde, kann mit der Verarbeitung der

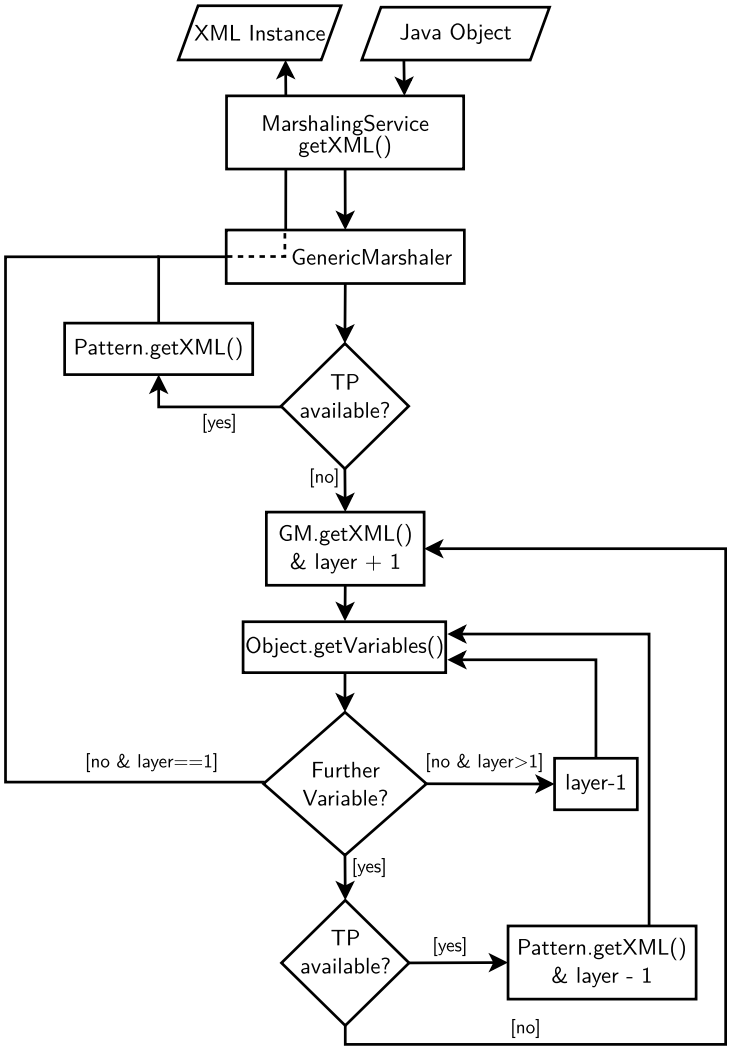


Abbildung 6.4: Ablauf des Marshaling-Prozess

Variablen auf dieser Ebene fortgefahren werden. Jeder Prozess endet genau dann, wenn ein Transformationspattern für das gefundene Element zur Verfügung steht. Da für alle primitiven Datentypen, deren Wrapper, für Arrays und alle Collections ein Pattern zur Verfügung steht, terminiert der Prozess sicher. Jede Java-Klasse ist schlussendlich nur eine Zusammenfassung von diesen Typen.

Durch diesen Prozess wird der vollständige Zustand eines Objekts ausgelesen und in einer XML-Struktur gesichert. Um das Auslesen, und spätere Setzen von Instanzvariablen mit sperrenden Zugriffsmodifikatoren zu ermöglichen, wird innerhalb des generischen Marshalers der Zugriff über die Java-Reflection-API realisiert. Über diese API können Zugriffsbeschränkungen übergangen werden. Falls für die OSGi-Plattform der Java-Security-Manager eingeschaltet ist, so muss dieser durch entsprechende Regeln so konfiguriert werden, dass die Verwendung der Reflection-API nicht eingeschränkt ist. Das

Auslesen des Objektzustands wird in dieser Arbeit als *Ejection* bezeichnet, das Setzen des Zustands als *Injection*.

Ejection

Im Ejection-Prozess kann es vorkommen, dass Java-Objekte verschiedene Instanzvariablen mit den selben Objekten enthalten. Diese Referenzstrukturen müssen abgefangen werden um das unnötige, wiederholte Marshaling des selben Objekts zu verhindern. Dazu werden während des Marshaling-Prozess alle umgewandelten Objekte im XML-Dokument mit einer eindeutigen Identifikationsnummer ausgezeichnet. Alle weiteren Referenzen auf ein solches Objekt werden in XML über das `refid`-Attribut referenziert. Somit wird jedes Objekt nur einmal transformiert und übertragen. Objektreferenzen auf sich selbst, oder zyklische Referenzen werden erkannt und aufgelöst, so dass der Prozess terminieren kann. Aus dem nun fertig erstellten XML-Dokument kann nun mittels *Injection* eine Kopie des ursprünglichen Java-Objekts erstellt werden.

Injection

Für die Erstellung einer Kopie des ursprünglichen Java-Objekts muss im Prozess der *Injection* eine Instanz der entsprechenden Klasse erzeugt werden. Anschließend werden mittels der Reflection-API alle Instanzvariablen mit den Werten aus dem übermittelten XML-Dokument belegt. Falls es sich bei dem Inhalt einer Variable wieder um ein Objekt handelt, muss dieses ebenso erzeugt und mit den übermittelten Werten belegt werden. Variablen aus primitiven Datentypen und Arrays können direkt mit einem Wert belegt werden.

Das Erzeugen einer Objektinstanz ist in der `ObjektFactory` gekapselt. Die Instantiierung von Objekten mit leerem Konstruktor ist nicht weiter schwierig. Falls ein solcher Konstruktor nicht vorhanden ist, wird der Konstruktor mit den wenigsten Argumenten aufgerufen. Dabei werden die Argumente mit Standardwerten belegt. Bei primitiven Zahlenwerten wird die `0` als Standardwert verwendet, boolesche Argumente werden mit `false` belegt und für Arrays werden entsprechende leere Arrays erzeugt. Werden Objekte als Argument erwartet, so wird `null` übergeben. Scheitert die Instantiierung mit dem Wert `null`, und eine Exception wird übermittelt, so versucht die `ObjektFactory` das entsprechende Objekt ebenfalls zu instantiieren. Führt dieser Weg nicht zum Erfolg, so muss der gesamte Marshaling-Vorgang abgebrochen werden. Im Erfolgsfall ist eine Objektinstanz erzeugt worden und die Instanzvariablen können anschließend mit den übermittelten Werten belegt werden.

Der beschriebene Vorgang der Injection ist nur für Klassen notwendig, die nicht durch ein Transformationspattern beschrieben sind. Ist ein solches Pattern vorhanden, so wird in der Implementierung auch die korrekte Erzeugung der Objektkopie übernommen.

Late Binding

In Unterabschnitt 6.1.3 wurde die Vererbungshierarchie von Java-Klassen, und wie diese in XML-Schema abgebildet werden kann, erläutert. Aus Performanzgründen wird in Comoros in der Standardkonfiguration darauf verzichtet die gesamte Vererbungshierarchie einer Klasse darzustellen. In Java kann es aber aufgrund des Vererbungskonzepts vorkommen,

dass zur Laufzeit der Methode ein Objekt einer anderen Klasse übergeben wird, als in der Methodensignatur angegeben ist. Angenommen es existiert eine Klasse `A`, die von einer Klasse `B` erbt. Die Methodensignatur verlangt ein Objekt der Klasse `B` als Eingebeparameter. Zu Laufzeit kann nun aber auch ein Objekt der Klasse `A` übergeben werden, da `A` durch die Vererbung gleichzeitig auch die Typdefinition von Klasse `B` enthält.

In XML-Schema ist es, ebenso wie in Java, möglich Typen einzuschränken oder zu erweitern. In der XML-Instanz können dann jedem Element auch eine Instanz eines erweiternden oder einschränkenden Typen zugewiesen werden. Das folgende Beispiel zeigt das Vorgehen:

```
1 <xsd:element name="Counter">
2  <xsd:complexType>
3   <xsd:sequence>
4     <xsd:element name="number" minOccurs="1" maxOccurs="1" type="xsd:int"/>
5   </xsd:sequence>
6  </xsd:complexType>
7 </xsd:element>
```

Listing 6.1: Schemadefinition einer Java-Klasse

```
1 <Counter>
2  <number xsi:type="xsd:short">5</number>
3 </Counter>
```

Listing 6.2: Überschreiben des im Schema definierten Typs im Instanz-Dokument

Zur Laufzeit wird in der XML-Instanz für das `number`-Element der Datentyp `xsd:short` anstelle des eigentlich definierten Datentyp `xsd:int` verwendet. Um die korrekte Interpretation zu gewährleisten, wird der tatsächliche Typ in der Instanz über das `xsi:type`-Attribut gesetzt. Dieser Vorgang wird *Late Binding* genannt. Comoros unterstützt dieses Konzept. Für eine korrekte Verwendung muss allerdings die Abbildung der gesamten Vererbungshierarchie in der Comoros-Konfiguration aktiviert sein, oder die Vererbungshierarchie an einzelnen Stellen über Transformationspattern geregelt werden. Zwar funktioniert der Prozess auch ohne diese Abbildung, die übertragene XML-Instanz kann dann aber nicht gegen das vorhandene XML-Schema validiert werden. An dieser Stelle muss der *Solution Deployer* entscheiden, ob, oder an welchen Stellen, das Late-Binding-Konzept unterstützt werden muss, oder ob auf eine Schema-Validierung verzichtet werden kann. So muss er die für den Anwendungsfall günstigste Konfiguration und den günstigsten Einsatz von Transformationspattern wählen.

6.2 Event-basierte Kommunikation

Events sind typischerweise Benachrichtigungen über signifikante Zustandsänderungen, die es interessierten Komponenten ermöglichen, auf diese Zustandsänderungen entsprechend zu reagieren. OSGi ermöglicht die Entwicklung von Applikationen, die auf einer Event-basierten Architektur beruhen. Zu diesem Zweck wurde in OSGi der *Event Admin Service* (siehe Unterabschnitt 2.1.3) eingeführt, der die Event-basierte Kommunikation zwischen OSGi-Bundles über das Whiteboard-Pattern realisiert.

Um eine vollständige plattformübergreifende Kommunikation zu gewährleisten, muss folglich auch die Event-basierte Kommunikation unterstützt werden. Durch die Comoros-Kernarchitektur wird der plattformübergreifende Event-Mechanismus nur implizit unterstützt. Wird die lokale Event-basierte Kommunikation über den Event-Admin-Service realisiert, so muss der interessierte Empfänger im Sinne des Whiteboard-Pattern einen `EventHandler`-Service in der Service-Registry registrieren. Schickt der Event-Erzeuger nun ein Event an den Event-Admin-Service, so leitet dieser das Event an alle registrierten Event-Handler-Services weiter. Damit nun eine interessierte Komponente einer entfernten Plattform das lokale Event empfangen kann, besteht mit Mitteln der Kernarchitektur die Möglichkeit, einen Event-Handler-Service zu registrieren und für diesen einen Proxy in der Plattform der Event-Quelle zu registrieren. Der lokale Event-Admin-Service wird das entsprechende Event nun auch an den Proxy-Service senden, so dass schlussendlich auch die entfernte Komponente über das Event informiert wird.

Dieser Ansatz hat mehrere Nachteile: In der RS-Spezifikation ist es vorgesehen, dass ein Service-Anbieter den Service mit speziellen Properties registrieren muss, wenn dieser auch für entfernte Clients zugreifbar sein soll. Dadurch wird aber nicht die Installation eines Proxy-Service auf allen entfernten Plattformen angestoßen. Dies geschieht nur, wenn lokal eine Client-Nachfrage für den entfernten Service besteht. Aufgrund des Whiteboard-Pattern ist in der Event-basierten Kommunikation der Event-Admin in der Rolle des Clients für `EventHandler`-Services. Diese zentrale und nicht anpassbare Komponente bindet sich lediglich an alle lokalen Event-Handler, eine Nachfrage an entferte Event-Handler besteht nicht. Aus diesem Grund würde kein Proxy-Service eines entfernten Event-Handlers in der Plattform der Event-Quelle eingebunden werden. Dies ist auch insofern ein Problem, da der Entwickler genau dieses Verhalten aber erwartet, und somit eine Fehlerquelle geschaffen wird. Dieses Problem kann mit Hilfe der konfigurationsgesteuerten Bereitstellung gelöst werden (siehe Unterabschnitt 5.3.1). Hier werden die Bindungen zwischen Client und Service durch die Übermittlung einer Konfiguration durch den *Solution Deployer* aufgebaut. Eine Beschränkung auf die konfigurationsgesteuerte Bereitstellung würde aber in einer unnötigen Einschränkung der in Tabelle 5.3 vorgestellten effizienten Unterstützung unterschiedlicher Anwendungsfälle resultieren.

Auch wenn die Verwendung des Event-Admin-Service für die Event-basierte Kommunikation in OSGi als Best-Practise gilt, so findet häufig in der Anwendungsentwicklung auch das *Publish/Subscribe*-Pattern (auch *Listener*- oder *Observer*-Pattern genannt) Verwendung. Dieses Pattern kann für die plattformübergreifende Event-basierte Kommunikation allerdings nicht einfach unterstützt werden, da hier eine Call-by-Reference-Semantik umgesetzt wird. In diesem Pattern bietet ein Subjekt, das in der Rolle der Event-Quelle agiert, eine Schnittstelle zur Registrierung von Observer-Objekten. Für eine entfernte Kommunikation muss die Service-Schnittstelle des Subjekts mittels eines Proxys auf der entfernten Plattform angeboten werden. Ruft ein entfernter Client die Proxy-Schnittstelle auf und übergibt ein Observer-Objekt, so wird, aufgrund der Beschränkung auf eine Call-by-Value-Semantik, eine Kopie dieses Objekts angelegt und im originalen Subjekt registriert. Event-Aktualisierungen auf dieser Kopie werden dann aber nicht an den entfernten Client übermittelt. In OSGi selbst werden System-Events über das *Listener*-Pattern versendet, diese sind mit den in der RS-Spezifikation angegebenen Mitteln von

der Verteilung ausgeschlossen.

Neben diesen Problemen, welche die Verteilung von OSGi-Events innerhalb der RS-Spzifikation betreffen, werden durch die erweiterte Architektur von Comoros zusätzliche Anwendungsfälle eingeführt. Diese betreffen die explizite Unterstützung von Altkomponenten und die Unterstützung von Clients und Services OSGi-fremder Technologien. Auch zwischen diesen Komponenten muss die Event-basierte Kommunikation unterstützt werden. Dies ist durch die Kernarchitektur nicht gesichert und daher müssen zusätzliche Mechanismen erschaffen werden.

6.2.1 Anforderungen

Die Anwendungsfälle für die Event-basierte Kommunikation umfassen das plattformübergreifende Versenden von Events zwischen OSGi-Komponenten und das Versenden und Empfangen zwischen OSGi-Komponenten und Komponenten OSGi-fremder Technologien. Der Empfang von Events OSGi-fremder Technologien setzt voraus, dass die entsprechende Technologie Daten in Form von Events verschickt und eine entsprechende Schnittstelle zum Abonnieren anbietet. Im Falle von DPWS-Services wird der WS-Eventing-Mechanismus verwendet. Die OSGi-Plattform, in der diese Events veröffentlicht werden sollen, abonniert die Events des DPWS-Service, führt das Marshaling der Nutzdaten durch, und veröffentlicht anschließend ein lokales OSGi-Event. Soll andersherum ein OSGi-fremder Client OSGi-Events der lokalen Plattform empfangen, so bietet die Plattform eine Event-Schnittstelle in der entsprechenden Technologie an, die der Client dann abonnieren kann. Der zweite Anwendungsfall, das Versenden zwischen zwei OSGi-Plattformen, ist eine Kombination aus den eben beschriebenen Szenarien. Es ergeben sich folgende Anforderungen:

Vollständige Event-basierte Kommunikation über den Event-Admin-Service

Für die Event-basierte Kommunikation wird in OSGi der *Event Admin Service* angeboten. Die plattformübergreifende Kommunikation wird implizit, mit den genannten Einschränkungen, über die Verteilung des `EventHandler`-Service realisiert. Für die Event-basierte Kommunikation in Comoros muss die Verwendung von Altkomponenten sichergestellt werden und die Beschränkung auf den Aufbau der Service-Bindungen durch die konfigurationsgesteuerte Bereitstellung der EventHandler-Proxys aufgehoben werden. Dies wird durch die Einführung einer Komponente erreicht, welche die Vermittlung von plattformübergreifenden Events übernimmt. Clients in den lokalen Plattformen nutzen dann wie gewohnt den lokalen Event-Admin-Service.

Plattformübergreifende Events mittels des Publish/Subscribe-Pattern

Im Zuge der Unterstützung von Altkomponenten muss auch die Verwendung des Publish/Subscribe-Pattern von Comoros angeboten werden. Neben den Altkomponenten ist auch das plattformübergreifende Versenden OSGi-System-Events von der Unterstützung dieses Patterns angewiesen. Interessierte Clients melden sich hierfür direkt als *Listener* am Framework an und müssen auch System-Events entfernter Plattformen empfangen können.

Event-basierte Kommunikation mit OSGi-fremden Komponenten

Die Event-basierte Kommunikation mit OSGi-fremden Technologien umfasst zwei Richtungen, zum einen den Empfang von Events OSGi-fremder Komponenten innerhalb einer OSGi-Plattform, zum anderen die Übermittlung von OSGi-Events an OSGi-fremde Clients. Für den ersten Fall abonniert die Eventing-Komponente der jeweiligen OSGi-Plattform die Event-Schnittstelle der OSGi-fremden Komponente. Dabei werden exakt die Technologien unterstützt, die generell in Comoros zur Kommunikation eingesetzt werden und zusätzlich Mechanismen für eine Event-basierte Kommunikation anbieten. Das Standardprotokoll ist das DPWS, weitere Protokolle, wie CoAP werden als Teil der erweiterten Architektur unterstützt. Sollen Komponenten dieser Technologien wiederum lokale OSGi-Events empfangen, so müssen die jeweiligen Clients die Events über die Schnittstelle der Eventing-Komponente abonnieren.

Minimierung des Datenverkehrs

Ist auf mehreren OSGi-Plattformen jeweils die Eventing-Komponente installiert, so können zwischen diesen Plattformen OSGi-Events ausgetauscht werden. Damit nicht alle Events zwischen allen Plattformen versendet werden, wodurch viele unnötige Nachrichten verschickt werden würden, muss exakt konfiguriert werden können, welche Events für den Versand zu welcher Plattform vorgesehen sind. Dabei wird zum einen konfiguriert, zwischen welchen Plattformen generell eine Event-basierte Kommunikation etabliert wird, und zum anderen werden nur Events mit einem spezifizierten Topic propagiert.

Integration in die Comoros-Umgebung

Die entwickelte Eventing-Komponente wird als OSGi-Bundle im Framework installiert und integriert sich nahtlos in die Comoros-Umgebung. Für das Marshaling der Nutzdaten eines Events bestehen Abhängigkeiten zu der Marshaling-Komponente und deren API und für die Kommunikation wird das DPWS-Bundle verwendet. Die Eventing-Komponente ist optional, Comoros bietet die restliche Funktionalität auch ohne diese Komponente an.

6.2.2 Architektur

Für die Event-basierte Kommunikation wird in Comoros der *Event Converter* eingeführt. Der Event-Converter wird in jeder Plattform, die Teil der Kommunikation ist, genau einmal installiert, und bietet dann Schnittstellen für den Austausch von OSGi-Events zwischen den Plattformen an. Plattformen ohne installierten Event-Converter können die Events entfernter Plattformen nicht empfangen. Eine Übersicht dieses Szenarios ist in Abbildung 6.5 abgebildet. Neben der Kommunikation zwischen OSGi-Plattformen wird vom Event-Converter auch der Austausch von Events zwischen OSGi-Plattformen und OSGi-fremden Knoten realisiert. Der Event-Converter verwendet, wie die gesamte Comoros-Software, das DPWS als Basis-Kommunikationsprotokoll. Daher wird im folgenden für OSGi-fremde Knoten lediglich das DPWS erläutert. Andere Technologien folgen analog.

Der Event-Converter realisiert insgesamt die folgenden Teilaufgaben:

- Empfangen von OSGi-Events

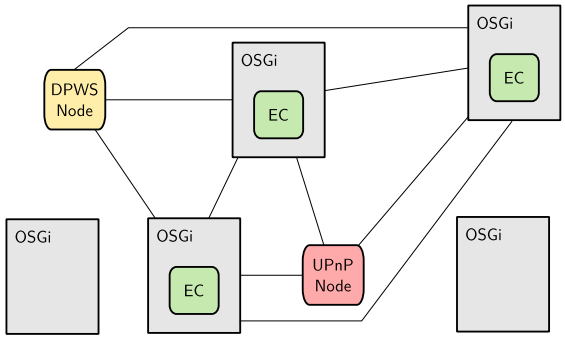


Abbildung 6.5: Event-basierte Kommunikation über den Event-Converter

- Marshaling der OSGi-Events zur Übertragung mittels DPWS
- Anbieten eines DPWS-Service zum Übertragen der umgewandelten Events
- Finden und Abonnieren anderer Event-Converter DPWS-Services
- Finden und Abonnieren von Event-anbietenden DPWS-Services
- Empfangen abonnierter DPWS-Events
- Unmarshalling der DPWS-Events in OSGi-Events
- Versenden der OSGi-Events

Die Teilaufgaben können grundsätzlich in Aufgaben, die auf der Client-Seite anfallen, und Aufgaben, die auf der Server-Seite anfallen, unterteilt werden. Die Architektur des Event-Converters ist nun so aufgebaut, dass er die Aufgaben beider Seiten erfüllen kann. Je nach Anwendungsfall werden die benötigten Teile dynamisch geladen. So ergibt sich für den Event-Converter die in Abbildung 6.6 abgebildete Architektur.

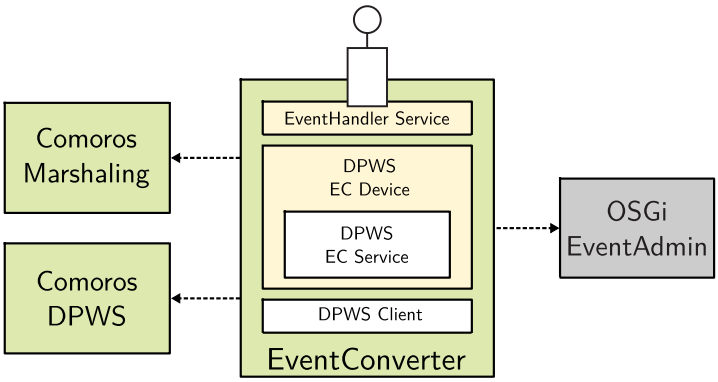


Abbildung 6.6: Die Architektur des Comoros Event-Converter

Der Event-Converter ist ein OSGi-Bundle, das einerseits Abhängigkeiten zu den von Comoros bereitgestellten Marshaling- und DPWS-Komponenten besitzt, und andererseits zu dem von OSGi bereitgestellten Event-Admin-Service. Er besteht aus drei wesentlichen Teilen, einem OSGi-`EventHandler`-Service, einem DPWS-Device mit DPWS-Service und einem DPWS-Client. Soll der Event-Converter lokale OSGi-Events empfangen und

entfernten Komponenten zur Verfügung stellen, so wird lediglich der `EventHandler`-Service und das DPWS-Device instantiiert. Über den Handler werden lokale Events empfangen, die dann über den DPWS-Service von anderen Event-Convertern oder OSGi-fremden Clients empfangen werden. Soll der Event-Converter entfernte Events empfangen und lokal als OSGi-Events veröffentlichen, so wird einzig der DPWS-Client benötigt. Dieser abonniert Events anderer Event-Converter oder normaler DPWS-Services. Die so empfangenen Events können dann lokal veröffentlicht werden. Nimmt der Event-Converter beide Rollen ein, so sind alle Teile zur Laufzeit geladen.

Genau wie in der Kommunikation mittels des Aufrufs von Methoden, gibt es auch in der Event-basierten Kommunikation eine Bereitstellungsphase, eine Kommunikationsphase und eine Deinstallationsphase.

Bereitstellungsphase

In der Bereitstellungsphase müssen Bindungen zwischen den unterschiedlichen Event-Convertern oder mit DPWS-Services/Clients aufgebaut werden. Der Ablauf ist in Abbildung 6.7 illustriert.

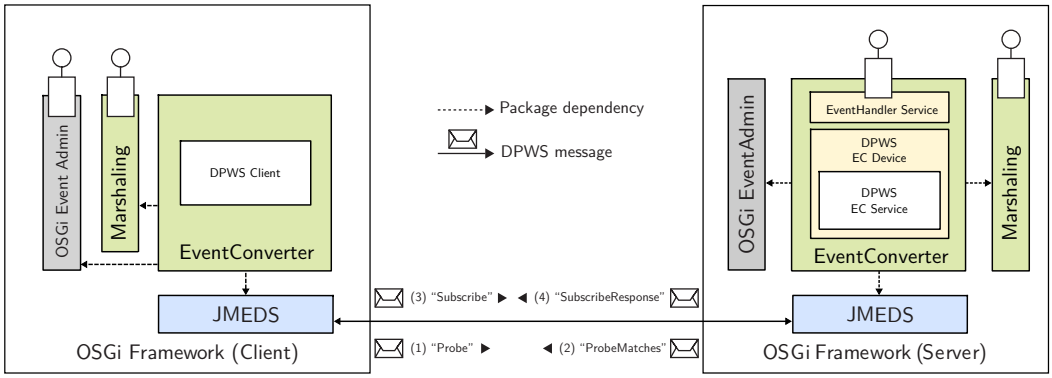


Abbildung 6.7: Der Event-Converter in der Bereitstellungsphase

Der Event-Converter auf der rechten Seite agiert in der Rolle des Event-Verteilers. Seine Aufgabe ist es, lokale OSGi-Events zu empfangen, und interessierten entfernten Clients zur Verfügung zu stellen. Aus diesem Grund hat er in OSGi einen `EventHandler`-Service registriert und ein DPWS-EventConverter-Service gestartet. Der Event-Converter auf der linken Seite agiert in der Client-Rolle und soll empfangene DPWS-Events in OSGi lokal propagieren. Um die Bindung zwischen den beiden Event-Convertern aufzubauen, sendet der Client wie gewohnt eine `Probe`-Nachricht in das Netz (1). Der Event-Converter auf der Server-Seite antwortet mit einem entsprechenden `ProbeMatches` (2). Anschließend – das Auflösen des gefundenen DPWS-Device wird an dieser Stelle nicht weiter beschrieben – versucht der Client die Event-Schnittstelle des DPWS-Event-Converter-Service zu abonnieren. Dazu sendet er eine `Subscribe`-Nachricht an das Device (3). Die `SubscribeResponse`-Nachricht signalisiert das erfolgreiche Abonnieren und die Event-basierte Kommunikation ist vorbereitet.

Kommunikationsphase

Die Kommunikationsphase ist in Abbildung 6.8 dargestellt. Sie folgt unmittelbar auf die Bereitstellungsphase und dauert so lange, bis die Bindung zwischen den jeweiligen Event-Convertern aufgelöst wird.

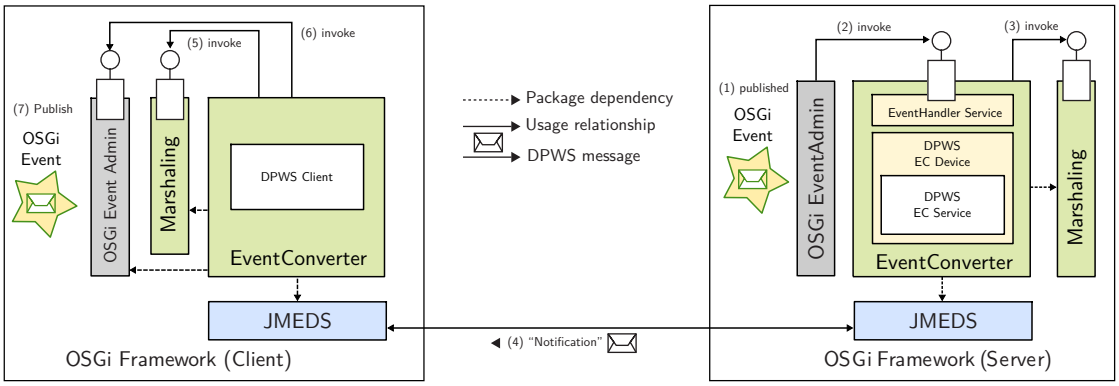


Abbildung 6.8: Der Event-Converter in der Kommunikationsphase

Auf der Server-Seite wird ein OSGi-Event veröffentlicht, das auf die Client-Seite übertragen werden soll (1). Dieser Vorgang wird vom OSGi-Event-Admin durchgeführt. Da der Event-Converter einen passenden `EventHandler`-Service registriert hat, ruft der Event-Admin diese Service auf und übergibt das entsprechende Event (2). Anschließend wird das Marshaling des Events durchgeführt, um dieses mittels DPWS übertragen zu können (3). Das so erzeugt DPWS-Event wird daraufhin über die JMEDS-Instanz in Form einer `Notification`-Nachricht an die Server-Seite verschickt (4). Die JMEDS-Instanz auf der Client-Seite, in ihrer Rolle als Objektadapter, nimmt diese Nachricht entgegen und leitet den Aufruf an den Client weiter. An dieser Stelle muss der Unmarshaling-Prozess angestoßen werden, um wieder ein OSGi-Event zu erzeugen (5). Abschließend ruft der Event-Converter den OSGi-Event-Admin-Service auf (6), so dass dieser das Event an alle Interessenten verteilen kann (7).

Deinstallationsphase

Die Deinstallationsphase läuft im wesentlichen genau so ab, wie die im Bereich der Kommunikation mittels eines Methodenaufrufs (siehe Unterabschnitt 5.1.3). Die Kommunikationsphase wird entweder durch den Wegfall des Event-Converter-Clients oder durch den Wegfall des Event-Converter-Service beendet. Daraufhin werden die entsprechenden `Bye`-Nachrichten versendet und somit die Bindung gelöst.

6.2.3 Aufbau von Kommunikationsmustern

Der Event-Converter muss, im Gegensatz zum Skeleton-Generator und Proxy-Generator, nicht dynamisch auf Veränderungen des Systems reagieren. Die Funktionalität des Event-Converters hängt ausschließlich von der aktuellen Konfiguration ab. Über diese Konfiguration können Konfigurationsmuster erstellt werden, die steuern, welche Events an welche Plattform, beziehungsweise an welche OSGi-fremde Komponente, versendet werden. Auf diese Weise wird der Datenverkehr auf ein Minimum reduziert. Abbildung 6.9 illustriert

ein exemplarisches Szenario zur Event-basierten Kommunikation.

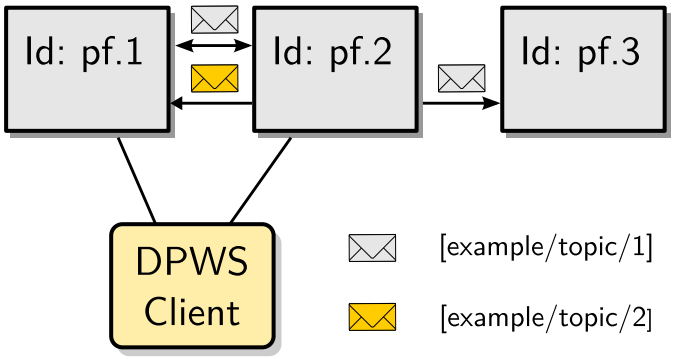


Abbildung 6.9: Aufbau eines Kommunikationsmuster durch eine gezielte Konfiguration

Events mit zwei verschiedenen Topics sollen zwischen drei OSGi-Plattformen und einem DPWS-Client ausgetauscht werden. Für die Kommunikation zwischen den OSGi-Plattformen müssen als erstes Kommunikationsbeziehungen erstellt werden. `Plattform 1` soll Events mit `Plattform 2` austauschen, `Plattform 2` wiederum mit `Plattform 3`. Zwischen `Plattform 1` und `Plattform 3` gbt es keine Beziehung. In einem ersten Schritt muss jeder Plattform eine eindeutige Identifikationsnummer zugeordnet werden. In dem vorliegenden Szenario wird nun der Event-Converter auf den Plattformen `pf.1` und `pf.2` in der Rolle als Event-Verteiler gestartet. Somit bieten die Event-Converter auf diesen Plattformen eine Schnittstelle an, über die entfernte Clients Events abonnieren können. `Plattform 3` empfängt lediglich Events und benötigt daher keinen Event-Converter in dieser Rolle. Auch die beiden anderen Plattformen empfangen Events, so dass alle Event-Converter zusätzlich noch in der Rolle des Clients gestartet werden müssen. Um nicht die Events aller Plattformen zu empfangen, wird der Event-Converter so konfiguriert, dass er nur ausgewählte Event-Converter-Services abonniert. Im konkreten Beispiel bindet sich der Event-Converter auf `Plattform 3` an den Event-Converter auf `Plattform 2` und lauscht dort auf auftretende Events. Die Plattformen `pf.1` und `pf.2` abonnieren sich gegenseitig. Zu diesem Zeitpunkt ist das grundsätzliche Kommunikationsmuster aufgebaut. Um abschließend die Event-basierte Kommunikation noch auf Events bestimmter Topics zu beschränken, wird auf den Plattformen noch ein LDAP-Filter hinterlegt. In dem Beispiel bietet `Plattform 2` Events mit den Topics `[example/topic/1]` und `[example/topic/2]` an. Der `EventHandler`-Service in dem Event-Converter dieser Plattform wird nun mit einem LDAP-Filter registriert, der genau diese beiden Topics filtert.

Ungelöst bis zu dieser Stelle ist das Problem, dass `Plattform 2` die Events mit den Topics `[example/topic/1]` und `[example/topic/2]` für andere Plattformen anbietet, `Plattform 3` aber nur Events mit dem Topic `[example/topic/1]` empfangen will. Eine Filterung beim Empfänger scheidet aus, da dann viele Event-Nachrichten unnötig versendet werden würden und die Netzauslastung damit erhöht würde. Also muss der Empfänger bei der Event-Quelle nicht ein generelles Interesse an Events sondern das Interesse an bestimmten Events anmelden. Dazu führt die WS-Eventing-Spezifikation [Box06] im Absatz 3 ein Filter-Element in der `Subscribe`-Nachricht ein. Allerdings wird dieser

Filter-Mechanismus in der DPWS-Spezifikation [Nix09b] so weit eingeschränkt, dass er nur für die Adressierung an die korrekte SOAP-Action verwendet werden kann. Weitergehende Filter-Definitionen sind hier nicht erlaubt. Die DPWS-Spezifikation bietet aber die Möglichkeit in der `Subscribe`-Nachricht beliebige zusätzliche Elemente zu definieren. Comoros fügt der Nachricht daher das `<com:Topic>`-Element hinzu. In diesem Element kann der Client die gewünschten Event-Topics angeben. Fortan wird er nur über Events benachrichtigt, die dem Topic-Filterausdruck entsprechen.

Hinsichtlich der Interoperabilität mit anderen DPWS-Implementierungen bleibt festzuhalten, dass das `<com:Topic>`-Element von diesen nicht interpretiert wird, die übrige Funktionalität dadurch aber nicht beeinträchtigt ist. Da bei der plattformübergreifenden Event-basierten Kommunikation auf allen Plattformen die Comoros-Middleware verwendet wird, wird somit auf allen Knoten die DPWS-Implementierung JMEDS verwendet. Dadurch ist sichergestellt, dass das `<com:Topic>`-Element interpretiert wird.

Neben dem Aufbau von Kommunikationsmustern durch eine entsprechende Konfiguration, soll es für den Entwickler auch möglich sein, das Ziel eines Events aus der Anwendung heraus zu steuern. Innerhalb der aufgebauten Kommunikationsbeziehungen ist es dann möglich, für jedes einzelne Event zu entscheiden, an welche Plattform es versendet wird. Soll in dem vorliegenden Beispiel nun ein einzelnes Event mit dem Topic `[example/topic/1]` von `Plattform 2` nur an `Plattform 3` aber nicht an `Plattform 1` gesendet werden, so muss in der Anwendung dem Event die Property `org.ws4d.dpws.event.destination` mit dem Wert `{pf.3}` (Liste von Plattform-Identifikationsnummern) hinzugefügt werden.

Für OSGi-fremde Komponenten, hier DPWS-Clients, gelten die gleichen Regeln. Ein DPWS-Client kann die Event-Schnittstelle jedes Event-Converters abonnieren und die Events empfangen. Lediglich hinsichtlich der Filterung von Event-Topics in der `Subscribe`-Nachricht gelten die Beschränkungen bezüglich der Interoperabilität. JMEDS-Clients können die Filterung vornehmen, andere DPWS-Implementierungen nur, wenn sie auch das eingeführte `<com:Topic>`-Element der Nachricht hinzufügen.

6.2.4 Das Publish/Subscribe-Pattern

Das *Publish/Subscribe-Pattern* ist ein klassisches Beispiel für die Umsetzung einer Call-by-Reference-Semantik, die von der RS-Spezifikation in einer verteilten Umgebung nicht unterstützt wird. Abbildung 6.10 illustriert das Problem.

In einer lokalen Umgebung registriert ein Client im Subjekt über die Observer-Schnittstelle eines Services ein sogenanntes Observer-Objekt. Dieses Observer-Objekt besitzt eine *Callback*-Methode, die im Falle eines Events vom Subjekt aufgerufen wird. So erhält der Client, und alle weiteren Parteien, die ein Observer-Objekt im Subjekt registriert haben, die vom Subjekt propagierten Informationen. Wenn der Service nun über die normalen Mechanismen auf eine entfernte Plattform verteilt wird, so wird in dieser Plattform ein Proxy für den originalen Service erstellt, der dann auch die Schnittstelle zur Registrierung von Observer-Objekten enthält. Ruft nun ein Client die Schnittstelle auf, so wird eine Kopie des Observer-Objekts erstellt und an den originalen Service übergeben. Im Proxy selbst werden keine Observer-Objekte gespeichert. Wird nun auf der Kopie die *Callback*-Methode aufgerufen, geht der Methodenaufruf ins Leere.

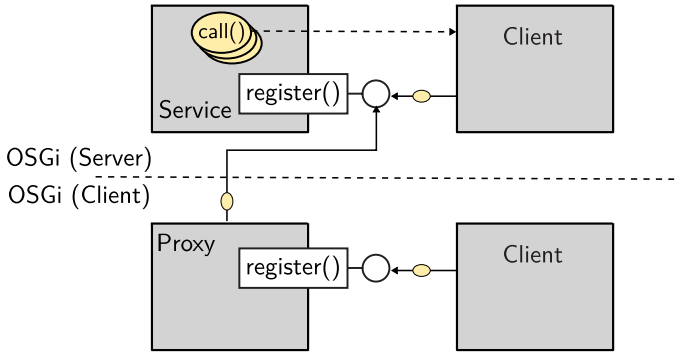


Abbildung 6.10: Das Publish/Subscribe-Pattern in einer verteilten Umgebung

Aufgrund dieser Problematik kann die Verwendung des Publish/Subscribe-Pattern in verteilten Umgebungen nicht transparent unterstützt werden. Da aber OSGi-System-Events auf diese Weise versendet werden, und die Unterstützung von Altkomponenten eine zentrale Anforderung an Comoros ist, müssen Events über diesen Mechanismus auch in entfernten Plattformen propagiert werden können.

Server-seitige Unterstützung

Der Event-Converter muss, wie beschrieben, nicht selbstständig auf dynamische Veränderungen der Umgebung reagieren, sondern setzt die Funktionalität nur aufgrund der aktuellen Konfiguration um. Auch bei der Unterstützung des Publish/Subscribe-Pattern übernimmt der Solution-Deployer die Aufgabe der Identifikation dieser Pattern und der Erstellung einer geeigneten Konfiguration. Abbildung 6.11 stellt nun die entwickelte Lösung zur Unterstützung dieses Event-Mechanismus mit dem Event-Converter dar.

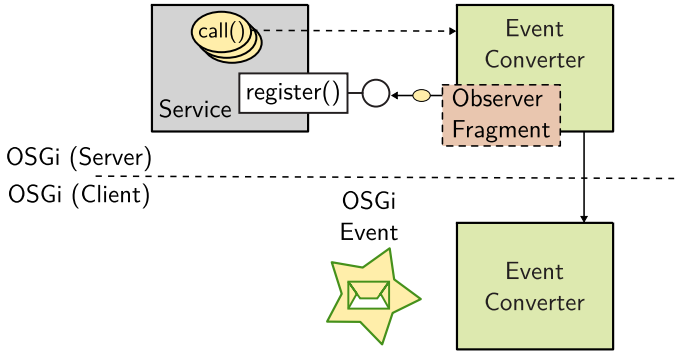


Abbildung 6.11: Events über das Publish/Subscribe-Pattern mit Hilfe des Event-Converters

Für die Unterstützung des Publish/Subscribe-Pattern wird keine Konfiguration im eigentlichen Sinne hinterlegt, vielmehr muss der Solution-Deployer Java-Code im Framework installieren. Dies geschieht in Form eines so genannten *Observer-Fragment*. Ein Observer-Fragment ist ein OSGi-Fragment-Bundle (siehe Unterabschnitt 2.1.3), das zur Laufzeit dem Klassenpfad des Event-Converters hinzugefügt werden kann. Es hat die Aufgabe, ein Observer-Objekt in dem entsprechenden Service zu registrieren. Fortan empfängt der Event-Converter Events über diese Schnittstelle, die dann in das für OSGi

spezifizierte Event-Format überführt werden und schlussendlich über die bekannten Mechanismen an entfernte Plattformen verteilt werden können.

Client-seitige Unterstützung

Die Client-seitige Unterstützung des Publish/Subscribe-Pattern ist wesentlich komplexer als die Server-seitige Unterstützung, da hier das Problem der Call-by-Reference-Semantik zum Tragen kommt. Der grundsätzliche Aufbau ist in Abbildung 6.12 zu sehen.

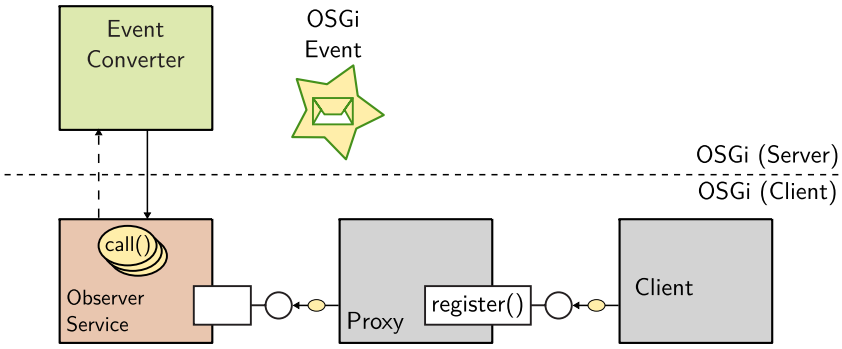


Abbildung 6.12: Client-seitige Unterstützung des Publish/Subscribe-Pattern

Ein Client ruft wie gewohnt den Proxy-Service mit einem Observer-Objekt auf. Damit dieses Objekt nicht als Kopie an den originalen Service weitergeleitet wird, muss der Solution-Deployer in den normalen Prozessablauf eingreifen. Dies geschieht durch die Installation eines *Observer Service* in der lokalen OSGi-Plattform, an den der Aufruf der Observer-Schnittstelle weitergeleitet wird. Die Installation zusätzlichen Java-Codes erfolgt, im Gegensatz zu den Fragmenten auf der Server-Seite, in Form eines Bundles mit OSGi-Service. Aufgrund der unterschiedlichen Arten von Proxys (Dynamische Proxys und Proxy-Bundles), ist eine Realisierung über Fragmente nicht möglich.

Somit ergibt sich folgender Ablauf für die Client-seitige Unterstützung des Publish/Subscribe-Pattern: Wird in einem Proxy eine beliebige Methode aufgerufen, muss überprüft werden, ob es sich bei dieser Methode um eine Observer-Schnittstelle handelt. Dies geschieht durch eine Abfrage der Service-Registry nach passenden Observer-Services. Observer-Services sind mit einer Property, welche die Signatur der Observer-Schnittstelle angibt, registriert, so dass eine Suche in der Registry einfach realisiert werden kann. Ist die Suche erfolgreich, so werden die Aufrufe dieser Methode an den Observer-Service weitergeleitet. Im Observer-Service werden nun einerseits die Observer-Objekte verwaltet, und andererseits die Events des entsprechenden EventConverter einer entfernten Plattform abonniert. Wird ein Event empfangen, so werden die *Callback*-Methoden der verwalteten Observer-Objekte aufgerufen.

System-Events

Für den Empfang von System-Events werden in einer OSGi-Plattform vom Client *Listener* direkt im Framework registriert. Sollen diese Events nun auch auf der entfernten Plattform veröffentlicht werden, so ergibt sich die Besonderheit, dass kein Proxy für die lokalen Framework-Schnittstellen im entfernten Framework existiert. Somit kann die beschriebene Client-seitige Unterstützung nicht realisiert werden. Allerdings können die System-Events

über den Server-seitigen Mechanismus zur Unterstützung des Publish/Subscribe-Pattern als OSGi-Events mittels des Event-Admin-Service in anderen Plattformen veröffentlicht werden. Interessierte Clients können diese dann über entsprechende `EventHandler`-Services empfangen. Der Event-Converter ist für alle System-Events bereits mit entsprechenden Observer-Fragmenten ausgestattet, so dass der plattformübergreifende Empfang dieser Events nur aktiviert werden muss.

Fazit

Insgesamt ist der Mechanismus zu Unterstützung des Publish/Subscribe-Pattern sehr komplex. Für den Entwickler entsteht mit der Entwicklung der Observer-Fragmente und der Observer-Services ein hoher Entwicklungsaufwand. Weiterhin ist die Performanz des gesamten Systems stark beeinträchtigt, da bei jedem Serviceaufruf nun im Framework anhand der installierten Observer-Services entschieden werden muss, ob die aufgerufene Methode eine Observer-Schnittstelle ist. Zusammenfassend steht also einem umfangreichen Entwicklungs- und Konfigurationsaufwand und einem Performanzverlust die Unterstützung eines Entwurfsmusters für Events gegenüber, das von der OSGi-Allianz nicht für die Verwendung empfohlen wird. Im Zuge der Anforderung einer vollständigen Unterstützung von Altkomponenten ist dieser Ansatz dennoch in Comoros integriert, allerdings standardmäßig deaktiviert. Der Client- und Server-seitige Mechanismus kann dabei separat aktiviert werden, um dem Entwickler eine möglichst feingranulare Entscheidung zu überlassen, welche Art der Unterstützung des Publish/Subscribe-Pattern für ihn sinnvoll ist.

6.3 Unterstützung von Legacy-Systemen

Der Aufbau einer verteilten Service-Infrastruktur über die OSGi-Remote-Services verlangt von jedem Service und Client, dass er bewusst für eine verteilte Nutzung entwickelt wurde. Das beinhaltet vor allem ein fest vorgegebenes Modell für die Implementierung. Services die für eine entfernte Nutzung vorgesehen sind müssen mit fest definierten Properties registriert werden, und auch die Suche nach entfernten Services verlangt einen speziellen Suchparameter. Diese Vorgaben verhindern die Integration von bestehenden, ursprünglich für den lokalen Gebrauch vorgesehenen Komponenten in verteilte Umgebungen. Die transparente Unterstützung dieser Altkomponenten in verteilten Umgebungen ist aber eine der Schlüsselanforderungen von Comoros (siehe Unterabschnitt 3.3.2 und Kapitel 4). An OSGi-Services und Clients müssen, für die Integration in verteilte Umgebungen, demnach keinerlei Modifikationen vorgenommen werden.

6.3.1 Anforderungen

In gewachsenen Strukturen eines OSGi-Umfelds liegen Altkomponenten in großer Menge vor. Die Weiterverwendbarkeit dieser Komponenten in verteilten Umgebungen bietet demnach einen großen Mehrwert für den Anwendungsentwickler. Es gelten folgende Anforderungen:

Integration von Altkomponenten in ein verteiltes System ohne Modifikationen

Altkomponenten sind Clients und Services die ursprünglich für die Verwendung in lokalen OSGi-Plattformen entwickelt wurden, und liegen dem *Solution Deployer* meist nur

in binärer Form vor, so dass keine Modifikationen im Code möglich sind. Mit Hilfe von Comoros können diese Komponenten dennoch transparent in ein verteiltes System integriert werden, das sonst aus Komponenten besteht, die explizit unter Beachtung der RS-Spezifikation für die Erstellung einer verteilten Anwendung entwickelt wurden.

Ausschluss von nicht kompatiblen Komponenten

Da Altkomponenten für eine lokale OSGi-Umgebung entwickelt wurden, unterliegen sie keinen Einschränkungen. In verteilten Umgebungen, wie sie durch Comoros aufgespannt werden, gelten aber solche Einschränkungen, wie z. B. die Verwendung einer Call-by-Value-Semantik. Komponenten, die diesen Regeln widersprechen, können nicht Teil einer verteilten Umgebung werden, und werden dementsprechend ausgefiltert.

Konfiguration bereitzustellender Services

Registrierte Services innerhalb von Altkomponenten sind nicht mit zusätzlichen Properties belegt, welche die Bereitstellung in verteilten Umgebungen steuern. Damit nicht einfach alle Services für die entfernte Nutzung aktiviert werden, müssen Zusatzinformationen in Form von Konfigurationen hinterlegt werden. Über diese wird sowohl das Importieren, als auch das Exportieren von Services aus Altkomponenten gesteuert.

6.3.2 Architektur

Für die Integration von Alt-Systemen wird in Comoros keine separate Komponente entwickelt. Vielmehr wurden für die Umsetzung dieser Anforderung bereits Designentscheidungen in der Kernarchitektur entscheidend beeinflusst. Alle Grundlagen wurden dort, oder in den neu hinzugefügten Komponenten der erweiterten Architektur gelegt, so dass an dieser Stelle nur eine kurze Zusammenstellung der in diesem Zusammenhang stehenden Architektur-Elemente folgt.

Unterabschnitt 5.3.3 beschreibt die Problematik, wenn sich Service-Interface und Service-Implementierung in einem Bundle befinden. Diese Situation ist in Komponenten, die ursprünglich für den lokalen Gebrauch entworfen wurden nicht ungewöhnlich. Die Generierung der Service-Interfaces auf der Client-Seite im Falle einer Integration in eine verteilte Umgebung ist hier die vorgestellte Lösung. Erst dadurch wird die wichtigste Voraussetzung für einen Plug-and-Play-Mechanismus von Altkomponenten in verteilte Umgebungen erfüllt, die Auflösung der Abhängigkeiten des Clients zu den Service-Interfaces. Ansonsten bliebe der Client im Zustand `installed`.

Der zweite Punkt für die Realisierung eines solchen Mechanismus wird durch die generische Marshaling-Komponente, eingeführt in Abschnitt 6.1, erreicht. Diese Komponente erhöht die Menge der Unterstützten Datentypen, so dass dadurch die Menge der unterstützten Altkomponenten signifikant erhöht wird. Vorher konnte nicht von einer relevanten Unterstützung gesprochen werden, da sich der Entwickler einer lokalen OSGi-Komponente kaum auf die Verwendung der in der RS-Spezifikation definierten Menge von Datentypen beschränkt. Die erweiterte Marshaling-Komponente übernimmt auch den Ausschluss von nicht kompatiblen Altkomponenten.

Der in Unterabschnitt 5.3.1 vorgestellte Mechanismus zur konfigurationsgesteuerten Bereitstellung von Service-Proxys wird auch dazu verwendet, entfernten Clients die Nutzung von Altkomponenten-Services zu ermöglichen. Durch Angabe des `Bundle Symbolic`

`Name` und des `Service Interface` auf der Server-Seite für den Export, und durch Angabe der `Plattform Id`, des `Bundle Symbolic Name` und des `Service Interface` auf der Client-Seite für den Import, können Bindungen zwischen entfernten Clients und Services etabliert werden. Durch die Konfiguration werden dabei abstrakte Kommunikationsbeziehungen definiert, die erst durch die Installation eines passenden Service aufgebaut werden. Dadurch wird der Plug-and-Play-Mechanismus für Altkomponenten weiter unterstützt.

Services, die im Sinne der RS-Spezifikation verteilt wurden, werden mit einer bestimmten Property registriert, die den Service als Proxy-Service auszeichnet. Die RS-Spezifikation sieht nun vor, dass diese Services nur von solchen Clients verwendet werden können, die explizit einen entfernten Service anfragen. Ein Legacy-Client kann somit niemals einen so bereitgestellten entfernten Service nutzen. Da dieses Verhalten im Widerspruch zu einer transparenten Integration von Altkomponenten steht, kann es durch Übermittlung einer entsprechenden Konfiguration deaktiviert werden.

6.4 Integration nativer Geräte und Services

Comoros unterstützt die komfortable und effiziente Anwendungsentwicklung in verteilten, OSGi-basierten Dienste- und Gerätesystemen. Die Integration aller Arten von Geräten ist somit ein integraler Bestandteil der Comoros-Middleware. Aus der Sicht von OSGi und Comoros lassen sich die Geräte in zwei Klassen unterteilen. Einerseits in Geräte, die bereits eine SOA-fähige Schnittstelle anbieten, andererseits in Geräte älterer Technologien, die z. B. nur eine serielle Kommunikation über Rohdaten anbieten. Beide Klassen von Geräten müssen in die OSGi-Plattform integriert und mittels Comoros Teil einer plattformübergreifenden Kommunikation werden.

6.4.1 Integration von SOA-Geräten

Bei der Integration von SOA-fähigen Geräten in OSGi kann die OSGi-Plattform sowohl die Rolle des Clients, als auch die Rolle des Servers einnehmen. Abbildung 6.13 zeigt ein OSGi-Bundle, das einerseits als Client auf einen DPWS- und einen UPnP-Service zugreift, und andererseits einen Service anbietet, der von einem DPWS-Client aufgerufen wird. Im folgenden werden erneut nur DPWS-Geräte betrachtet, die beschriebenen Konzepte sind aber auch auf andere Technologien, wie eben UPnP, anzuwenden.

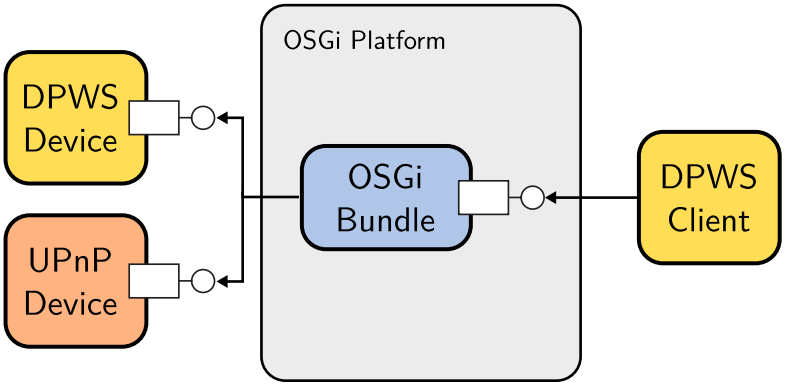


Abbildung 6.13: Zusammenspiel von OSGi und OSGi-fremden Komponenten

Eines der Hauptprobleme in diesem Szenario ist erneut die Heterogenität der verwendeten Datenformate. In der bisher betrachteten OSGi-zu-OSGi-Kommunikation wurden von einem OSGi-Client lediglich speziell angepasste DPWS-Skeletons aufgerufen. Diese Skeletons sind spezielle Abbildungen eines OSGi-Service und verwenden demnach nur eine eingeschränkte Menge der möglichen XML-Schema Datentypen (siehe Unterabschnitt 5.2.3 und Abschnitt 6.1). Diese Einschränkung gilt für generelle DPWS-Services nicht. Sollen nun diese generellen DPWS-Geräte (Services und Clients) in einer verteilten OSGi/DPWS-Umgebung partizipieren, so ergibt sich eine neue Form der Datenformat-Abbildung.

Anforderungen

Da die Kommunikation zwischen OSGi-Komponenten und DPWS-Komponenten durch Comoros bereits grundsätzlich realisiert ist, handelt es sich bei der Integration genereller DPWS-Geräte um ein Problem der Datenformat-Abbildung. Ausgehend von dem Anwendungsfall, dass bestehende, semantisch kompatible Services und Clients miteinander kommunizieren, ist die Abbildung der Datenformate so komplex, dass die Entwicklung eines sinnvollen generischen und automatischen Ansatzes sehr schwierig ist. Daher wird diese Abbildung mit Hilfe eines Werkzeugs realisiert, dem *Comoros.MappingTool*. Mit diesem Werkzeug kann der Entwickler das Daten-Mapping zwischen OSGi- und DPWS-Komponenten graphisch definieren und dadurch die Generierung von passenden Transformationspattern veranlassen. An das Werkzeug werden die folgenden Anforderungen gestellt:

Einlesen von DPWS-Services Das Comoros.MappingTool kann zur Laufzeit im Netz befindliche DPWS-Services auffinden und die Operationen mit ihren Datenstrukturen, wie in der WSDL definiert, darstellen. Liegt eine WSDL bereits zur Entwicklungszeit vor, so kann diese auch eingelesen werden, ohne dass der DPWS-Service erreichbar ist.

Einlesen von OSGi-Services Genau wie DPWS-Services, kann das MappingTool auch OSGi-Services in den jeweiligen Plattformen auffinden und die im Interface definierten Methoden und Datenstrukturen darstellen. Interfaces oder ganze API-Bundles können auch zur Entwicklungszeit im Werkzeug geladen werden. Verwenden Methoden als Ein- oder Ausgabeparameter komplexe Klassen, so müssen diese Klassendefinitionen vorliegen, ansonsten kann der Service nicht vollständig dargestellt werden.

Datenformat-Abbildung Das Zuweisen von XML-Elementen – in der WSDL des DPWS-Service definiert – und Java-Objekten – im Interface des OSGi-Service definiert – wird visuell unterstützt. Für die Abbildung werden passende Elemente aus beiden Seiten selektiert und miteinander verbunden.

Generierung von Transformationspattern Aus den definierten Abbildungen werden Transformationspattern generiert, die anschließend als Bundle vorliegen. Diese verwenden die Schnittstellen der Comoros-Marshaling-Komponente und können daraufhin im

Framework installiert werden. Innerhalb des Pattern ist zusätzlich definiert, welches OSGi-Service-Interface durch die Abbildung angesprochen wird. Mit Hilfe dieser Information kann bei einer Anfrage an einen Service mit diesem Interface von Comoros ein Proxy in Abhängigkeit zu dem entsprechenden generellen DPWS-Service erstellt werden.

Parametrisierung der Abbildungen Die definierten Abbildungen müssen einzeln parametrisiert werden können. So können beispielsweise arithmetische Funktionen innerhalb einer Abbildung ausgeführt und so die übertragenen Werte angepasst werden. Weiterhin können gezielte Typumwandlungen vorgenommen werden.

Definition von $N-to-N$ Abbildungen Mehrere Elemente können auf ein einzelnes Element abgebildet werden. Die übertragenen Werte werden über eine definierte Berechnung aggregiert. Ebenso kann ein einzelnes Element auf beliebig viele Elemente abgebildet werden. Auch hier kann eine Berechnung selbst definiert werden, ansonsten wird der übertragene Wert repliziert und an alle in Beziehung stehende Elemente verteilt.

Definition einer OSGi-Service-API Mit Hilfe des Werkzeugs muss anhand einer vorliegenden WSDL ein OSGi-Service-Interface erstellt werden können. Dazu werden die Elemente der WSDL auf neu definierte Java-Elemente abgebildet. Dadurch entsteht ein Service-Interface, das vom Tool generiert wird. Dieses Interface kann nun vom Entwickler verwendet werden um einen passenden OSGi-Client für die Benutzung des DPWS-Service zu implementieren.

Architektur

Die Architektur zur Integration von SOA-fähigen Geräten besteht im wesentlichen aus dem Werkzeug Comoros.MappingTool, das in einer ersten Version in der studentischen Arbeit von Trenkmann [Tre14] implementiert wurde, und den davon erzeugten Komponenten, die anschließend im OSGi-Framework installiert werden. Bei den Komponenten handelt es sich im einzelnen um ein Transformationspattern (siehe Unterabschnitt 6.1.3) und um eine Komponente zur Abbildung von OSGi-Suchanfragen. Abbildung 6.14 illustriert in einer Übersicht das Zusammenspiel aller beteiligten Komponenten.

Auf der obersten Ebene befindet sich das Comoros.MappingTool. Das Werkzeug kann die Service-Beschreibungen der DPWS-Geräte (WSDL) und der OSGi-Bundles (Interface) sowohl zur Laufzeit als auch zur Entwicklungszeit einlesen. In einer WSDL befinden sich die XML-Schema-Definitionen entweder innerhalb des `types`-Element oder innerhalb einer weiteren importierten WSDL. Die Java-Datentypen können direkt aus den Definitionen innerhalb des Service-Interface ausgelesen werden, oder über Klassen die an dieser Stelle importiert werden. Der Benutzer erstellt nun innerhalb des Werkzeugs die Abbildungen zwischen den unterschiedlichen Datentypen. Die so definierten Regeln werden in Form eines Transformationspattern automatisch generiert, und diese anschließend im entsprechenden Framework installiert. Für die Nutzung des DPWS-Service muss nun noch eine Bindung zwischen diesem Service und dem OSGi-Client aufgebaut werden. OSGi-Clients suchen nach Service-Interfaces, die keine direkte Entsprechung in der DPWS-Welt haben. Das Comoros.MappingTool kennt aber sowohl die WSDL des DPWS-Service

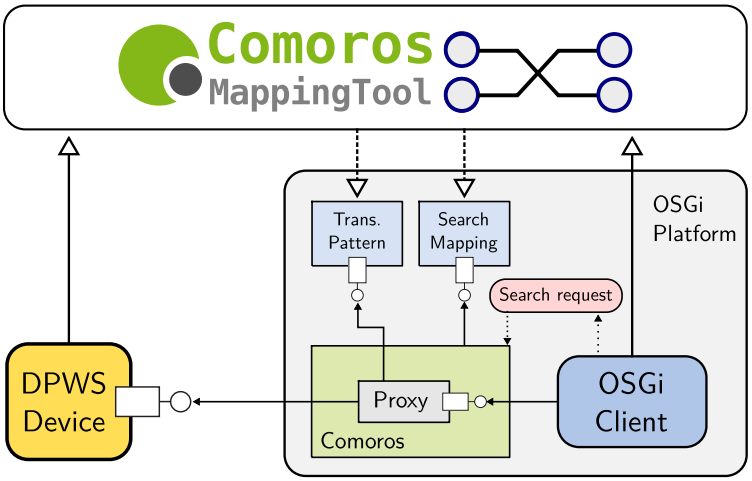


Abbildung 6.14: Architektur zur Integration von SOA-fähigen Geräten

als auch das gesuchte Java-Interface. Mit diesen Information generiert das Werkzeug eine Komponente, die eine Zuordnung zwischen Java-Interface und der innerhalb einer WSDL definierten Services verwaltet. Die Suchanfrage eines OSGi-Clients kann so auch generellen DPWS-Services zugeordnet und somit ein entsprechender Proxy instantiiert und gestartet werden. Der Aufruf des Clients folgt nun den bekannten Mustern.

Der umgedrehte Anwendungsfall, ein DPWS-Client möchte einen OSGi-Service nutzen, erfolgt analog mit umgedrehten Rollen.

Mapping der Datentypen Im Comoros.MappingTool werden die Abbildungen zwischen den Datentypen manuell realisiert. Damit die Abbildungen syntaktisch korrekt sind, muss der Benutzer Regeln einhalten. Eine Abbildung des Datentyp `xsd:string` auf den Java Datentyp `short` darf z.B. nicht erlaubt werden, da es zur Laufzeit zu einer `ClassCastException` führen würde.

Regeln für die Abbildungen von Java-Datentypen auf XML-Schema-Typen wurden bereits ausführlich in Unterabschnitt 5.2.3, Tabelle 5.2 vorgestellt. Einige der Abbildungen im Bereich der simplen Datentypen sind symmetrisch und können somit in umgekehrter Richtung verwendet werden. Bei den simplen Datentypen unterscheidet XML-Schema zwischen den *built-in* und den *use-derived* Typen. Die *built-in* Typen sind in der XML-Schema-Spezifikation definiert und sind die atomaren Bausteine zum Erstellen beliebiger komplexer Datentypen. Es wird zwischen *ur types*, *built-in primitive types* und *built-in derived types* unterschieden. Die Menge der *ur types* besteht lediglich aus den allgemeinem Datentyp `anyType` und dessen Einschränkung `anySimpleType`. Von diesen Typen sind die *built-in primitive types* abgeleitet, die wiederum Basis der *built-in derived types* sind.

use-derived Typen gehören auch zu den simplen Datentypen, wurden aber vom Entwickler selbst definiert. Sie basieren auf den *built-in* Typen und schränken diese weiter ein oder erweitern diese. Somit muss bei der Validierung einer Abbildung eines *use-derived* Typen auf einen Java-Datentyp der Basis-Datentyp in Betracht gezogen werden. Das folgende Beispiel definiert ein Passwort aus einem simplen Typen, der den Datentyp

`string` erweitert. Er schränkt die Länge des Passworts auf 15 Zeichen ein. Das sichere Passwort wiederum fügt mit einer minimalen Länge von 8 Zeichen eine weitere Restriktion ein.

```
<xs:simpleType name="password">
  <xs:restriction base="xs:string">
    <xs:maxLength value="15"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="securePassword">
  <xs:restriction base="cst:password">
    <xs:minLength value="8"/>
  </xs:restriction>
</xs:simpleType>
```

Sowohl `password`, als auch `securePassword` sind vom Datentyp `string` abgeleitet, so dass dieser Datentyp für eine Validierung verwendet wird.

Aus diesen Überlegungen heraus müssen für eine Validierung der durch den Benutzer definierten Abbildungen lediglich Regeln für die *built-in* Typen definiert werden, zusammengefasst in Tabelle 6.2. Bei den Java-Datentypen ist die für Comoros insgesamt gültige Beschränkung auf das CDC-Profil zu beachten.

Tabelle 6.2: Regeln für die Abbildung von simplen *built-in* XML-Datentypen auf Java-Datentypen

XML-Schema	*Java*	*Bemerkung*
anySimpleType	java.lang.String	Der Datentyp `anySimpleType` ist Basis aller anderen simplen Datentypen. Da der textuelle Inhalt jedes XML-Elements als `String` repräsentiert werden kann, wird durch die eingeführte Regel keine Beschränkung vorgenommen.
string	java.lang.String	
normalizedString	java.lang.String	Repräsentation eines `String` ohne *carriage return*, *line feed* und Tabulatoren.
token	java.lang.String	Ein `normalizedString` ohne führende oder angehängte Leerzeichen und ohne sich wiederholende Leerzeichen.
int	int	
short	short	
float	float	
double	double	
boolean	boolean	
decimal	double	`decimal` beschreibt einen unendlichen Wertebereich und ist somit in Java nicht korrekt darstellbar. Eine sinnvolle Abbildung bietet aber der Datentyp `double`, da so die allermeisten Anwendungsfälle abgedeckt sind.

weiter auf der nächsten Seite

XML-Schema	*Java*	*Bemerkung*
integer	java.Math.BigInteger	Im Gegensatz zu reelen Zahlen können in Java CDC ganze Zahlen mit unendlichem Wertebereich dargestellt werden.
positiveInteger	java.Math.BigInteger	siehe `integer`
negativeInteger	java.Math.BigInteger	siehe `integer`
nonPositiveInteger	java.Math.BigInteger	siehe `integer`
nonNegativeInteger	java.Math.BigInteger	siehe `integer`
positiveInteger	java.Math.BigInteger	siehe `integer`
unsignedByte	short	In Java gibt es keine simplen Datentypen für die Darstellung vorzeichenloser Ganzzahlen. Dieses Problem kann umgangen werden, indem in Java eine vorzeichenlose ganze Zahl durch eine vorzeichenbehaftete ganze Zahl mit der nächstgrößeren Weite repräsentiert wird. So wird ein `unsignedByte` auf den `short`-Datentyp abgebildet.
unsignedShort	int	siehe `unsignedByte`
unsignedInt	long	siehe `unsignedByte`
unsignedLong	java.Math.BigInteger	64 bit sind in Java die maximale Weite eines simplen Datentyps. Daher kann ein `unsignedLong` nicht durch einen simplen Typen dargestellt werden, sondern nur durch `BigInteger`.
dateTime	java.util.Calendar	
date, gYear, gYearMonth, gMonth, gMonthDay, gDay, duration	java.lang.String	Eine automatische Verarbeitung von Zeit- und Datums-bezogenen Daten wie bei `dateTime` kann mangels einer Entsprechung in Java CDC nicht realisiert werden. Somit bleibt die übertragene Zeichenkette unangetastet.
Name	java.lang.String	
QName	org.ws4d.java.types.QName	
NCName	java.lang.String	
anyURI	org.ws4d.java.types.URI	
language, ID, IDREF, ENTITY, NMTOKEN	java.lang.String	
IDREFS, ENTITIES, NMTOKENS	java.lang.String[]	Diese XML-Schema-Datentypen repräsentieren eine Liste aus den zuvor definierten einzeln auftretenden Typen.
base64Binary	org.ws4d.java.attachment.Attachment	Binäre Daten vom Typ `base64Binary` werden in DPWS in einer modifizierten SOAP-Nachricht innerhalb des MIME-Teil übertragen.
hexBinary	byte[]	

Elemente können in XML-Schema mit Attributen versehen werden, welche die Anwendung der eben definierten Abbildungsregeln teilweise verhindern.

- Wenn ein simpler Datentyp in einem optionalem XML-Attribut verwendet wird, ist die Abbildung auf einen primitiven Java-Datentyp nicht möglich, da ein solcher Typ immer mit einem Wert belegt werden muss.
- Wenn ein simpler Datentyp für ein Element verwendet wird, das zusätzlich die Attribute `minOccurs=0` oder `nillable="true"` gesetzt hat, so ist auch hier die Verwendung eines primitiven Datentyps nicht möglich.
- Wenn ein simpler Datentyp für ein Element verwendet wird, in dem zusätzlich das Attribute `maxOccurs` mit einem Wert > 0 angegeben ist, so werden die übermittelten Instanzen in einer `java.util.List` abgelegt. Auch das ist mit primitiven Datentypen nicht möglich.

Tritt einer dieser Fälle auf, so wird statt des primitiven Datentyps deren Wrapper-Klasse verwendet. Diese Objekte können den Wert `null` annehmen und auch in Listen verwaltet werden.

Komplexe Datentypen in XML-Schema sind grundsätzlich lediglich Arrangements von simplen Typen, so dass die Aufstellung von Regeln zur Abbildung komplexer Typen nicht notwendig ist.

Parametrisierung der Abbildungen Wenn bestehende OSGi-Clients mit bestehenden DPWS-Services kommunizieren sollen, ergibt sich oftmals das Problem, dass eine semantische Übereinstimmung besteht, jedoch keine syntaktische. Gegeben sei z. B. eine Patienten-Datenbank mit einer DPWS-Schnittstelle und ein OSGi-Client, der auch für den Zugriff auf eine Patienten-Datenbank geschrieben wurde. Trotz dieser grundsätzlichen semantischen Entsprechung ist es unwahrscheinlich, dass die beiden Schnittstellen zueinander passen. Für dieses Problem existieren in der Forschung bereits diverse Ansätze. In [Sam06] werden Services mit Hilfe semantischer Beschreibungen auf Kompatibilität analysiert, in [Mot07] werden syntaktische Analysen der Interfaces mit den Methodensignaturen vorgenommen. Diese Ansätze bieten aber keine automatische Lösung an und beziehen sich auf Services der gleichen Technologie. Sie sind in der Anwendung somit sehr beschränkt.

Das Comoros.MappingTool bietet einen einfachen aber praktikablen manuellen Ansatz an. Die Inkompatibilitäten eines Service-Schnittstellen können einerseits eine größere oder kleinere Menge an Informationen anbieten als der Client erwartet, andererseits können Informationen auch unterschiedlich formatiert sein. Beide Fälle sorgen für eine syntaktische Inkompatibilität. Der erste Fall kann durch die freie Definition der Abbildungen im Comoros.MappingTool einfach überwunden werden. Simple Datentypen auf der DPWS-Seite werden mit den entsprechenden Typen auf der Java-Seite verknüpft. Informationen aus nicht benötigten Elementen werden einfach ausgelassen. Sind die Informationen aber unterschiedlich formatiert, oder kann eine Information für den Client nur aus der Aggregation mehrerer Elemente der Server-Seite berechnet werden, müssen Abbildungen parametrisiert werden.Abbildung 6.15 zeigt diese Anwendungsfälle.

Im Beispiel der Patienten-Datenbank kann es z. B. vorkommen, dass die Dosierung eines Medikaments vom Server in einer anderen Einheit angeboten wird als vom Client

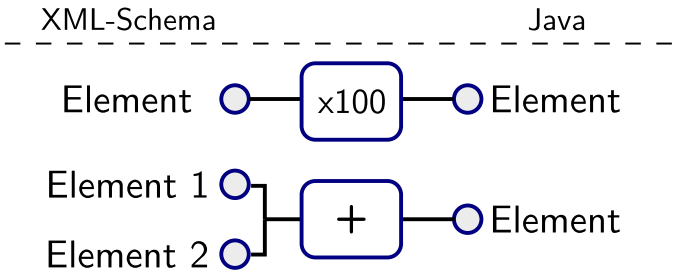


Abbildung 6.15: Parametrisierung von $1 - to - 1$ und $N - to - N$ Abbildungen

erwartet. Diese Abbildung muss dann mit einer entsprechenden arithmetischen Operation parametrisiert werden. Insgesamt ist die Parametrisierung aber nicht nur auf arithmetische Operationen beschränkt, sondern kann vom Entwickler beliebig vorgenommen werden.

Werkzeug-Unterstützung

Um den von Comoros geforderten hohen Nutzungskomfort zu erreichen (siehe Abschnitt 4.5), ist die Werkzeug-Unterstützung des Entwicklers für komplexe Aufgaben unerlässlich. Das Comoros.MappingTool bietet die Möglichkeiten Abbildungen zwischen XML- und Java-Datentypen komfortabel zu definieren. Ähnliche Werkzeuge, wie der *XSLT-Mapper* aus dem *JDeveloper*-Toolkit oder der *BizTalk Schema Mapper* [Rob05], realisieren lediglich frei definierbare Abbildungen zwischen XML-Schema-Datentypen. Für die Abbildung zwischen XML und Java ist kein Tool bekannt.

Für die Demonstration der Funktionalität des Werkzeugs wird das Beispiel der Patienten-Datenbank wieder aufgegriffen. Die Datenbank eines Krankenhauses verwaltet Patienten mit einer Schilddrüsen-Fehlfunktion. Sie speichert neben den persönlichen Daten der Patienten – Name, Vorname, Alter und Adresse – den letzten gemessenen Wert des Thyreoida-stimulierenden Hormons (TSH) in der Einheit `mU/Liter` und für die Medikation die entsprechende Dosis an Levothyroxin-Natrium für den Tag und für die Nacht. Das Krankenhaus bietet eine DPWS-Schnittstelle mit den in Abbildung 6.3 definierten Daten.

Ein kooperierendes Ärztehaus hat Zugriff auf diese Schnittstelle um einen schnellen Einblick in die Daten der Patienten, die zu der weiterführenden Behandlung überwiesen wurden, zu erlangen. Das Informationssystem des Ärztehauses basiert auf OSGi. Der Client, der die Patientendaten abfragt, erwartet teilweise unterschiedliche Daten. Name und Vorname sind in einem Feld untergebracht. Neben dem Alter ist für die Behandlung nicht der komplette Wohnort, sondern nur das Land von Interesse, da hierdurch Rückschlüsse über die Jod-Versorgung getroffen werden können. Der TSH-Wert wird in `mU/ml` erwartet, die Dosis an an Levothyroxin-Natrium wird nur einmal pro Tag verabreicht. Das entsprechende Java-Interface ist wie folgt definiert:

```
1 public interface TSHPatient {
2   public PersonalData getPersonalData(int patientId);
3   public MedicalData getMedicalData(int patientId);
4 }
5
```

```xml
<Patient>
  <name>Mueller</name>
  <surname>Hans</surname>
  <age>33</age>
  <Address>
    <Street>Ulmenallee 4</Street>
    <Town>Osnabrueck</Town>
    <Country>Germany</Country>
  </Address>
  <TSHValue>0,07</TSHValue>
  <NightDose>50</NightDose>
  <DayDose>50</DayDose>
<Patient>
```

(a) XML-Instanz

```xml
<xs:element name="Patient">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Name" type="xs:string"/>
   <xs:element name="Surname" type="xs:string"/>
   <xs:element name="Age" type="xs:int"/>
   <xs:element name="Address">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Street" type="xs:string"/>
       <xs:element name="Town" type="xs:string"/>
       <xs:element name="Country" type="xs:string"/>
      </xs:sequence>
     </xs:complexType>
   </xs:element>
   <xs:element name="TSHValue" type="xs:float"/>
   <xs:element name="NightDose" type="xs:int"/>
   <xs:element name="DayDose" type="xs:int"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

(b) XML-Schema

Listing 6.3: Eintrag einer Patientendatenbank. XML-Instanz mit dem dazugehörigen XML-Schema.

```
public class PersonalData {
  public String name;
  public int age;
  public String country;
}

public class MedicalData {
  public float tshValue;
  public int dailyDosis;
}
```

An diesem Beispiel ist gut zu erkennen, dass zwar eine semantische Übereinstimmung vorliegt, die Interfaces und Daten aber syntaktisch nicht zueinander passen. Mit Unterstützung des Comoros.MappingTool können die Komponenten aber dennoch kommunizieren.

Einlesen der DPWS-Komponente Das Einlesen der Informationen über DPWS-Devices ist in Abbildung 6.17 dargestellt. Es gibt die Möglichkeit eine vorhandene WSDL zu laden oder im Netz nach DPWS-Geräten zu suchen. In beiden Fällen wird im *DPWS-Explorer*-Tab die Struktur der Geräte und deren Services abgebildet.

Einlesen der OSGi-Komponente Ähnlich wie bei der DPWS-Komponente können auch für OSGi-Komponenten Informationen sowohl zu laufenden als auch zu nicht gestarteten Services eingelesen werden. Im ersten Fall muss zuerst über eine *JMX*-Schnittstelle eine Verbindung zu einem OSGi-Framework aufgebaut und anschließend der gesuchte Service ausgesucht werden. Weiterhin können auch nicht gestartete API-Bundles oder

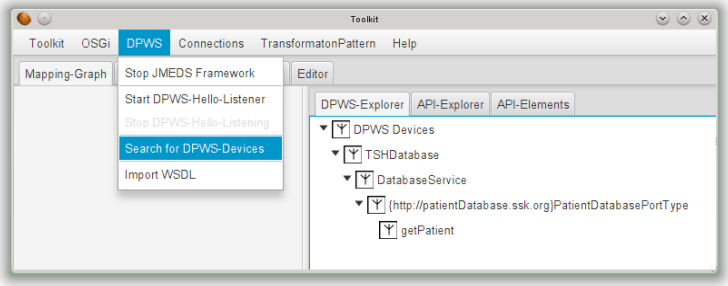


Abbildung 6.17: Einlesen der DPWS-Servicebeschreibungen über eine WSDL

Java-Interfaces in das Tool geladen werden. Anschließend erscheint, wie in Abbildung 6.18 abgebildet, die Service-API im *API-Explorer*. Dort kann bereits die Struktur analysiert werden.

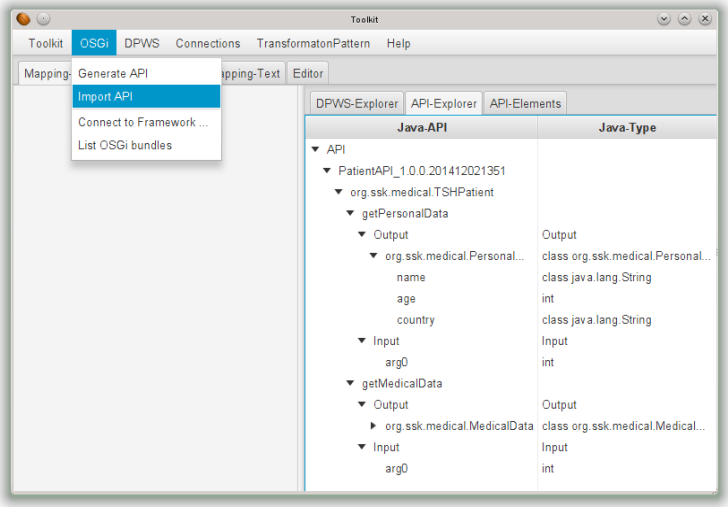


Abbildung 6.18: Einlesen der OSGi-Servicebeschreibungen über ein Java-Interface

Erstellen einer Abbildung Aus den Explorer-Tabs können die unterschiedlichen Komponenten mittels einer Drag&Drop-Geste in das freie Bearbeitungsfeld gezogen werden. Hier können, wie in Abbildung 6.19 illustriert, die Abbildungen definiert werden.

Im Beispiel erfolgt die Abfrage der Patienten-Daten auf der DPWS-Seite mittels einer Operation, während auf der OSGi-Seite zwei Methoden definiert sind. Eine Methode zur Abfrage der Personendaten, und eine Methode zur Abfrage der medizinischen Daten. Für die Abbildung der Datentypen ist das nicht von Relevanz, da hier lediglich die Zuordnungen der Nutzdaten definiert werden. Wie bereits erläutert, besteht in dem Beispiel lediglich beim Alter und beim Land eine direkte Übereinstimmung. Hier kann einfach per Drag&Drop-Geste eine Verbindung zwischen den jeweiligen Elementen gezogen

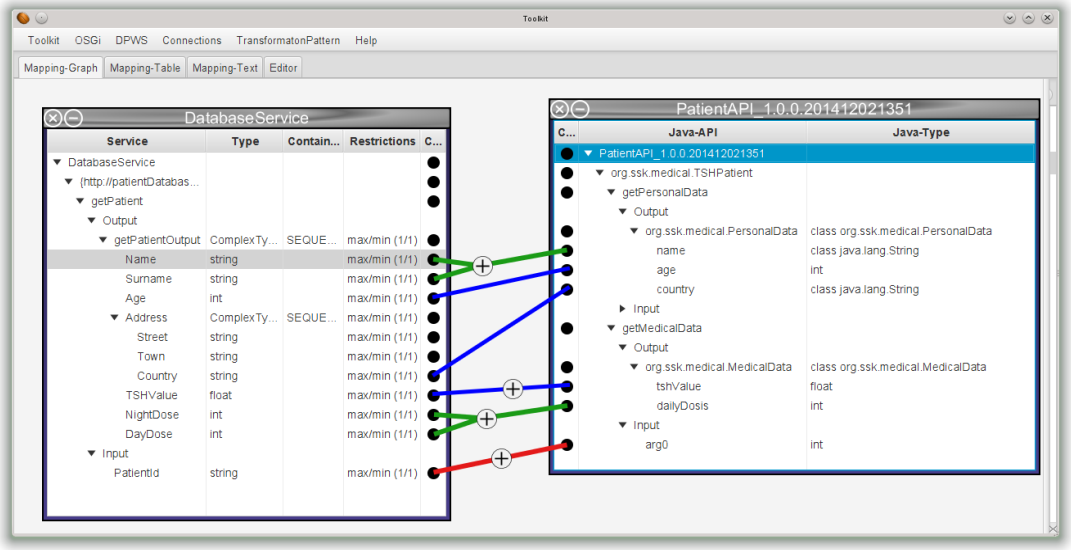


Abbildung 6.19: Erstellen einer Datenformat-Abbildung

werden. Die nächte Verbindung betrifft die Information über den aktuellen TSH-Wert des Patienten. Hier liefert das Krankenhaus den Wert in der Einheit `mU/Liter` während das Ärztehaus die Einheit `mU/ml` verwendet. Die erstellte Verbindung muss also parametrisiert werden. Das Einfügen des Parametrisierungs-Symbols erlaubt das Öffnen eines Editors, in dem definiert wird, dass der übermittelte Wert des Krankenhauses mit `100` multipliziert wird. Das Parametrisierungs-Objekt erlaubt auch die Aggregation von Verbindungen. Dies ist hier notwendig, da das Ärztehaus im Gegensatz zum Krankenhaus die Verabreichung der Medikamente nur einmal am Tag vornimmt. Bei der Medikamenten-Dosis werden also die Tag- und Nacht-Werte in das Parametrisierungs-Objekt geführt und anschließend auf die tägliche Dosis im Ärztehaus abgebildet. Innerhalb der Parametrisierung wird eine einfache Addition definiert. Beim Namen hingegen werden Vor- und Nachname des Krankenhauses konkateniert und auf das Feld im OSGi-Interface abgebildet. Zuletzt wird noch der Eingabe-Parameter verbunden. Auf der DPWS-Seite wird ein `string` erwartet, während im OSGi-Service als Patienten-Id jeweils ein `int` übermittelt wird. Aufgrund der definierten Regeln zur Validierung von Abbildungen (siehe Tabelle 6.2) wird diese Verbindung als ungültig (rot) markiert. Um die Bindung zwischen Service und Client dennoch zu ermöglichen können aber auch ungültige Verbindungen parametrisiert werden. So kann an dieser Stelle aus einer Patienten-Id des Ärztehauses eine gültige Patienten-Id des Krankenhauses erzeugt werden.

Generierung des Transformationspattern Aus den definierten Abbildungen können mit dem Comoros.MappingTool auch die dazu passenden Transformationspattern erzeugt werden. Dazu wird wie in Abbildung 6.20 gezeigt, Quelltext generiert. Der Entwickler kann den Quelltext weiter bearbeiten um so weitere Funktionen zu implementieren, die

über die des Werkzeugs hinausgehen. Anschließend kann der Quelltext kompiliert und über eine *JMX*-Schnittstelle in ein OSGi-Framework installiert werden. Beim Deployment des Transformationspattern wird automatisch die Komponente zur Abbildung einer OSGi-Suchanfrage auf den hier behandelten DPWS-Service installiert.

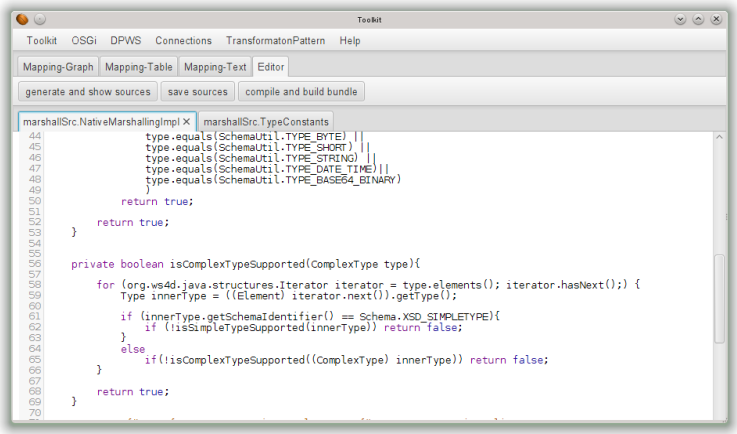


Abbildung 6.20: Generierung des Transformationspattern

Erstellung einer OSGi-API Soll für einen DPWS-Service ein neuer OSGi-Client entwickelt werden, so liegt keine Service-API vor. Diese muss zuvor erzeugt werden. Das Comoros.MappingTool erlaubt die Erstellung mit graphischen Hilfsmitteln wie in Abbildung 6.21 dargestellt. Innerhalb des Service-Interface können über Drag&Drop-Mechanismen beliebige Methoden und Klassen erstellt werden. Anschließend kann das erzeugte Interface für die Definition der Abbildungen verwendet werden.

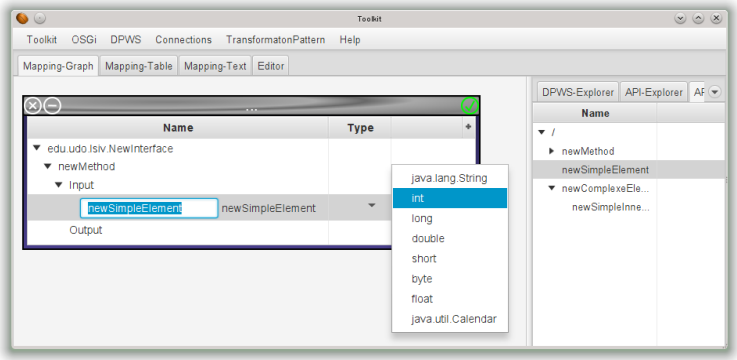


Abbildung 6.21: Erstellung einer OSGi-Service-API

6.4.2 Integration serieller Geräte

SOA-fähige Geräte, wie z. B. DPWS- oder UPnP-Geräte, bieten ihre Funktionalität über Service-Schnittstellen an und ermöglichen somit eine Anwendungsentwicklung ohne

Kenntnis über die Gerätetreiber. Diese Entkopplung ist ein wichtiger Baustein zur komfortablen und beschleunigten Entwicklung von neuen Systemen.

Alte Technologien, wie z. B. RS 232, bieten keine Service-Schnittstellen an. Um diese Geräte dennoch innerhalb einer OSGi-Umgebung nutzen zu können, hat die OSGi-Allianz die *Device Access Specification* [The12a] (siehe Unterabschnitt 2.1.4) veröffentlicht. Diese Spezifikation regelt aber nur, wie ein passender Treiber für den Gerätezugriff im System installiert wird, nicht jedoch, wie die Anwendung auf den Gerätetreiber zugreift. Um die Abhängigkeit von Gerätetreiber und Anwendung aufzulösen müssen demnach Schnittstellen für Gerätetreiber definiert werden. Gerätetreiber und Anwendungen können dann gleichzeitig entwickelt und beliebig ausgetauscht werden.

Anforderungen

Als Ziel gilt die Integration von Geräten in verteilte OSGi-Umfelder, die von sich aus keine Service-Schnittstelle anbieten. In einer verteilten Dienste- und Geräteumgebung existieren oftmals eine große Anzahl solcher Geräte mit unterschiedlichen Kommunikationstechnologien, wie z. B. Bluetooth, ZigBee, WLAN oder RS 232. Die Geräte unterschiedlicher Hersteller realisieren zudem den Zugriff auf die Funktionalität auf verschiedene Arten. Einige Geräte versenden Rohdaten während andere Geräte bereits Programmierschnittstellen anbieten, die die Funktionen kapseln. Für eine einfache Anwendungsentwicklung und eine sinnvolle Verteilung der Funktionen eines Gerätes durch die Comoros-Middleware müssen solche Schnittstellen, Bindeglied zwischen Anwendung und Gerätetreiber, einheitlich für alle Geräte eingeführt werden. Dafür gelten die folgenden Anforderungen:

Abstraktion der Gerätefunktionalität auf Service-Schnittstellen Die OSGi-Device-Access-Spezifikation wird um eine Schicht erweitert, in der Funktionen eines Geräts als OSGi-Service registriert werden. Dieser Service wird von Anwendungen aufgerufen. Die Aufrufe werden dann an den Gerätetreiber, der über den Device-Access-Mechanismus im OSGi-System installiert wurde, weitergeleitet. Auf diese Weise wird eine Entkopplung zwischen Anwendung und Gerätetreiber erreicht.

Discovery-Mechanismen für serielle Geräte Viele ältere Übertragungstechnologien bieten kein automatisches Discovery von Geräten an, wie von DPWS über den WS-Discovery-Mechanismus realisiert. Um diese Technologien dennoch vollständig zu unterstützen wird eine Discovery-Schnittstelle für die jeweiligen Technologien eingeführt. Über diese Schnittstelle können Geräte manuell hinzugefügt oder Suchmechanismen angestoßen werden.

Allgemeingültige Definitionen von Schnittstellen für unterschiedliche Dienste Um die Entwicklung von Anwendungen innerhalb einer Geräte-basierten Umgebung weiter zu vereinfachen, müssen die angesprochenen Schnittstellen eine allgemeine Gültigkeit besitzen. Ähnlich der Geräteprofile in UPnP und DPWS wird so für jede Art einer Funktion genau ein Interface definiert. Die so entwickelten, öffentlich zugänglichen Schnittstellen ermöglichen den Austausch von Geräten ohne Anpassung des Anwendungs-Codes.

Architektur

Die Geräteintegration wurde in Zusammenarbeit mit der Universität Rostock und dem Forschungsinstitut OFFIS innerhalb des OSAmI-Projekts entwickelt [Dai11; Gor10]. Dabei lag der Fokus der Universität Rostock auf der Abstraktion der Funktionen eines Geräts von den Übertragungstechnologien und der Entwicklung geeigneter Discovery-Mechanismen. Der Fokus von OFFIS lag auf der Definition allgemeingültiger Schnittstellen während die TU Dortmund die Integration in eine verteilte OSGi-Umgebung sicherstellte. An dieser Stelle wird nur ein grober Überblick über die Architektur gegeben, detaillierte Ausführungen können in den angegeben Publikationen eingesehen werden.

Die hier entwickelte Geräteintegration erweitert die OSGi-Device-Access-Spezfikation (siehe Unterabschnitt 2.1.4). In dieser werden Geräte durch einen *Base Driver* erkannt, der daraufhin einen *OSGi Device Service* erzeugt und registriert. Der *Device Manager* erkennt diesen Service, sucht nach einem passenden Treiber für das Gerät und installiert diesen. Das Gerät kann nun in OSGi verwendet werden.

Innerhalb der Geräteintegration werden nun zwei weitere Treiber eingeführt. Der *Composite Function Driver* wird, wie in der Device-Access-Spezifikation, über den *Device Manager* an die OSGi-Abstraktion des Gerätes gebunden. Innerhalb dieses Treibers werden die Funktionen des Gerätes identifiziert und eine Menge von *Function Interfaces* installiert, die als Service-Schnittstelle für die Anwendung dienen. Der *Discovery Driver* ist eine Erweiterung des *Base Driver* und bezieht sich demnach auf eine bestimmte Klasse von Technologien. So existieren *Discovery Driver* z. B. für Bluetooth oder KNX. Innerhalb des *Base Driver* ist das Auffinden von Geräten nicht geregelt und aus einer Anwendung heraus kann dieser Prozess auch nicht angestoßen werden. Über die Schnittstellen des *Discovery Driver* kann nun aus der Anwendung heraus innerhalb der unterstützten Technologie nach neuen Geräten gesucht und diese als *OSGi Device Service* installiert werden. Danach greifen die Mechanismen der *Device Access Specification*. Alle beteiligten Komponenten sind in Abbildung 6.22 abgebildet.

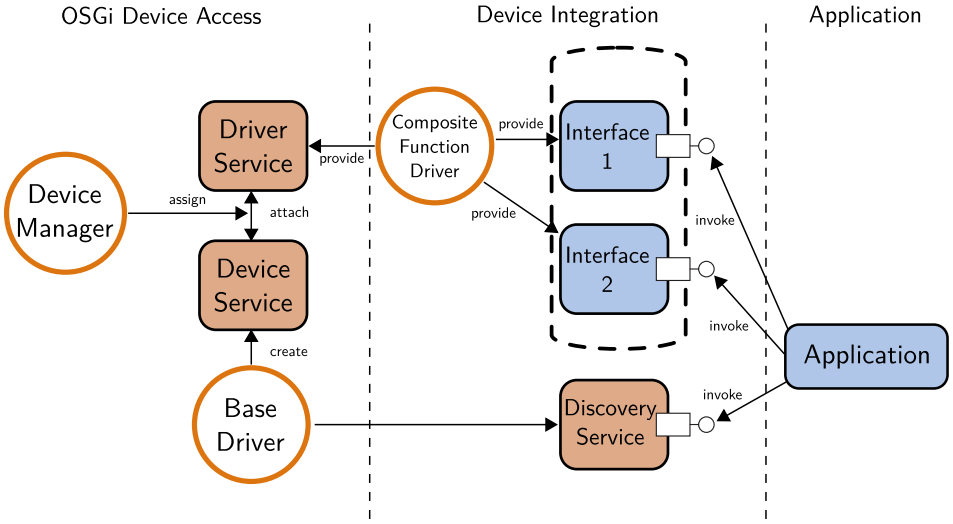


Abbildung 6.22: Komponenten der Geräteintegration

Die bereitgestellten *Function Interfaces* sind öffentlich verfügbar und beschreiben allgemeine Gerätetypen. Innerhalb des OSAmI-Projekts wurden vom Forschungsinstitut OFFIS solche Schnittstellen für medizinische Geräte, unter anderem für EKG-, Blutdruck-, Sauerstoffsättigugs- und Herzfrequenz-Dienste erstellt [Mül12]. Mittlerweile existieren auch Schnittstellen für Geräte aus dem Bereich der Gebäudeautomatisierung.

Damit die Services, welche diese Schnittstellen implementieren, durch die Comoros-Middleware auf entfernte Plattformen verteilt werden können, müssen die Interfaces Datentyp-Konventionen einhalten, die von Comoros vorgegeben werden. Diese Konventionen wurden für die Geräteintegration explizit definiert [OSA11]. Zusammenfassend können durch den Mechanismus der Geräteintegration auch serielle Geräte in OSGi verwendet werden und an einer plattformübergreifenden Kommunikation teilnehmen.

Die Ergebnisse der Forschungsarbeiten wurden in Form eines RFP bei der OSGi-Allianz eingereicht.

6.5 Kommunikationsprotokolle

ws4d.Comoros verwendet als Basiskommunikationsprotokoll das *Devices Profile for Webservices*, mit dem innerhalb der Kernarchitektur die *OSGi Remote Services* realisiert werden. Um die Flexibilität zu erhöhen und ein größeres Anwendungsfeld zu generieren, wurde durch eine modulare Architektur die Möglichkeit erschaffen weitere Kommunikationsprotokolle in Comoros zu integrieren. Abbildung 5.1 präsentiert die Gesamtarchitektur mit den möglichen Protokollerweiterungen. Neben der Integration zusätzlicher Protokolle wird auch die Effizienzsteigerung des DPWS-Protokolls untersucht, um den Einsatz von Comoros in Ressourcen beschränkten Umgebungen weiter zu optimieren.

6.5.1 Integration weiterer Kommunikationsprotokolle

Die Integration weiterer Kommunikationsprotokolle erfolgt über das JMEDS-Framework. Die JMEDS-Architektur ist sowohl für die Client-Seite, als auch für die Server-Seite in drei Schichten unterteilt, die Anwendungs-Schicht, die Dispatching-Schicht und die Kommunikationsschicht. Die Bestandteile der einzelnen Schichten sind in Abbildung 6.23 dargestellt.

Die beiden oberen Schichten zeigen die Nutzerschnittstelle zum Framework. Auf der Anwendungsschicht hat der Benutzer Zugriff auf die Geräte, die Services und deren Operationen. Der Client greift auf die DPWS-Geräte und Services über Proxy-Objekte zu, die sich auf die jeweilige entfernte Komponente beziehen. Auf der Server-Seite werden Bindings definiert, in denen Geräte und Services konkrete Protokolle für den Zugriff zugewiesen werden. Die Dispatching-Schicht fungiert als Bindeglied zwischen Anwendungs-Schicht und Kommunikations-Schicht und sorgt für den korrekten Nachrichtenaustausch. Für das Discovery der DPWS-Geräte wurde auf der Client-Seite der Search-Manager implementiert. Durch vorhergehende Suchanfragen bereits gefundene Komponenten werden in der Device/Service-Registry verwaltet, um unnötigen Netzwerk-Traffic zu vermeiden.

Die Kommunikationsschicht enthält einen oder mehrere *Communication Manager*. Der Communication-Manager empfängt Nachrichten der oberen Schicht in Form von

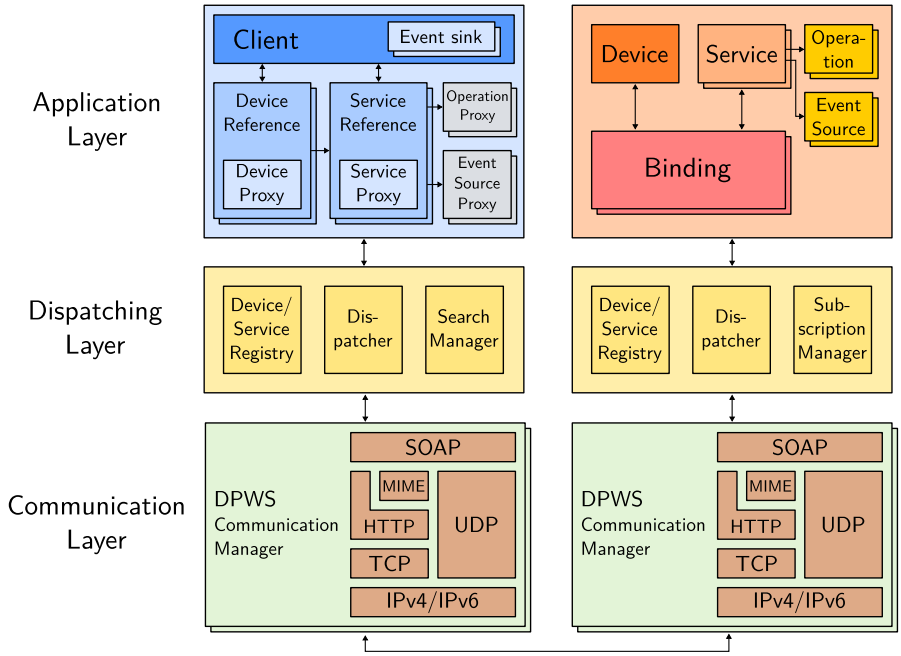


Abbildung 6.23: Architektur des JMEDS-Frameworks

Objekten und serialisiert diese zu SOAP-Nachrichten und sendet sie über die in diesem Manager umgesetzte Technologie über das Netzwerk an entfernte Endpunkte. Um weitere Kommunikationsprotokolle für Comoros anzubieten, müssen also lediglich weitere Communication-Manager für das JMEDS-Framework implementiert werden. Durch die Entkopplung der Anwendungsentwicklung vom verwendeten Transportprotokoll bleibt die Implementierung der Comoros-Schicht unberührt.

Für die JMEDS-Bibliothek existieren bis zu diesem Zeitpunkt Implementierungen für das DPWS-Protokoll, für UPnP, für Bluetooth und für ZigBee. Eine Implementierung für das CoAP-Protokoll ist in Entwicklung. Damit ein OSGi-Service mittels Comoros über eines der weiteren Protokolle entfernt zugreifbar ist, muss bei der Service-Registrierung ein entsprechender ConfigurationType angegeben werden. Für UPnP ergibt sich folgendes Beispiel:

```
service.exported.configs = org.ws4d.upnp
org.ws4d.upnp.address    = 129.217.16.20
org.ws4d.upnp.port       = 8090
org.ws4d.upnp.path       = service-id/24
```

6.5.2 Effizienzsteigerung des DPWS-Protokolls

Für die Kommunikation zwischen DPWS-Geräten werden SOAP-Nachrichten verwendet, die auf der *Extensible Markup Language (XML)* basieren. XML-Nachrichten sind Textnachrichten, die vom Menschen gelesen werden können, aber daher auch entsprechend schwergewichtig sind. Dadurch wird im Vergleich zu binären Datenformaten ein erhöhter Speicherbedarf generiert. Um nun den Anspruch von Comoros gerecht zu werden, in

Ressourcen beschränkten Umgebungen eingesetzt werden zu können, kann die Effizienz des DPWS-Protokolls auf verschiedene Arten gesteigert werden:

- Durch die Kompression der Nachrichten.
- Durch die Einführung effizienter Transportmechanismen für SOAP-Nachrichten.

Komprimierung von SOAP-Nachrichten

Die Kompressionsverfahren in der Kommunikation wurden in Unterabschnitt 2.1.7 bereits eingeführt. Alle dort vorgestellten Verfahren können grundsätzlich für den beschriebenen Anwendungsfall verwendet werden. Werner [Wer06a; Wer06b] liefert dazu in seiner Arbeit eine Übersicht über die Effizienz der einzelnen Verfahren. Mit dem *Efficient XML Interchange (EXI)* Format wurde ein binäres XML-Format entwickelt, das im besonderen Maße zur Kompression der DPWS-Nachrichten geeignet ist, wie die vom W3C vorgenommene Evaluierung [Bou08] des Standards belegt. Moritz [Mor12] beschreibt in seiner Arbeit zusätzlich die Anwendbarkeit von EXI auf das DPWS-Protokoll in Ressourcen beschränkten Netzwerken wie dem 6LoWPAN, und vergleicht unterschiedliche Kompressionsverfahren in dieser Umgebung.

Für das vom W3C spezifizierte EXI-Format existierte zum Zeitpunkt der Comoros-Entwicklung keine verwendbare Implementierung. Eine solche Implementierung wurde im Rahmen einer studentischen Abschlussarbeit [Beh11] entwickelt, die sich wie in Abbildung 6.24 dargestellt, in die Comoros-Architektur integriert.

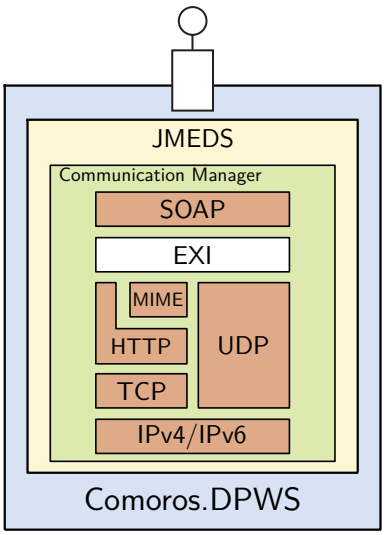


Abbildung 6.24: JMEDS-Erweiterung um EXI-Kodierung (vgl. Abbildung 6.23)

Das Comoros-DPWS-Bundle kapselt die DPWS-Implementierung JMEDS. Innerhalb von JMEDS sind die verschiedenen Kommunikationsprotokolle in den *Communication Manager* modular implementiert. Die EXI-Kodierung wird nun in den DPWS-Communication-Manager mittels einer optionalen Erweiterung integriert. SOAP-Nachrichten werden nun EXI-kodiert und nicht mehr als UTF8-kodiertes XML versendet.

Diese Funktionalität muss über das Setzen von entsprechenden Properties vom Benutzer aktiviert werden.

Durch die Verwendung von EXI wird zwar die Effizienz gesteigert, aber auch die Interoperabilität verringert. EXI-kodierte Nachrichten können nur von DPWS-Geräten interpretiert werden, denen dieses Nachrichtenformat bekannt ist. Für die Kommunikation zwischen OSGi-Plattformen ist die Interpretierbarkeit folglich gesichert. Generelle DPWS-Geräte sprechen diesen Dialekt zumeist nicht. Da es sich jedoch um einen offenen W3C-Standard handelt, kann die Unterstützung jeder DPWS-Implementierung hinzugefügt werden.

Einführung effizienter Transport-Bindings

Der Transport von SOAP-Dokumenten zwischen zwei Endpunkten ist innerhalb des SOAP-Messaging-Framework [Gud07] beschrieben. Ein konkretes Transportprotokoll wird dabei nicht angegeben, vielmehr ist eine Entkopplung von diesen realisiert. Das konkret verwendete Protokoll wird erst vom Dienst-Entwickler innerhalb eines *Binding* definiert.

In Absatz 2.1.7 wurde eine effiziente Transportbindung für das DPWS präsentiert, das *SOAP-over-CoAP*-Binding. Die Implementierung von Moritz ist als gSOAP-Plugin realisiert und in die DPWS-Implementierung WS4D-gSOAP [1] integriert. Daher steht lediglich eine C-Implementierung zur Verfügung. Für die Integration in Comoros wurde eine neuartige Java-Implementierung geschaffen. Diese entstand in Teilen in der Masterarbeit von Avramov [Avr13]. An die Java-Implementierung wurden folgende Anforderungen gestellt:

- **SOAP-over-CoAP-Unterstützung für die Bereiche Discovery, Eventing, Description und Messaging:** Um die gesamte DPWS-Funktionalität abzudecken müssen die SOAP-Nachrichten aus allen genannten Bereichen über das CoAP-Protokoll versendet werden.
- **EXI-Kodierung für SOAP-Nachrichten:** Um die Effizienz des Transportmechanismus weiter zu verbessern, muss die Nutzlast der CoAP-Nachricht, also das SOAP-Dokument, in effizienter, binärer Form kodiert werden können. Hierbei wird auf das EXI-Kodierverfahren zurückgegriffen. Durch die Verkleinerung der Nachricht werden Fragmentierungen der CoAP-Nachrichten vermindert. Um die Kodierung festzulegen wird innerhalb des CoAP-Headers der Medien-Typ (`application/xml` oder `application/exi`) im `options`-Feld definiert.
- **Zuverlässigkeit:** Der zuverlässige Transport von Nachrichten muss gesichert werden. Dazu werden alle beschriebenen Transportmodi umgesetzt.

Ein Überblick gebendes Schaubild zur veränderten Architektur durch das SOAP-over-CoAP-Binding ist in Abbildung 6.25 dargestellt. Das übergeordnete Szenario ist die Entwicklung verteilter OSGi-Anwendungen. Die verteilte Nutzung wird durch die OSGi-Remote-Services-Spezifikation definiert und von Comoros umgesetzt. Unterhalb dieser

1 Ed.: WS4D, URL: `http://ws4d.e-technik.uni-rostock.de/gsoap/`, Abruf: 09.06.2015

beiden Schichten befindet sich das Kommunikationsprotokoll, in diesem Fall das DPWS. Diese Schichten werden durch das JMEDS-Framework angeboten. Die Neuigkeiten finden sich auf den drei untersten Schichten. Im Gegensatz zu der OASIS-WS-DD-Spezifikation wurden die Protokolle TCP und HTTP durch CoAP und UDP ersetzt. Weiterhin wurde der SOAP-Schicht die Möglichkeit zur EXI-Kodierung hinzugefügt. Die unterste Schicht repräsentiert die unterstützten Protokolle der Netzwerk-Schicht. Durch den Einsatz von CoAP werden nun auch Ressourcen beschränkte 6LoWPAN-Netzwerke unterstützt. Somit kann durch die Integration von *SOAP-over-CoAP* in Comoros eine Verbindung zwischen Geräten in diesen Netzwerken und Ressourcen starken Netzwerken ohne Proxys realisiert werden.

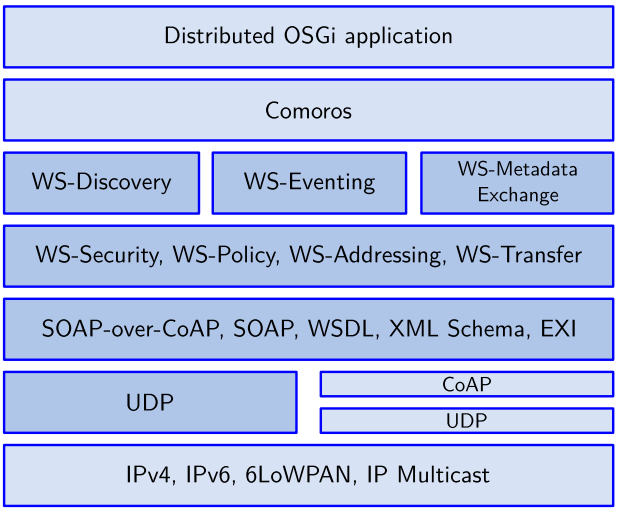


Abbildung 6.25: Comoros-Stack mit der SOAP-over-CoAP-Erweiterung

Um das SOAP-over-CoAP-Binding in Comoros zu integrieren, muss analog zu der Einführung neuer Kommunikationsprotokolle in JMEDS ein neuer *Communication Manager* entwickelt werden. Anstelle einer kompletten Neuentwicklung wurde der bestehende DPWS-Communication-Manager um die CoAP-Funktionalitäten erweitert. Die Architektur ist in Abbildung 6.26 illustriert.

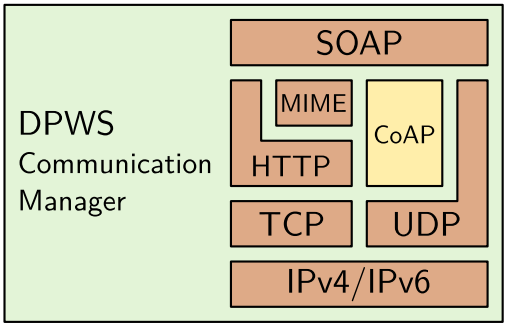


Abbildung 6.26: Architektur des erweiterten DPWS-Communication-Managers (vgl. Abbildung 6.23)

Während das UDP-Protokoll im ursprünglichen Communication-Manager ausschließlich für das SOAP-over-UDP-Binding verwendet wird, ist dieses nun die integrale Basis für das CoAP-Protokoll. Die SOAP-Schicht bleibt unberührt und wird von allen drei Bindings (SOAP-over-HTTP, SOAP-over-UDP und SOAP-over-CoAP) in gleicher Weise verwendet. DPWS-Services, die das CoAP-Binding verwenden, können nur mit Geräten kommunizieren, die dieses Binding ebenfalls unterstützten. Bei der Entwicklung wurde die Interoperabilität mit den aus der Arbeit von Moritz entstandenen Implementierungen sichergestellt.

Die erweiterte JMEDS-Funktionalität wird innerhalb der Comoros-Middleware über das *DPWS-Bundle* angeboten, das die JMEDS-Bibliothek kapselt. OSGi-Applikationen können das neue Transport-Binding nun über entsprechende ConfigurationTypes verwenden:

```
service.exported.configs = org.ws4d.dpws
org.ws4d.dpws.encoding   = EXI
org.ws4d.dpws.binding    = CoAP
org.ws4d.dpws.address    = 129.217.16.20
org.ws4d.dpws.port       = 8090
org.ws4d.dpws.path       = service-id/24
```

6.6 Management der Middleware

Die Comoros-Middleware ist aus den besonderen Anforderungen der medizinischen Anwendungsdomäne entstanden, die in den Bereichen Interoperabilität, Zuverlässigkeit, Datensicherheit und Adaptivität liegen. Die Interoperabilität heterogener Dienste- und Geräte-Systeme wurde von Comoros durch Standardkompatibilität und -konformität bereits adressiert, so dass bisher isolierte Systeme unterschiedlicher Hersteller und unterschiedlicher Technologien zusammenarbeiten können. Die betrachteten Systeme stehen zusätzlich zumeist in wechselseitiger Interaktion mit ihrer Umgebung und müssen sich den äußeren Einflüssen anpassen. Veränderte Bedingungen, sowie Fehler und Ausfälle zur Laufzeit verlangen nach einer flexiblen und adaptiven Systemkonfiguration um ein vorhersehbares und zuverlässiges Systemverhalten zu garantieren. Gerade in Ressourcen beschränkten Umgebungen sind diese Merkmale schwierig umzusetzen, da zusätzlich die zur Verfügung stehenden Ressourcen verwaltet werden müssen und es schnell zu veränderten Systemeigenschaften kommt.

Adaptive und gütegesicherte Dienstesysteme werden zumeist über ganzheitliche Managementsysteme realisiert. Dabei besteht das technische Management, über das in Unterabschnitt 2.1.9 ein abstrakter Überblick gegeben wurde, im wesentlichen aus den folgenden drei Kernfunktionen:

- Monitoring
- Konfiguration
- Rekonfiguration

Werden die drei Punkte in das durch das *OSI Management Framework* definierte FCAPS eingeordnet, ergibt sich eine klare Zuordnung der Konfiguration und Rekonfi-

guration in das Configuration Management. Für die Überwachung existieren Anwendungsmöglichkeiten im Fault Management, Performance Management und im Security Management.

Eine Ausprägung eines Management-Systems für das OSGi-Umfeld wurde innerhalb des OSAmI-Projekts entwickelt [Doh10]. Comoros ist in dieses System integriert, der schematische Aufbau ist in Abbildung 6.27 abgebildet.

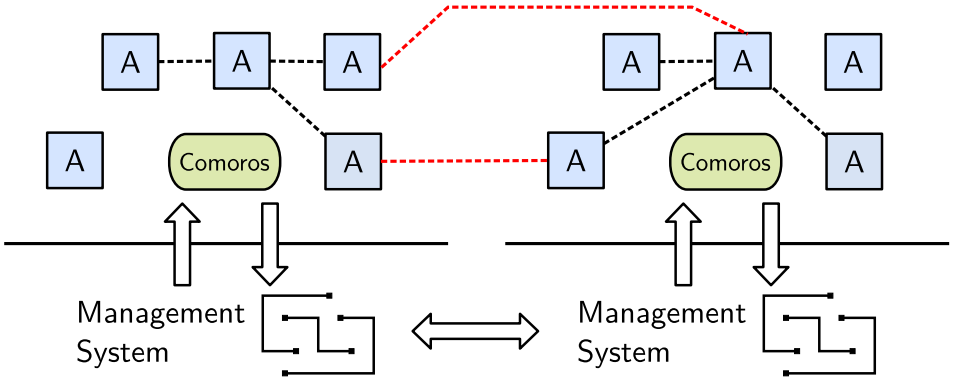


Abbildung 6.27: Comoros als Teil eines Management-Systems zur Umsetzung einer adaptiven Systemlandschaft

In der oberen Ebene sind die Systemelemente der verteilten Anwendung abgebildet. Aus Managementsicht werden diese in Komponenten und Assoziationen aufgeteilt. Komponenten definieren Geräte und Software, deren logische und physikalische Beziehungen untereinander als Assoziationen modelliert werden. Eine besondere Stellung im Comoros-Umfeld haben Software-Komponenten, die als OSGi-Bundle implementiert werden. Neben dem Anwendungsteil einer Softwarekomponente beinhaltet jede Komponente eine Schnittstelle zum Managementsystem um verwaltet, überwacht und gesteuert zu werden. Eine Relation zwischen zwei Komponenten wird durch eine Assoziation beschrieben, von denen der Typ der Service-Nutzung von besonderem Interesse ist, da diese von Comoros aufgebaut und verwaltet werden. Um in diesem dynamischen Prozess zu partizipieren wird Comoros selber als Komponente definiert und wird nach dem definierten Muster vom Management-Laufzeitsystem verwaltet.

Insgesamt wird innerhalb dieses Managementsystems zwischen der Planungsphase und dem Laufzeit-Managementsystem unterschieden. In der Planungsphase werden über den Ansatz des *Modellbasierten Managements (MBM)* [Ill06] Regeln für das System definiert, die anschließend durch das Laufzeitsystem umgesetzt werden. An dieser Stelle ist nur das Laufzeitsystem von Interesse.

6.6.1 Anforderungen

Grundsätzlich gelten die in Abschnitt 4.4 aufgestellten Anforderungen bezüglich der Konfiguration und der Überwachung von Comoros. Durch die Umsetzung dieser Anforderungen kann Comoros grundsätzlich an wechselnde Anforderungen und Anwendungsfälle im Sinne von Unterabschnitt 5.3.5 angepasst werden. Um in einem Managementsystem zu partizipieren, das die Erstellung eines adaptiven und gütegesicherten Dienste- und Ge-

rätesystems realisiert, muss die Comoros Middleware zusätzlich folgende Anforderungen erfüllen:

Defnition des Zustandsraums

Der für das Management relevante Zustandsraum wird in Form eines MIB [ISO89] ähnlichen Schemas als Management-Variablen festgelegt. Die Management-Variablen werden in *Statusvariablen* und *Konfigurationsvariablen* unterteilt. Statusvariablen legen den Management relevanten Zustand einer Komponente fest, werden aus der Anwendung gesetzt und können vom Management-Teil lediglich gelesen werden. Sie dienen explizit dem Monitoring-Prozess. Die Konfigurationsvariablen hingegen werden vom Management-Teil gesetzt und vom Anwendungsteil gelesen.

Öffentliche Schnittstelle zur Management-Funktionalität

Die Konfiguration und Überwachung des Systems soll mittels entfernter Management-Clients möglich sein. Aus diesem Grund wird eine öffentlich zugreifbare Schnittstelle benötigt. Die Schnittstelle soll sowohl lokal und entfernt von OSGi-Clients verwendet werden können, als auch von entfernten Webservice-Clients.

Konsistenzsicherstellung der Konfigurationen

Konfigurationen, die durch Management-Clients an Comoros übermittelt werden, können Fehler enthalten und ein ungewolltes Systemverhalten verursachen. Um dieses Problem zu verhindern müssen übermittelte Konfigurationen auf Konsistenz und Korrektheit überprüft werden. Fehlerhafte Konfigurationen werden nicht akzeptiert.

Standardkonformität

Die Umsetzung der Management-Funktionen muss auf offenen und akzeptierten Standards basieren. Nur so kann gesichert werden, dass Comoros einfach in beliebige Management-Systeme integriert und durch eine Vielzahl von Management-Clients direkt angesprochen werden kann.

6.6.2 Architektur

Um Comoros in eine Infrastruktur für adaptive Systeme einbetten zu können, wurde die in Abbildung 6.28 abgebildete grundlegende Architektur entworfen.

Comoros wird in eine Management-fähige Komponente, implementiert als OSGi-Bundle, eingebettet, in der der Zustandsraum abgebildet wird und dieser über eine Management-Schnittstelle abgerufen und manipuliert werden kann. Als Management-Clients werden sowohl OSGi-basierte Clients, als auch Webservice-basierte Clients unterstützt. Das gesamte Management wird in Konfiguration und Überwachung unterteilt.

Konfiguration

Für die lokale Konfiguration von OSGi-Bundles sieht die OSGi-Allianz die Verwendung des *Configuration Admin Service* [The12a] vor, der die Komponenten einzeln konfiguriert. Die Comoros-Middleware besteht allerdings aus mehreren Komponenten (DSW, Event-Converter, Marshaling, Logging), die alle konfiguriert werden müssen. Dazu ergeben sich zwei Varianten:

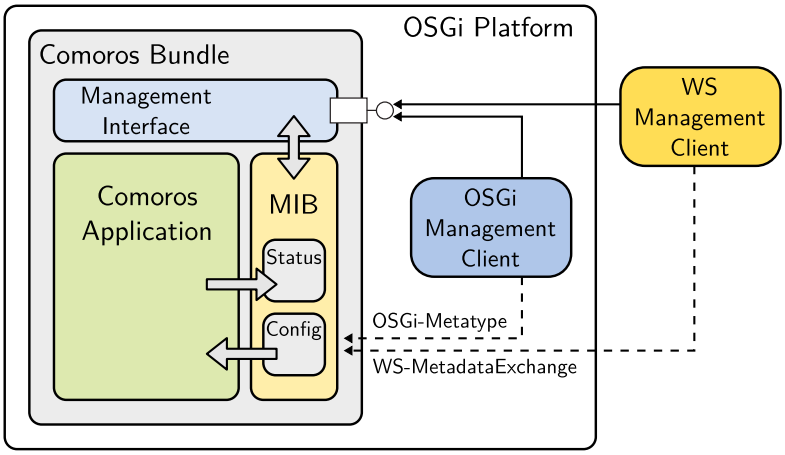


Abbildung 6.28: Überblick über die Comoros-Managementstruktur

1. Zentrale Konfiguration
2. Dezentrale Konfiguration

Die dezentrale Konfiguration hat einen gravierenden Nachteil. Werden Konfigurationen für die einzelnen Komponenten der Comoros-Middleware erstellt, so können diese nicht ohne Weiteres abgeglichen werden. Ohne diesen Abgleich kann aber leicht eine inkonsistente Gesamtkonfiguration entstehen und so Fehlerfälle herbeigeführt werden. Um das zu vermeiden, wird in die Comoros-Architektur eine weitere Komponente – das Management-Bundle – eingeführt, über das eine zentrale Konfiguration der gesamten Middleware möglich ist. Dieser Vorgang ist in Abbildung 6.29 illustriert.

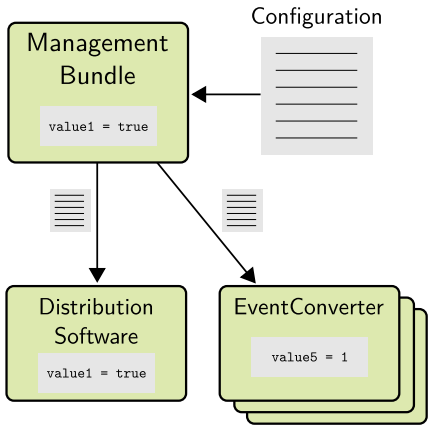


Abbildung 6.29: Konzept der zentralen Konfiguration

Dem Management-Bundle wird eine Konfiguration für das gesamte System übermittelt. Aus dieser Konfiguration werden nun Subkonfigurationen für die einzelnen Komponenten erstellt und an diese weitergereicht. Dabei müssen die Informationen für die einzelnen Komponenten nicht unbedingt direkt angegeben, sondern können auch aus anderen Werten abgeleitet werden. Innerhalb des Management-Bundles werden die übermittelten

Konfigurationen auf Vollständigkeit und Konsistenz überprüft um sicherzustellen, dass gültige Subkonfigurationen erstellt werden können. Um einen sinnvollen Betrieb der Comoros-Middelware zu garantieren, starten die einzelnen Komponenten nur, wenn eine gültige Konfiguration vorliegt. Aus diesem Grund ist im Management-Bundle eine Standard-Konfiguration hinterlegt, die an die einzelnen Komponenten verteilt wird solange keine Konfiguration durch den Benutzer übermittelt wurde.

Das Setzen einer Konfiguration betrifft nur einen Teil des Management-relevanten Zustands, nämlich die Konfigurationsvariablen. Diese sind in einer MIB-ähnlichen Struktur hinterlegt, wie in Abbildung 6.30 abgebildet.

Jede OSGi-Plattform wird durch eine Comoros-Instanz repräsentiert und enthält den gleichen Variablenraum. Dieser ist in vier Teile gegliedert, welche die einzelnen Comoros-Bundles – *DSW*, *Logger*, *EventConverter*, *DPWS* – darstellen. Der DSW-Teil wiederum ist in die Bereiche *Client*, *Server* und *General* unterteilt. Im Server-Teil wird festgelegt welche OSGi-Services für einen entfernten Zugriff zur Verfügung gestellt werden. Dazu wird eine Filter-Liste angelegt, deren Einträge aus den Filterausdrücken und den jeweils zugehörigen Endpunkten, den Bindings und dem Encoding bestehen. Im Client-Teil wird auch eine Filter-Liste verwaltet. Neben den Filterausdrücken wird hier die Art des Proxys festgelegt – Dynamischer Proxy oder Proxy-Bundle – und die Generierung der Service-Interfaces angestoßen. Im EventConverter wird auf der Server-Seite konfiguriert welche Events versendet und auf der Client-Seite von welchen Plattformen Events empfangen werden. Im DPWS-Teil werden generelle Angaben zu den Transportprotokollen verwaltet und der Logging-Mechanismus konfiguriert.

Jeder Eintrag im MIB-Schema ist über den Pfad eindeutig identifizierbar und ebenso eindeutig definiert. Das folgende Beispiel zeigt die Struktur eines Eintrags. Der `ObjectIdentifier` identifiziert jede Konfigurationsvariable eindeutig. Während die `Description` eine für den Menschen lesbare Beschreibung angibt, definieren `DataType`, `DefaultValue` und `ValueRange` die Variable aus programmiertechnischer Sicht.

```
1 ObjectIdentifier := {org.ws4d.comoros.platform1.dsw.client.hideRS}
2
3 Description: = {Hide remote services for local clients . This is useful
4    if clients should not use remote services due to network related
5    problems (e.g. latency) as long as they do not explicitly ask for one.}
6
7 DataType := Boolean
8 DefaultValue := true
9 ValueRange := [true, false]
```

Die Schnittstelle zum Management-System soll von OSGi- und Webservice-Clients aufgerufen werden können. In OSGi wird dafür der *Configuration Admin Service* verwendet. Der Config Admin Service verwaltet persistente Konfigurationen, die mit einer speziellen Identifikationsnummer, der *Persistence Identification (PID)*, belegt werden. Die PID ist mit der Konfigurationsschnittstelle eines Bundles, dem `ManagedService`, fest verknüpft. Neue Konfigurationen oder Änderungen an bestehenden Konfigurationen können so einer Schnittstelle zugeordnet und dem Bundle übermittelt werden. Die Konfigurationen selbst werden von Management-Clients direkt dem Config Admin übergeben. Für den

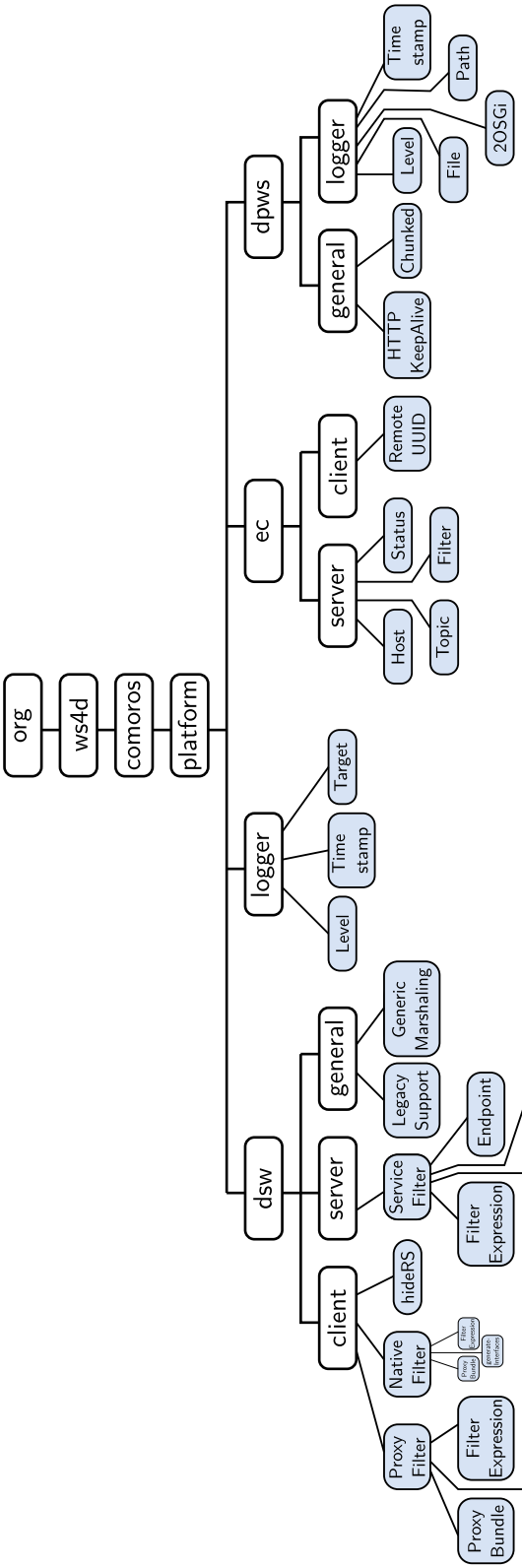


Abbildung 6.30: Zustandsraum der Comoros-Middelware in Form einer MIB-ähnlichen Struktur

Konfigurationsvorgang von Comoros durch einen lokalen OSGi-Management-Client ergibt sich das Schaubild Abbildung 6.31.

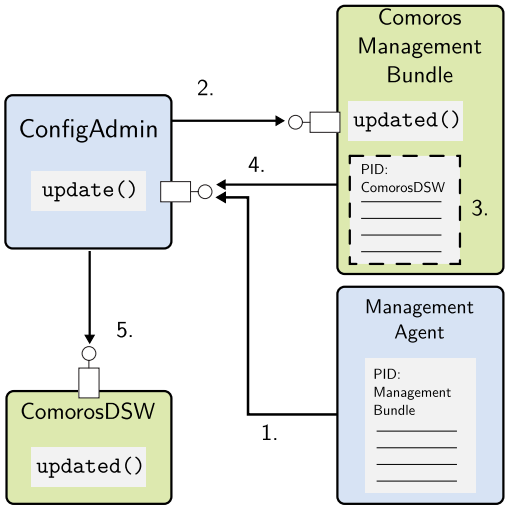


Abbildung 6.31: Zentrales Management mittels des OSGi-Config-Admins

Ein Management-Agent übermittelt eine Konfiguration an den Config-Admin-Service (1.). Daraufhin wird die `updated()`-Methode des Managed-Service mit der entsprechenden PID aufgerufen (2.). Das Management-Bundle erstellt nun eine gültige Subkonfiguration beispielhaft für die Comoros-DSW (3.) und übermittelt diese wieder an den Config-Admin-Service (4.). Abschließend wird der Managed-Service dieser Komponente mit der entsprechenden Konfiguration aufgerufen (5.).

Für eine entfernte Konfiguration der Comoros-Middleware über Webservice-Clients wird nun einfach die `ManagedService`-Schnittstelle des Management-Bundles als DPWS-Service von Comoros selber publiziert. Dazu erstellt die Distribution Software für diesen Service einen speziellen DPWS-Skeleton, dessen besondere Funktionsweise in Abbildung 6.32 abgebildet ist.

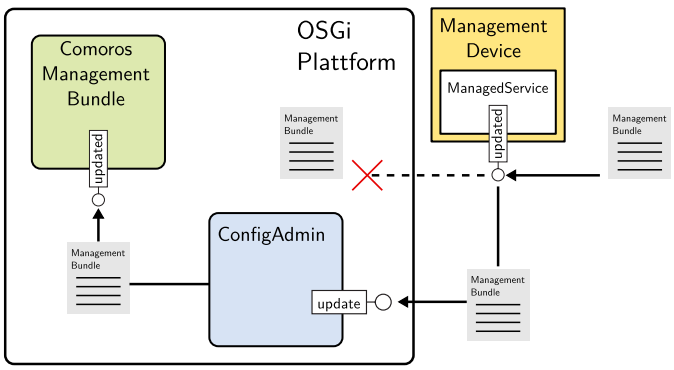


Abbildung 6.32: Spezieller Skeleton für einen OSGi-ManagedService

Wie gewohnt ist der DPWS-Skeleton die Abbildung eines OSGi-Service, in diesem Fall des Managed-Service. Ein Aufruf der `updated`-Operation des Skeletons, also die

Übermittlung einer neuen Konfiguration, darf nun aber nicht direkt an die `updated()`-Methode des Management-Bundles übertragen werden. Stattdessen wird der Aufruf an den Configuration-Admin-Service weitergeleitet, der wiederum die Konfiguration an das Management-Bundle weiterleitet. Ohne diesen Umweg würde der Persistenz-Mechanismus des Config-Admin umgangen werden und zukünftige Probleme wären nicht ausgeschlossen. Um das Konzept der zentralen Konfiguration von Comoros zu erhalten wird dieser spezielle Skeleton ausschließlich für den Managed-Service des Management-Bundles erstellt, die Managed-Services der weiteren Comoros-Komponenten werden verworfen und können somit nicht über entfernte Clients aufgerufen werden.

Neben der entfernten Konfiguration durch Webservice-Clients wird die Konfiguration auch durch entfernte OSGi-Clients ermöglicht. Auf `Plattform 1` befindet sich im Folgenden ein Managed-Service und auf `Plattform 2` sein Proxy. Aufrufe an den durch Comoros erstellten Skeleton werden wie beschrieben an den Configuration-Admin weitergeleitet (siehe Abbildung 6.32). Der im Proxy registrierte Managed-Service wird auch durch einen lokalen Configuration-Admin kontrolliert. Ein Aufruf des Configuration-Admin auf `Plattform 2` wird nun, wie in Abbildung 6.33 illustriert, an das Bundle auf `Plattform 1` weitergeleitet.

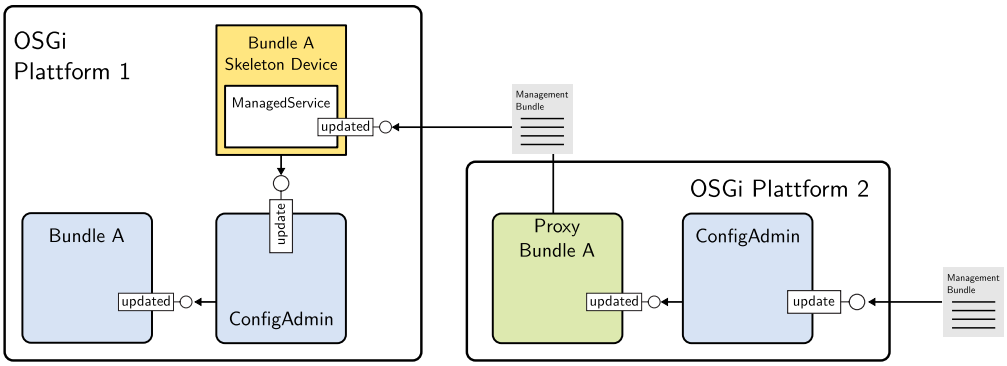


Abbildung 6.33: Konfiguration eines entfernten OSGi-Bundles mittels des Configuration-Admins

Um die Konsistenz auf beiden Plattformen zu sichern, müssen Änderungen auf `Plattform 1` auch auf `Plattform 2` bekannt gemacht werden. Abbildung 6.34 beschreibt diesen Vorgang. Der DPWS-Skeleton wird um eine Event-basierte Operation `configurationChanged` erweitert, die ein DPWS-Event mit veränderten Konfigurationen veröffentlicht. Dieses Event wird durch ein spezielles OSGi-Event des lokalen Configuration-Admin ausgelöst, das dieser bei eine geänderten Konfiguration versendet. Der Proxy auf `Plattform 2` abonniert die Event-basierte Schnittstelle des Skeletons und erhält so die veränderte Konfiguration. Um diese schließlich lokal zu publizieren übermittelt der Proxy die Konfiguration an den zuständigen Configuration-Admin-Service.

Damit der Management-Agent weiß, wie die erwartete Konfiguration zu einer bestimmten PID aussieht, muss er die Möglichkeit haben eine Beschreibung der Konfiguration, z. B. Namen und Typen der einzelnen Attribute, zu ermitteln. Diese Aufgabe erledigt der *Metatype Service* [The12a]. Der Metatype-Service bietet eine einheitliche Schnittstelle

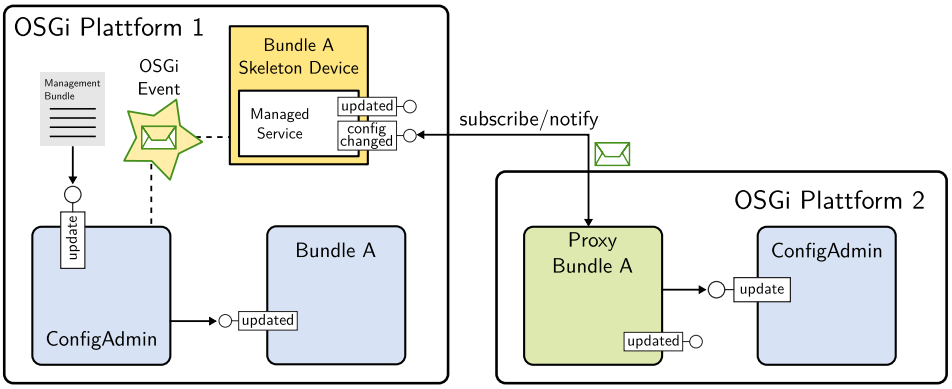


Abbildung 6.34: Sicherung der Konsistenz bei entferntem Management von OSGi-Bundles

für den Zugriff auf Metadaten, die von Management-Agenten genutzt wird. Metadaten können für beliebige Entitäten angelegt werden, in der Praxis ist aber vor allem das Zusammenspiel mit dem Configuration-Admin relevant. Für entfernte Webservice- und OSGi-Clients müssen diese Metadaten auch einsehbar sein. Bei der Erstellung eines DPWS-Skeletons für einen Managed-Service werden dazu die zugehörigen Metadaten vom Metatype-Service abgerufen und für die `updated`-Operation in einem passenden XML-Schema hinterlegt. Entsprechende Clients haben somit genaue Informationen über die Struktur der erwarteten Konfiguration. Ein Proxy, der für diesen Skeleton von Comoros erstellt wird, registriert nun neben der Service-Schnittstelle für die `updated`-Operation auch einen *Metatype Provider* über den die Metadaten-Informationen abgerufen werden können. Da der Metatype-Service eine uniforme Schnittstelle aller Metatype-Provider anbietet, können lokale Management-Clients wie gewohnt die Metadaten abrufen.

Monitoring

Für eine adaptive Systemlandschaft ist die Überwachung der internen Zustände eines Systems eine notwendige Voraussetzung. So können zur Laufzeit Fehler und auftretende Ereignisse frühzeitig erkannt werden und durch eine passende Reaktion auf diese Ereignisse die allgemeine Zuverlässigkeit deutlich erhöht werden.

Für die standardisierte Überwachung eines OSGi-basierten Systems hat die OSGi-Allianz den *Monitor Admin Service* [The12a] spezifiziert. Jede Komponente, die Zustandsinformationen nach außen mitteilen will, muss nun das `Monitorable`-Interface implementieren und als Service im Framework registrieren. Für jeden zu überwachenden Wert wird ein Objekt vom Typ `StatusVariable` angelegt, das fortan über den Monitorable-Service von jedem Monitorable-Client abgefragt werden kann. Der Wert einer Statusvariable ist in den Datentypen auf Long, Double, Boolean und String beschränkt, so dass für eine verteilte Nutzung dieser Management-Schnittstelle keine Probleme auftreten. Bei den Statusvariablen wird zwischen zwei Typen unterschieden:

1. Monitoring-Daten, welche die Performanz eines Systems überwachen oder Logging-Daten für die Dokumentation der Abläufe einzelner Prozesse werden passiv im laufenden Betrieb gesammelt. Diese Art der Daten sind dann durch Monitoring-

Clients manuell abrufbar oder werden den Client in einem fest definiertem zeitlichen Intervall übermittelt.

2. Auftretende Fehler, Mangel an Ressourcen oder andere spezielle Ereignisse müssen dem Monitoring-Client aktiv übermittelt werden, sobald sie auftreten, da hier eine direkte Reaktion erfolgen muss.

Abgesehen von der manuellen Abfrage der Statusvariablen erfolgt die Übermittlung der Variablen über den OSGi-Event-Mechanismus in Form von `Monitoring Events`. Jeder lokale OSGi-Client kann diese Events empfangen und verarbeiten. Die Events werden entweder eigenständig durch die überwachte Komponente ausgelöst oder über *Monitoring Jobs*, die der Monitoring-Client anlegt.

Die Statusvariablen der Comoros-Middleware sind in Abbildung 6.35 abgebildet. Sie sind logisch in drei Teile gegliedert. Ein Teil der Variablen beschreibt den aktuellen Zustand des Systems, der zweite Teil sammelt Daten für statistische Auswertungen und der letzte Teil übermittelt auftretende Fehler. Da Statusvariablen im Sinne des OSGi-Monitor-Admin-Service keine komplexen Datenstrukturen wie Collections oder Maps enthalten dürfen, aber Daten in Bezug auf einzelne Proxys und Skeletons gesammelt werden sollen, müssen Statusvariablen dynamisch erzeugt werden. Wird beispielsweise ein neuer Skeleton mit der Id `1` in der Plattform erzeugt, so werden in Bezug zu diesem Skeleton die Variablen `Endpoint`, `Binding`, `Encoding` und `Deployment` erzeugt. Der Zugriff auf die `Encoding`-Variable für diesen speziellen Skeleton erfolgt nun über den Pfad:

```
1 org\ws4d\comoros\platform\state\server\1\Encoding
```

Auf diese Weise wird der Baumstruktur der MIB ein neuer Teilbaum hinzugefügt.

Das Sammeln der Informationen über den aktuellen Zustand des Systems ist besonders wichtig, da eine Diskrepanz zwischen der Konfiguration und dem aktuellen Zustand bestehen kann, beispielsweise falls die Erzeugung eines Skeletons oder Proxys fehlschlägt oder sich eine Konfiguration auf einen aktuell nicht im System befindlichen Service bezieht. Mit Hilfe der Daten aus dem Statistik-Teil können in einem Management-System umfangreiche Analysen über das System vorgenommen werden und aufgrund dieser Erkenntnisse eine geeignete Konfiguration nach den in Tabelle 5.3 aufgeführten Konfigurationsempfehlungen vorgenommen werden. Ein spezieller Datenpunkt in diesem Teilbereich ist die Variable `IncreasedMarshalingEffort`, deren Zustandsänderungen direkt über ein `Monitor Event` an den Management-Client übermittelt werden. Wenn im Zuge des Marshaling-Prozesses definierte Grenzwerte in Verarbeitungszeit und generierter XML-Schemagröße überschritten werden, so wird dieses angezeigt. Dies ist dann ein deutlicher Hinweis für den Entwickler für die entsprechende Klasse ein Transformationspattern zu erzeugen. Ebenfalls automatisch übermittelt werden Zustandsänderungen in den Variablen aus dem Fehler-Teil.

Zur Ausweitung der Überwachung von OSGi-Komponenten durch entfernte OSGi-Clients wurde der *Device Management Tree (DMT)* [The12a] spezifiziert. Eine einfache entfernte Überwachung, die zusätzlich auch Webservice-basierte Clients mit einschließt,

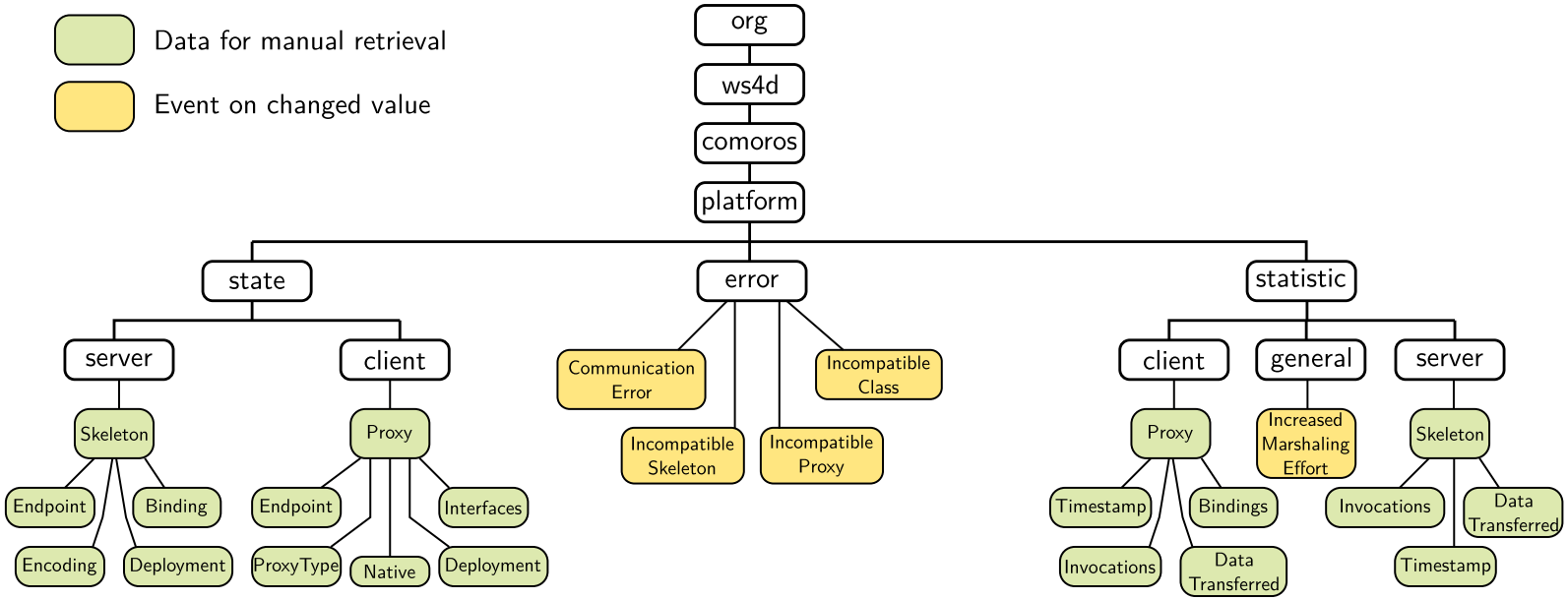


Abbildung 6.35: Zustandsraum der Comoros-Middelware in Form einer MIB-ähnlichen Struktur – Statusvariablen

kann auch durch die Comoros-Middelware realisiert werden. Dazu wird der EventConverter (siehe Abschnitt 6.2) verwendet. Er erzeugt aufgrund einer hinterlegten Konfiguration Monitoring-Jobs und versendet die dadurch ausgelösten Monitoring-Events an entfernte OSGi-Plattformen. Die Event-Schnittstelle des EventConverters kann nun auch durch entsprechende DPWS-Clients abonniert werden. Um den Overhead zu reduzieren wurde im EventConverter eine Schnittstelle speziell für Monitoring-Events hinzugefügt, um so den Empfang von allgemeinen OSGi-Events durch Monitoring-Clients grundsätzlich zu verhindern.

6.6.3 Werkzeugunterstützung

Die Konfiguration der Comoros-Middleware kann mit Hilfe von in der Metatype-Service-Spezifikation definierten Management-Agents durchgeführt werden. Die Informationen zur Darstellung der Konfiguration wird mit Hilfe des Metatype-Service abgerufen, die durch das Werkzeug erstellte Konfiguration dann an den Configuration-Admin-Service übergeben. Es existieren einige wenige Implementierungen solcher Agents in Form von GUI-basierten Clients, wie z. B. die Knopflerfish-ConfigManagement-Erweiterung [1]. Da diese Implementierung aber an den Knopflerfish-Desktop gebunden ist und somit eine große Anzahl an Abhängigkeiten hat, ist im Rahmen dieser Arbeit der *Comoros.ConfigAgent* entwickelt worden.

Der Comoros.Config Agent hat die einfache Aufgabe alle Konfigurationen einer OSGi-Plattform, die durch einen Metatype-Service beschrieben werden, darzustellen, diese durch den Benutzer manipulieren zu lassen, und anschließend an den Configuration-Admin-Service zu übermitteln. Das Werkzeug ist explizit nicht nur auf die Konfiguration von Comoros beschränkt. Diese wir ausschnittsweise in Abbildung 6.36 dargestellt. Der Comoros.ConfigAgent hat außer der notwendigen Abhängigkeit zum Metatype und Configuration-Admin-Service keine weitere Abhängigkeit und kann somit einfach in allen OSGi-Systemen mit grafischer Benutzerschnittstelle verwendet werden.

1 Ed.: Knopflerfish, URL: http://www.knopflerfish.org/releases/5.0.0/docs/bundledoc/cm/index.html, Abruf: 09.06.2015

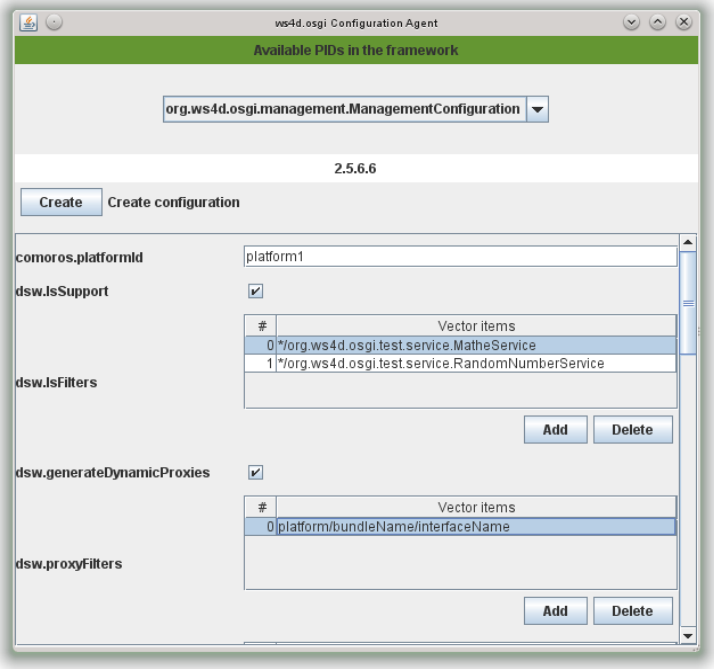


Abbildung 6.36: Konfiguration mittels des Comoros.ConfigAgent

6.7 Sicherheit

Die Sicherheit in der durch Comoros angebotenen Kommunikation umfasst drei Bereiche, die sich im wesentlichen auf die durch die DPWS-Spezifikation vorgesehenen Mechanismen beschränken: Die Sicherheit in der Kommunikation, das Absichern des Discovery-Prozesses, und ein Authorisierungs-Mechanismus.

- **Sicherheit im Discovery-Mechanismus**
  Das Sicherheitsmodell für den Discovery-Mechanismus sichert im Sinne der Schutzziele in der Informationssicherheit die *Integrität* und die *Verbindlichkeit*. Hierbei werden nur die UDP-basierten Nachrichten – Unicast und Multicast – des Discovery-Prozesses behandelt. Diese umfassen `Hello-`, `Bye-`, `ProbeMatches-`, `ResolveMatches`-Nachrichten auf der Server-Seite und `Probe-` und `Resolve`-Nachrichten auf der Client-Seite.

  Der Mechanismus zur Erstellung der Signatur entspricht der gängigen Praxis. Die zu signierende Nachricht wird in eine kanonische Form überführt und über diese Form ein Hash-Wert gebildet. Über diesen Hash-Wert bildet der Sender der Nachricht mit seinem privatem Schlüssel eine digitale Signatur. Diese Signatur wird zusammen mit der Schlüssel-Id zusammen mit den Nutzdaten in einer XML-Datei übertragen. Dazu wird in der WS-Discovery-Spezifikation [Bea09] ein *Compact Signature Format* definiert. Der Empfänger der Nachricht entschlüsselt den Hash-Wert anschließend mit dem öffentlichen Schlüssel des Senders und vergleicht den so

erhaltenen Hash-Wert mit dem selber berechneten Hash-Wert der Nutzdaten.

Ein Service, der nun eine `Probe`- oder `Resolve`-Nachricht empfängt, muss darauf nicht antworten, falls die Signatur dieser Nachrichten nicht verifiziert werden kann. Analog dazu kann ein Client nun `ProbeMatches` oder `ResolveMatches` bei einer erfolglosen Verifizierung verwerfen.

- **Sicherheit in der Kommunikation**
  Sicherheit in der Kommunikation bedeutet die Gewährleistung der *Vertraulichkeit*, der *Integrität* und eine *Authentifizierung*. Im Sinne von DPWS werden an dieser Stelle nur TCP-basierte Nachrichten, wie `Invoke`-, `Get`- und `GetResponse`- oder `GetMetadata`- und `GetMetadataResponse`-Nachrichten betrachtet. Somit ist auch ersichtlich, dass hier nur das SOAP-over-HTTP-Binding unterstützt wird. Andere Transportbindungen, wie das effizientere SOAP-over-CoAP müssen auf Sicherheitsmerkmale verzichten.

  Um die Schutzziele zu gewährleisten kann in Comoros mit Hilfe der JMEDS-Bibliothek eine auf SSL basierende sichere *HTTPS*-Verbindung aufgebaut werden. Hier greifen die bekannten Mechanismen. In der Handshake-Phase werden mit Hilfe von Client- und Server-Zertifikaten die Identitäten der Kommunikations-Endpunkte geprüft. Zur Verschlüsselung der Daten werden anschließend Sitzungsschlüssel generiert, mit deren Hilfe auch Integritätsprüfungen während der Datenübertragung realisiert werden.

- **Autorisierungs-Mechanismus**
  Der *Authorization Manager* ist nicht Teil von *WS-DD*, sondern Teil der JMEDS-Implementierung. Dieser Mechanismus führt auf Ebene der Nachrichten-Typen Authorisierungs-Überprüfungen ein. Jeder Nachrichten-Typ besteht aus einer Menge von Subelementen – beispielsweise Empfänger- oder Absenderadresse – für die jeweils Regeln implementiert werden können. So kann z. B. überprüft werden, ob eine Absender `A` eine Operation `B` aufrufen darf. Für die Implementierung der Regeln gibt es keine Vorgaben.

6.8 Zwischenfazit

Die hier vorgestellten Komponenten erweitern die Comoros-Kernarchitektur, und damit die RS-Spezifikation, um Bereiche, die eine komfortable und effiziente Entwicklung von verteilten OSGi-Anwendungen ermöglichen. Dabei bleiben die Grundprinzipien der Kernarchitektur unangetastet. Die zusätzlichen Anforderungen im Vergleich zur RS-Spezifikation ergeben sich vor allem aus den Anwendungsdomänen. Anwendungsentwicklung in Domänen wie der Gebäudeautomatisierung oder im AAL-Bereich finden zumeist in heterogenen, gewachsenen Strukturen statt, in denen eine Vielzahl von Geräten, Diensten und Technologien eingesetzt werden. Um in dieser Umgebung eine verteilte OSGi-basierte Anwendung zu entwickeln, ist die Erweiterung der Kernarchitektur unumgänglich.

Die in dieser Arbeit entwickelten Erweiterungen umfassen dabei unterschiedlichste Bereiche: Das generisches Marshaling erhöht die Kompatibilität von OSGi-basierten Services und Clients, da die Restriktionen der Datentypen quasi aufgehoben werden. In

der Event-basierten Kommunikation werden nun alle in OSGi anwendbaren Mechanismen unterstützt. Beide Punkte zusammen sind Voraussetzung für die Integration von Altkomponenten in das verteilte System. Weitere Punkte betreffen die Integration von Geräten, die Unterstützung verschiedener Kommunikationsprotokolle, die Einführung einer umfassenden Management-Schnittstelle und die Umsetzung von Sicherheitsmechanismen.

Kapitel 4 enthält eine ausführliche Analyse aller für Comoros relevanten Anwendungsfälle, die teilweise bereits durch die Kernarchitektur erfüllt werden (siehe Abschnitt 5.5). Durch die erweiterte Architektur werden nun alle Weiteren abgedeckt:

- **Ein OSGi-Client nutzt einen entfernten OSGi-fremden Service**
  OSGi-fremde Services können durch Softwarekomponenten oder durch Geräte angeboten werden. In diesem Fall kann der Service einerseits direkt durch ein SOA-fähiges Geräte angeboten werden oder durch ein Gerät, dessen Funktionalität im Rahmen der Geräteintegration als Service abstrahiert wurde. Die Nutzung dieser Services erfolgt nach den bekannten Prinzipien.
- **Ein OSGi-fremder Client nutzt einen entfernten OSGi-Service**
  Auch hier erfolgt die Kommunikation über bekannte Prinzipien. Durch die Unterstützung weiterer Kommunikationsprotokolle wurde die Menge der möglichen Kommunikationspartner deutlich erhöht.
- **Event-basierte Kommunikation**
  Die RS-Spezifikation sieht lediglich eine implizite Event-basierte Kommunikation über die Verteilung von `EventHandler`-Services vor. In Comoros können alle lokal in OSGi verwendbaren Event-Mechanismen auch in verteilten Umgebungen eingesetzt werden, was zusätzlich die Integration von Altkomponenten vereinfacht. Die Umsetzung ist weiterhin so konfigurierbar, dass nur eine minimale Anzahl von Nachrichten zwischen den Plattformen ausgetauscht wird, so dass der Einsatz von Comoros in Ressourcen beschränkten Umgebungen weiter gewährleistet ist.
- **Geräteintegration**
  Geräte, die ihre Funktionalität nicht über SOA-basierte Schnittstellen, sondern beispielsweise über serielle Schnittstellen anbieten, können über das Konzept der Geräteintegration in eine OSGi-Plattform integriert werden. Dabei wird die Funktionalität in Form von OSGi-Services angeboten. Geräteprofile erleichtern die Entwicklung und Austauschbarkeit von Geräten. Die angebotenen Services können wie jeder andere OSGi-Service auch über die Grenzen einer Plattform verteilt werden.
- **Zugriffstransparenz**
  Der Zugriff auf einen entfernten Service unterscheidet sich beim Einsatz von Comoros für einen Client nicht von einem lokalen Zugriff. Diese Anforderung gilt auch bei der Integration OSGi-fremder Services, da auch hier das Proxy-Skeleton-Prinzip umgesetzt wird. Auf diese Weise können Altkomponenten auch OSGi-fremde Services ohne Modifikationen verwenden.
- **Adaptierbarkeit**
  Die Anpassung an wechselnde Anforderungen ist eine der wichtigsten Voraussetzun-

gen für die Realisierung von effizienten Systemen. Comoros bietet Schnittstellen zur Konfiguration und zur Überwachung des Systems an, über die Comoros in umfassende Management-Systeme integriert werden kann, und so Teil einer adaptiven Systemlandschaft wird. Um die einfache Integration in Management-Systeme zu gewährleisten ist der Zustandsraum von Comoros über eine MIB-ähnliche Struktur klar definiert und die Zugriffe erfolgen über bestehende Management-Standards. So kann Comoros, auch zur Laufzeit, an den speziellen Anwendungsfall angepasst werden und eine effiziente Realisierung anbieten.

- **Nutzerkomfort**
  Ein hoher Komfort für den Anwendungsentwickler ist Voraussetzung für eine hohe Akzeptanz der Comoros-Middleware. In der erweiterten Architektur wird dieser Komfort weiter erhöht. Durch das generische Marshaling kann der Benutzer, bis auf wenige Einschränkungen, sämtliche Java-Datentypen in einer verteilten Umgebung verwenden ohne sich um das Marshaling zu kümmern. Weiterhin existiert ein Werkzeug, dass die Integration von SOA-fähigen Geräten stark vereinfacht, und das Daten-Mapping grafisch realisiert. So können Transformationspattern für die Services der Geräte erzeugt werden und diese anschließend einfach innerhalb von OSGi verwendet werden. Daneben existiert auch ein Werkzeug zur grafischen Konfiguration von Comoros. Neben diesen Vereinfachungen wird der generelle Komfort auch durch den erhöhten Funktionsumfang erhöht, da der Entwickler nun weitere OSGi-Konzepte in verteilten Umgebungen nutzen kann, ohne dass sich die Nutzung von der in lokalen Umgebungen unterscheidet.

- **Altkomponenten**
  Neben dem Nutzerkomfort ist die Verwendung von bestehenden Komponenten in verteilten Umgebungen der zweite wichtige Faktor für eine hohe Nutzerakzeptanz. Das erweiterte Marshaling, die Event-basierte Kommunikation und die Konfiguration von Kommuniaktionsbeziehungen sind wichtige Bausteine der erweiterten Architektur hinsichtlich dieses Aspektes.

- **Effizienz**
  Sowohl in der Kernarchitektur, als auch in der erweiterten Architektur wird die Entwicklung effizienter verteilter OSGi-Anwendungen unterstützt. Gerade durch die vielfältigen Möglichkeiten der Konfiguration können sowohl der Footprint, die Verarbeitungszeit als auch der Datenverkehr für jeden Anwendungsfall minimiert werden. Im Bereich des erweiterten Marshaling hat der Entwickler zudem die Möglichkeit über Transformationspattern die Verarbeitung von XML-Nachrichten zu optimieren und so die Verarbeitungszeit und den Datenverkehr auch an dieser Stelle weiter zu minimieren. Im Bereich der Kommunikation können nun neben dem DPWS auch weitere effiziente Protokolle verwendet werden. Aber auch das DPWS-Protokoll kann durch die Verwendung des EXI-Kompressionsverfahren oder durch effiziente Transportbindungen, wie dem SOAP-over-CoAP-Binding, weiter auf die besonderen Bedürfnisse Ressourcen beschränkter Umgebungen angepasst werden.

Die erweiterte Architektur schließt nun die Lücke, welche, nach der Analyse in Kapitel 3 und Kapitel 4 zur komfortablen Anwendungsentwicklung innerhalb flexibler Dienste- und Gerätesysteme, durch die Kernarchitektur offen geblieben war. Im Vergleich zu den vorgestellten Lösungen in Kapitel 3 bietet Comoros nun den umfassendsten Funktionsumfang und bleibt dennoch, ganz im Gegensatz zu vielen der anderen Projekte, standardkonform. So können OSGi-Anwendungsentwickler verteilte Lösungen realisieren ohne vom Programmierparadigma lokaler Anwendungen abzuweichen.

Einige Projekte, wie das *Eclipse Communication Framework (ECF)* bieten zwar eine wesentlich größere Anzahl an unterschiedlichen Kommunikationsprotokollen an, durch die modulare Struktur von Comoros können aber auch hier beliebige weitere Protokolle einfach integriert werden. Genau wie in der Kernarchitektur, wird auch in der erweiterten Architektur von der Verwendung schwergewichtiger Bibliotheken abgesehen, so dass Comoros trotz des umfassenden Funktionsumfangs leichtgewichtiger als andere Lösungen, wie das EXF, Apache CXF, Newton oder ClusteredOSGi bleibt. In der Fragestellung des Nutzungskomforts ist Comoros durch die Beibehaltung des OSGi-Programmiermodells anderen Projekten weiter im Vorteil, eine explizite Werkzeugunterstützung existiert zudem auch nur in einem anderen Projekt. R-OSGi bietet ein Deployment-Tool [Rel07b], mit dem Services grafisch auf verschiedene Knoten verteilt werden und Bindungen aufgebaut werden können.

Insgesamt bietet Comoros durch den höchsten Funktionsumfang im Bereich OSGi-basierter Middleware-Lösungen, durch eine effiziente Umsetzung und einen hohen Nutzungskomfort gute Voraussetzungen für eine hohe Akzeptanz bei der Entwicklung verteilter OSGi-Anwendungen.

7 Evaluierung

Die Nutzerakzeptanz für Softwareprodukte korreliert zumeist direkt mit der angebotenen Softwarequalität. Auch für die Comoros-Middleware gilt diese Aussage, so dass in diesem Kapitel die Qualität von Comoros evaluiert wird. Dabei werden vorrangig folgende Fragestellungen beantwortet:

- Wie ist Softwarequalität im Sinne von Comoros definiert?
- Welche Auswirkungen hat die Einführung der erweiterten Architektur auf verschiedene Aspekte der Softwarequalität?
- Wie verhält sich die Leistungsfähigkeit und Effizienz von Comoros?

Unterabschnitt 2.1.8 setzt sich bereits mit der vielfältigen Definition von Softwarequalität auseinander. Da es keine eindeutige und allgemeingültige Definition gibt, ist es im aktuellen Stand der Technik üblich die Softwarequalität über so genannte Qualitätsmodelle zu erfassen. Diese bieten eine Abstraktion von Merkmalen, Kriterien und Eigenschaften um Softwarequalität systematisch zu definieren, zu beurteilen und zu messen. Qualitätsmodelle können in unterschiedliche Arten unterteilt werden. Normative Gesamtmodelle unterstellen eine Allgemeingültigkeit und können in diesem Sinne auf jedes Softwareprodukt angewendet werden. Interessanter für Comoros ist allerdings die Erstellung eines angepassten Qualitätsmodells, das die individuellen Anforderungen an Comoros berücksichtigt und dementsprechende Qualitätsmerkmale einführt.

Die Erstellung eines individuellen Qualitätsmodells umfasst entweder die komplette Neuerstellung, auch auf Basis von Metamodellen, oder die Anpassung von bestehenden Modellen. Für letzteren Prozess wurden verschiedene normative Modelle analysiert [Boe76; Deu86; McC77] und schlussendlich das Modell von Schweiggert [Sch85] als

Grundlage für ein individuelles Qualitätsmodell ausgewählt. Das so entstandene Comoros-Qualitätsmodell ist in Abbildung 7.1 zu sehen.

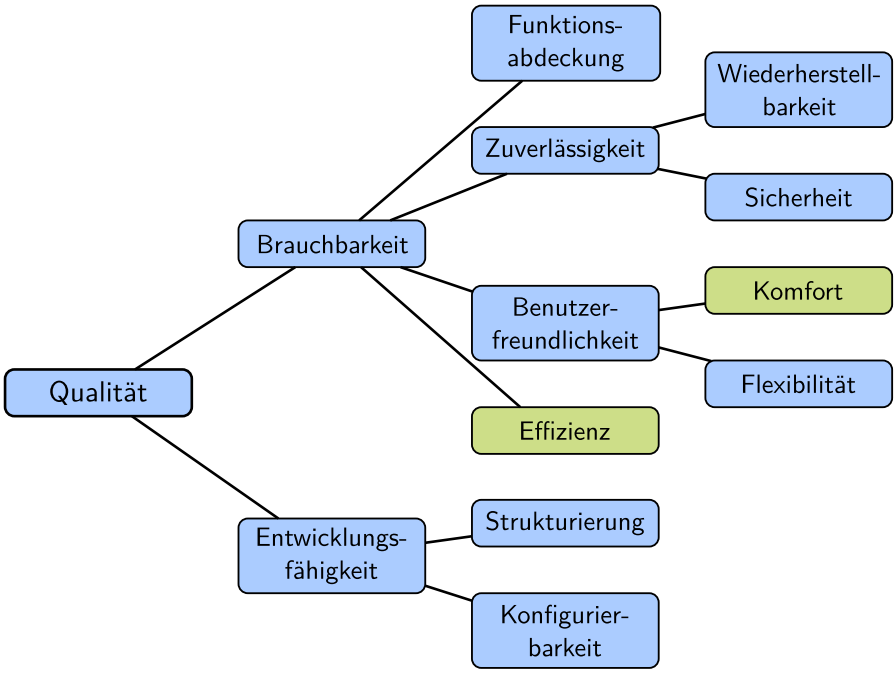


Abbildung 7.1: Comoros-Qualitätsmodell nach Schweiggert

Das Modell beinhaltet drei Hierarchieebenen und insgesamt zwölf Qualitätsmerkmale. Die erste Verzweigung unterscheidet die nach ISO 25000 eingeführte *interne Qualität* und *externe Qualität*. Die interne Qualität beschreibt die internen Merkmale der Software und spiegelt damit den Blick des Softwareentwicklers wieder. Die Merkmale der externen Qualität sind dagegen für den Nutzer von Interesse und verdecken die Sicht auf die internen Programmstrukturen. Diese Merkmale verdeutlichen, inwieweit die Software in der Lage ist, die Anforderungen der Nutzer zu erfüllen.

Dem Pfad der internen Qualität folgend, unterteilt sich die *Brauchbarkeit* in die Merkmale *Funktionsabdeckung*, *Zuverlässigkeit*, *Benutzerfreundlichkeit* und *Effizienz*, die dann teilweise weiter gegliedert werden. Die *Entwicklungsfähigkeit*, also die externe Qualität, wird durch die Merkmale *Strukturierung* und *Konfigurierbarkeit* spezialisiert.

Für jedes Blatt des Baumes werden im folgenden über den GQM-Ansatz (siehe Unterabschnitt 2.1.8) Metriken hergeleitet um die Merkmale vermessen zu können. Ein besonderes Augenmerk wird dabei auf die Merkmale *Effizienz* und *Komfort* gelegt, da diese Merkmale von essentieller Bedeutung sind und die in den vorherigen Kapitel gestellten Anforderungen und Behauptungen verifizieren.

7.1 Wiederherstellbarkeit

Die Wiederherstellbarkeit beschreibt, ob und wie ein Programm nach einem Programmabbruch wieder in den letzten funktionsfähigen Zustand zurückversetzt werden kann. Bei dem Abbruch kann es sich dabei sowohl um ein reguläres Beenden der Software handeln,

als auch um einen ungewollten Systemausfall. Für beide Fälle darf sich der Zustand nach dem Neustart nicht unterscheiden. Tabelle 7.1 zeigt die GQM-Zerlegung für die Wiederherstellbarkeit und die somit abgeleiteten Metriken für die Vermessung.

Tabelle 7.1: GQM-Zerlegung des Zielfaktors Evaluierung der Wiederherstellbarkeit

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Wiederherstellbarkeit
Objekt	von Comoros
Perspektive	aus Sicht des Nutzers
Frage	Metrik
Q1: Welchen Umfang hat der Zustandsraum?	**M1.1:** Anzahl der Konfigurationsvariablen
	M1.2: Anzahl der Systemzustände
Q2: Wird der gesamte Zustandsraum persistent gehalten?	**M2.1:** Anzahl persistenter Zustände
Q3: Wird der Zustandsraum beim Neustart wiederhergestellt?	**M3.1:** Anzahl der wiederhergestellten Zustände

Messergebnisse

Für die Vermessung des Zustandsraums wurden die Konfigurationsvariablen (**M1.1**) und die Systemzustände (**M1.2**) analysiert. Da Comoros keine Anwendung ist, sondern eine Middleware, müssen keine Systemzustände persistent gehalten werden. Der Zustandsraum der Middleware definiert sich einzig aus den Konfigurationsvariablen. Aus diesem Grund wird die Anzahl der Systemzustände mit 0 vermessen. Die Konfigurationsvariablen wurden in Abschnitt 6.6 definiert und haben einen Umfang von 26. Die persistente Speicherung der Daten übernimmt der *OSGi Config Admin*, der diese im Dateisystem ablegt. In dafür eigens definierten Testfällen konnte dieser Vorgang für alle 26 Variablen nachgewiesen werden (**M2.1**). Bei einem Neustart des Systems werden zudem die Werte aller 26 Variablen aus dem Dateisystem ausgelesen und ins Laufzeitsystem übertragen (**M3.1**).

Interpretation

Die 26 Konfigurationsvariablen ermöglichen die Anpassung an unterschiedliche Anforderungen. So können verschiedene Arten der Proxy-Generierung gewählt, Kompressionsverfahren aktiviert oder die Integration entfernter Services angestoßen werden. Über die Konfigurationsvariablen von Comoros wird dessen gesamter Zustandsraum definiert. So werden bei einem Neustart von Comoros die geplanten Bindungen zwischen Clients und Services wieder aufgebaut und somit der zuvor geltende Zustand der Middleware wiederhergestellt. Dabei hat es keine Bedeutung, ob die Software regulär beendet wurde oder ob es einen ungeplanten Systemausfall gab. Solange die Variablen persistent gespeichert werden, kann der alte Zustand wieder erreicht werden. Aus diesem Blickwinkel hat der Test der Wiederherstellbarkeit eine enorme Bedeutung für die Funktion der Comoros-Middleware, und alles andere als eine vollständige Wiederherstellung des Zustandes kann

als grobes Fehlverhalten angesehen werden. Der Test für Comoros liefert allerdings ein erfolgreiches Ergebnis.

7.2 Funktionsabdeckung

Innerhalb der Funktionsabdeckung wird analysiert, ob aus Sicht des Anwenders alle an die Software aufgestellten Anforderungen erfüllt wurden. Tabelle 7.2 gibt an, wie dieses Merkmal im Folgenden vermessen wird.

Tabelle 7.2: GQM-Zerlegung des Zielfaktors Evaluierung der Funktionsabdeckung

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Funktionsabdeckung
Objekt	von Comoros
Perspektive	aus Sicht des Nutzers
Frage	**Metrik**
Q1: Wie viele Funktionen wurden geplant?	**M1.1:** Anzahl der geplanten Funktionen
Q2: Wie viele Funktionen wurden umgesetzt?	**M2.1:** Anzahl der umgesetzten Funktionen

Messergebnisse

Die Anzahl der Funktionen (**M1.1**) kann direkt aus den in Kapitel 4 definierten funktionalen Anforderungen abgeleitet werden. Somit ergeben sich folgende essentielle Funktionen für die Comoros-Software:

- OSGi-Client nutzt OSGi-Service
- OSGi-fremder-Client nutzt OSGi-Service
- OSGi-Client nutzt OSGi-fremden-Service
- Verteilte Event-basierte Kommunikation
- Transformationspattern steuern das Marshaling
- Verteilte Unterstützung des Life-Cycle einer Komponente
- OSGi-Client nutzt Funktionen von seriellen Geräten
- Unterstützung einer Stream-basierten Kommunikation
- Unterstützung heterogener Systemstrukturen (Transportprotokolle, Daten)
- Konfiguration des Systems
- Überwachung des Systems

Somit ergeben sich 11 Kernfunktionen, die verschiedene Unterfunktionen enthalten. Für die Kernfunktionen wurden Testfälle identifiziert, mit denen nachgewiesen werden kann, dass die jeweilige Funktion vorhanden und auch ausführbar ist. Hierbei wird der Testfall auf das Normalverhalten und das Ausnahmeverhalten des Prüfgegenstandes ausgerichtet. Ein Test für eine Kernfunktion ist genau dann erfolgreich, wenn auch alle Unterfunktionen erfolgreich getestet wurden. Ein beispielhafter Testfall für die erste

Tabelle 7.3: Testfälle für die Funktion: OSGi-Client nutzt entfernten OSGi-Service

Testfall	*Ja*	*Nein*
Suchanfrage für entfernten Service wurde abgeschickt	X	
Suchanfrage wurde auf der Service-Plattform empfangen	X	
Proxy wurde auf Client-Plattform erstellt	X	
Aufruf des Proxys liefert erwartetes Ergebnis	X	
Aufruf bei Netzwerkfehler bedingt Proxy-Deinstallation	X	

Kernfunktion ist in Tabelle 7.3 beschrieben. Für die Durchführung wurde das von der Comoros-Software mitgelieferte Beispiel *org.ws4d.osgi.example.math* verwendet.

Die Durchführung aller Testfälle ergibt, dass von den 11 Kernfunktionen alle 11 erfolgreich getestet werden konnten (**M1.2**).

Interpretation

Das erfolgreiche Testen aller 11 Kernfunktionen zeigt, dass die Comoros-Software eine vollständige Funktionsabdeckung besitzt. Durch das Aufstellen detaillierter Testfälle konnten zudem im Entwicklungsprozess zwischenzeitlich vorhandenes Fehlverhalten aufgedeckt, und somit die Softwarequalität weiter erhöht werden.

7.3 Sicherheit

Unter dem Merkmal Sicherheit wird hier explizit und ausschließlich die Kommunikationssicherheit angesprochen, die für eine Middleware eine besondere Bedeutung hat. Die Metriken zur Analyse der Sicherheit sind in Tabelle 7.4 aufgeführt.

Tabelle 7.4: GQM-Zerlegung des Zielfaktors Evaluierung der Sicherheit

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Kommunikationssicherheit
Objekt	von Comoros
Perspektive	aus Sicht des Nutzers
Frage	**Metrik**
Q1: Welche Schutzziele der Kommunikationssicherheit werden realisiert?	**M1.1:** Liste der Schutzziele
Q2: Welche Sicherheitsfunktionen werden eingesetzt?	**M2.1:** Liste der Sicherheitsfunktionen

Messergebnisse

Für die Bewertung von Sicherheit in der Informationstechnik hat sich der Standard *Common Criteria for Information Technology Security Evaluation (CC)* [ISO09b] etabliert, der von der ISO unter der Nummer 15408 veröffentlicht wurde.

Grundsätzlich werden nach CC im Zuge der Evaluierung der Sicherheit aus Sicherheitszielen konkrete Sicherheitsanforderungen bestimmt, die der Evaluierungsgegenstand, also

die Comoros-Middleware, durch seine Sicherheitsfunktionen erfüllt. Daraus ergibt sich für Comoros die Tabelle 7.5.

Tabelle 7.5: Analyse der Sicherheit von Comoros nach CC

Sicherheitsziel
Schutz hinsichtlich der Bedrohungen nicht-autorisierte-Preisgabe, Modifizierung und Identitätsdiebstahl.
Sicherheitsanforderungen (M1.1)
SA1: Sicherstellung der Vertraulichkeit
SA2: Sicherstellung der Datenintegrität
SA3: Sicherstellung der Authentifizierung
Sicherheitsfunktionen (M2.1)
SF1: Verschlüsselung (SA1, SA2)
SF2: Zertifikate (SA3)

Interpretation

Für die Realisierung der Sicherheitsfunktionen wird in Comoros auf die JMEDS-Bibliothek zurückgegriffen, die wiederum auf der Java-Bibliothek JSSE basiert. Für eine Bewertung der Stärke der Sicherheitsfunktionen werden die Eigenschaften der JSSE-Bibliothek aufgeschlüsselt.

- Protokolle
  - SSL 3.0
  - TLS 1.0
  - TLS 1.1
  - TLS 1.2 und TLS 1.2 Suite B
- Schlüsselaustausch-Algorithmen
  - RSA
  - DHE-RSA (max. 2048 bit)
  - DHE-DSS (max. 2048 bit)
  - ECDH-ECDSA
  - ECDH-RSA
- Verschlüsselungs-Algorithmus
  - AES GCM
  - AES CBC
  - 3DES EDE CBC
- Daten-Integrität
  - HMAC-MD5
  - HMAC-SHA1
  - HMAC-SHA256/384
  - AEAD

Die JSSE-Bibliothek kann nach diesen Eigenschaften konfiguriert werden und so eine möglichst hohe Stärke der Sicherheitsfunktionen erreicht werden.

Um die Vertrauenswürdigkeit des EVG zu bewerten, sieht die CC mehrere Evaluierungsstufen vor, den so genannten *Evaluation Assurance Level (EAL)*. Dabei ist EAL-1 die niedrigste und EAL-7 die höchste Stufe der Vertrauenswürdigkeit. Für die Evaluierung der Sicherheitsfunktionen innerhalb von Comoros wurde die Stufe EAL-2 umgesetzt. CC beschreibt diese Stufe so, dass Vertrauenswürdigkeit dadurch erlangt wird, dass die Sicherheitsfunktionen unter Verwendung einer funktionalen Spezifikation und einer Schnittstellenspezifikation sowie von Handbüchern und des Entwurfs des EVG auf hoher Ebene analysiert werden, um das Sicherheitsverhalten zu verstehen. Zusätzlich werden die Sicherheitsfunktionen umfangreich getestet.

Die Analyse der Schnittstellen und Dokumentation basiert im Falle von Comoros auf denen der JMEDS-Bibliothek. So konnte ein umfangreiches Verständnis der angebotenen Sicherheitsfunktionen erlangt werden. Die ausgeführten Tests auf Ebene von Comoros wiederum haben belegt, dass die Sicherheitsanforderungen in der Stärke der gewählten Konfiguration umgesetzt wurden.

7.4 Flexibilität

McCall definiert die Flexibilität einer Software mit “der Leichtigkeit, mit der ein System verändert werden kann, um es an neue Anforderungen anzupassen” [McC77]. Diese Anpassung muss sowohl zur Design-Zeit, beispielsweise durch die Anpassung oder Erweiterung der Implementierung, als auch zur Laufzeit, durch eine neue Konfiguration erfolgen können. Diese Anpassung wird von Eden [Ede06] in seiner Abhandlung als Evolutionsschritt definiert, der in Abbildung 7.2 illustriert ist.

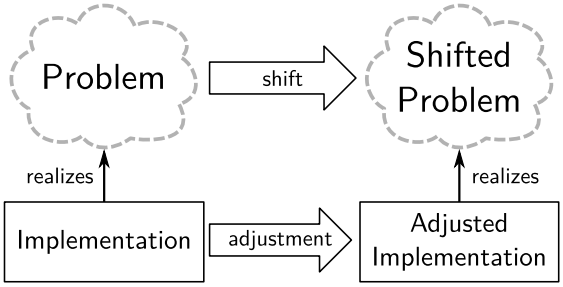


Abbildung 7.2: Evolutionsschritt nach Eden [Ede06]

Die Flexibilität hat somit auch direkten Einfluss auf andere Merkmale, insbesondere auf die Effizienz und die Funktionsabdeckung. Für die Vermessung der Flexibilität existiert in der Literatur kein allgemeingültiges Verfahren. Vielmehr wird das Fehlen von Metriken in diesem Bereich beklagt [Ede06]. Viele der vorgestellten Metriken haben einen sehr theoretischen Ansatz, der nur auf ganz spezielle Problemstellungen angewendet werden kann. Tabelle 7.6 zeigt die GQM-Zerlegung für die Evaluierung der Flexibilität von Comoros und die hier eingesetzten Metriken.

Tabelle 7.6: GQM-Zerlegung des Zielfaktors Evaluierung der Flexibilität

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Flexibilität
Objekt	von Comoros
Perspektive	aus Sicht des Nutzers
Frage	Metrik
Q1: Wie ist der Umfang der Konfigurationsmöglichkeiten?	**M1.1:** Anzahl der Freiheitsgrade
Q2: Inwieweit kann das System zur Design-Zeit angepasst werden?	**M2.1:** Liste der optionalen Komponenten **M2.2:** Abhängigkeitsgraph der Komponenten
Q3: Kann das System auch in Bereichen angewendet werden, für die es nicht entwickelt wurde?	**M3.1:** Liste der wiederverwendbaren Module

Messergebnisse

Die Flexibilität der Comoros-Software hinsichtlich der Konfiguration zur Laufzeit definiert sich im wesentlich über die Anzahl der Freiheitsgrade (**M1.1**). Für Anpassungen zur Design-Zeit werden die optionalen Komponenten der Comoros-Software bestimmt (**M2.1**) und deren Abhängigkeiten untersucht (**M2.2**). Die Liste der wiederverwendbaren Module wird in Verhältnis zu allen Comoros-Modulen gesetzt und ergibt so eine von McCall eingeführte Metrik [McC77], welche die Anwendbarkeit in anderen Bereichen evaluiert (**M3.1**).

Als Freiheitsgrad wird die Zahl der voneinander unabhängigen – und in diesem Sinne "frei wählbaren" – Entscheidungsmöglichkeiten eines Systems bezeichnet. Um die Übersichtlichkeit zu wahren wurden bei der Vermessung der Comoros-Freiheitsgrade das Logging und technologiespezifische Konfigurationen, wie die Parametrisierung des HTTP-Protokolls, nicht berücksichtigt. Diese haben entweder keine Relevanz für den Programmablauf oder sind keine durch Comoros eingeführte Freiheitsgrade. In Comoros existieren folgende Entscheidungsmöglichkeiten:

- Proxy-Bundles vs. Dynamische Proxys
- Konfigurationsgesteuerte Bereitstellung vs. Anfragegesteuerte Bereitstellung
- Generierte Interfaces vs. Manuelle Bereitstellung
- RS-Marshaling vs. Generisches Marshaling
- Wahl des Kommunikationsprotokoll (DPWS vs. CoAP vs. UPnP etc.)
- Abgesicherte Verbindung vs. Nicht abgesicherte Verbindung
- Legacy-Services vs. RS-Services
- EventConverter vs. EventHandler

Insgesamt existieren in Comoros also 8 verschiedene Freiheitsgrade, die durch die unterschiedlichen Konfigurationsvariablen eingestellt werden können.

Abbildung 7.3 zeigt den Abhängigkeitsgraphen der Comoros-Bundles und kennzeichnet gleichzeitig die optionalen Komponenten. Es gibt insgesamt 4 optionale Komponenten, wobei eins der beiden Bundles `marshal` oder `marshal.generic` zwingend im System vorhanden sein muss.

Die Bundles `marshal.api`, `marshal`, `marshal.generic` und `dpws` können auch losgelöst von der Comoros-Middleware durch andere OSGi-basierte Applikationen verwendet werden. Die restlichen 3 Bundles sind an Comoros gebunden. Das ergibt ein Verhältnis von $4/7 = 0{,}57$.

Interpretation

Der Mangel an allgemeingültigen Metriken zur Vermessung der Flexibilität erschwert auch den Vergleich mit anderer Software und so die Einstufung von Comoros. Shen definiert ein System flexibler als ein anderes, falls dieses eine größere Anzahl an Konfigurationen ermöglicht, und die Anpassungen in kürzerer Zeit und mit geringeren Kosten ausgeführt werden können [She06]. Die Anzahl der Konfigurationsmöglichkeiten einer Software ergibt sich allerdings jeweils aus deren Anforderungen und kann so nicht sinnvoll verglichen werden.

Allerdings zeugen die 8 Freiheitsgrade von Comoros und die damit verbundene große Anzahl an Permutationen von einer großen Flexibilität für den Benutzer, der so eine individuelle für die jeweils vorliegenden Bedingungen passende Konfiguration vornehmen kann. Der Einfluss der Freiheitsgrade auf die Effizienz wird an späterer Stelle noch detaillierter untersucht.

Neben der Konfiguration der Software können auch Teile der Software nicht im System installiert werden um so eine minimale Laufzeitumgebung für den eigenen Anwendungsfall zu erschaffen. Diese Entscheidung wird zur Design-Zeit getroffen. Falls z. B. nur eine plattformübergreifende Event-basierte Kommunikation gewünscht wird, so muss das `dsw`-Bundle nicht installiert werden. Auf diese Weise wird die Flexibilität weiter erhöht. Zusätzlich ermöglicht Comoros auch die Implementierung und Integration von neuen Bundles um die Funktionalität zu erhöhen. Es existieren Schnittstellen zur Integration weiterer Kommunikationsprotokolle oder von Transformationspattern für das Marshaling der Daten. Auch auf diese Weise kann auf neue oder veränderte Anforderungen reagiert werden.

Abschließend zeugt das Verhältnis von allgemein anwendbaren Modulen zu der Gesamtanzahl aller Module von 0,57 von einer erhöhten Anwendbarkeit in anderen Umgebungen.

7.5 Strukturierung

Die Strukturierung einer Software ist ein Merkmal aus Sicht des Entwicklers der Comoros-Middleware. Sie beschreibt den Umfang und die Komplexität des Programmcodes, dessen Modularisierung und die Beziehungen zwischen den einzelnen Modulen. Die Strukturierung gibt somit einen Hinweis darauf, wie viel Aufwand ein Entwickler für die Einarbeitung in den Programmcode benötigt. Die GQM-Zerlegung ist in Tabelle 7.7 zu sehen.

Tabelle 7.7: GQM-Zerlegung des Zielfaktors Evaluierung der Strukturierung

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Strukturierung
Objekt	von Comoros
Perspektive	aus Sicht des Entwicklers
Frage	**Metrik**
Q1: Welchen Umfang und welche Komplexität hat das Comoros Projekt?	**M1.1:** Programmcode in *Lines of Code* (LOC) **M1.2:** Umfangsberechnung mit der Halstead-Metrik **M1.2:** Berechnung der Komplexität nach McCabe
Q2: Wie ist der Grad der Modularisierung?	**M2.1:** Anzahl der Bundles **M2.2:** Anzahl der Klassen
Q3: Welche Abhängigkeiten existieren zwischen den Modulen?	**M3.1:** Abhängigkeitsgraph

Messergebnisse

Die Vermessungen des Comoros-Projekts hinsichtlich der aufgestellten Fragestellungen wurden mit Hilfe der Werkzeuge `CodePro Analytix`[1] von Google und des Open-Source-Tools `Metrics`[2], jeweils als Eclipse-Plugins, durchgeführt. Um eine bessere Übersicht zu erlangen, wurden die Bundles des Comoros-Projekts in die Bereiche `Middleware`, `Werkzeuge` und `Beispiele` unterteilt, deren Inhalte in Tabelle 7.8 aufgelistet sind. Nicht berücksichtigt wurden dabei alle Implementierungen für die Evaluierung, individuelle Transformationspattern, die CoAP-Implementierung, die als Erweiterung der JMEDS-Bibliothek realisiert wurde, die Geräteintegration und alle verwendeten externe Bibliotheken. Alle diese Teile sind nicht Bestandteil der Comoros-Middleware.

Q1: Welchen Umfang und welche Komplexität hat das Comoros-Projekt?

Der Umfang des Projekts wird durch die Lines-of-Codes (LOC) (**M1.1**), die Halstead-Metrik (**M1.2**) und die Zyklomatische-Komplexität nach McCabe (**M1.3**) angegeben. Bei der LOC-Berechnung wurden Kommentare jeglicher Art, sowie Leerzeilen nicht mitberechnet. Die Halstead-Metrik (siehe Unterabschnitt 2.1.8) besteht aus der Größe des Vokabulars `n`, der Länge der Implementierung `N`, dem Programmvolumen `V`, dem Schwierigkeitsgrad `D` und dem Aufwand zum Verständnis des Programmcodes `E`. Die Ergebnisse der Messungen finden sich in Tabelle 7.9.

1 Ed.: Google, URL: https://developers.google.com/java-dev-tools/codepro/doc/, Abruf: 11.03.2015

2 http://metrics.sourceforge.net/, Abruf: 11.03.2015

Tabelle 7.8: Einteilung der Comoros-Bundles

Gruppe	*Bundles*	
Middleware	org.ws4d.osgi.management	org.ws4d.osgi.marshal
	org.ws4d.osgi.dpws	org.ws4d.osgi.marshal.api
	org.ws4d.osgi.dsw	org.ws4d.osgi.marshal.generic
	org.ws4d.osgi.ec	
Werkzeuge	org.ws4d.osgi.configAgent	org.ws4d.osgi.mappingTool
Beispiele	org.ws4d.osgi.streamingAPI	org.ws4d.osgi.javaPersistenceAPI
	org.ws4d.osgi.streamingService	org.ws4d.osgi.javaPersistenceService
	org.ws4d.osgi.streamingClient	org.ws4d.osgi.javaPersistenceClient
	org.ws4d.osgi.xmlAPI	org.ws4d.osgi.MathAPI
	org.ws4d.osgi.xmlService	org.ws4d.osgi.MathService
	org.ws4d.osgi.xmlClient	org.ws4d.osgi.MathClient
	org.ws4d.osgi.marshallingPatterns	

Q2: Wie ist der Grad der Modularisierung?

Der Grad der Modularisierung gibt an, inwieweit das Programm in einzelne Teile aufgeteilt ist. Ein hoher Grad der Modularisierung minimiert den Aufwand zum Verständnis des Programmcodes. Gemessen werden die Anzahl der Bundles (**M2.1**) und die Anzahl der Klassen (**M2.2**), die Ergebnisse sind in Tabelle 7.10 festgehalten.

Q3: Welche Abhängigkeiten existieren zwischen den Modulen?

Für die Berechnung der Abhängigkeiten zwischen den einzelnen Software-Modulen werden lediglich die Bundles aus dem Bereich der Middleware betrachtet. Die Werkzeuge sind generell unabhängig von den restlichen Teilen und die Beispiele sind kleine Anwendungen und fallen somit auch aus dem Fokus des Interesses. Abbildung 7.3 zeigt den Abhängigkeitsgraphen der Comoros-Software.

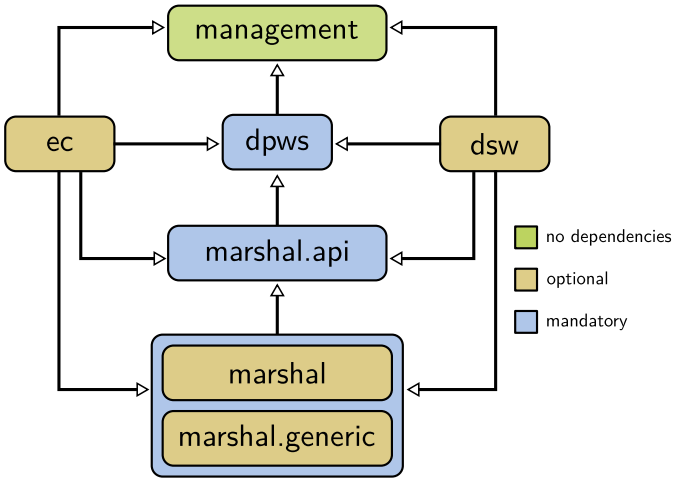


Abbildung 7.3: Abhängigkeiten zwischen den Bundles der Comoros-Middleware

Tabelle 7.9: Ergebnisse der Umfangs- und Komplexitäts-Berechnung

Bundle	*LOC*	*CC*	*n*	*N*	*V*	*D*	*E*
org.ws4d.osgi.management	1990	2,45	650	6748	63055	133,59	8423737,39
org.ws4d.osgi.dpws	1035	3,35	443	2872	25245	58,1	1467024,25
org.ws4d.osgi.dsw	13085	2,89	2770	49382	564716	228,02	128769246,53
org.ws4d.osgi.ec	1201	3,15	495	3520	31508	59,74	1882483,68
org.ws4d.osgi.marshal	2415	3,52	713	8898	84333	122,17	10303657,26
org.ws4d.osgi.marshal.api	271	1,17	166	804	5929	30,54	181125,59
org.ws4d.osgi.marshal.generic	3173	3,19	842	10368	100752	119,89	12079699,64
org.ws4d.osgi.configAgent	1128	2,37	447	3305	29097	74,04	2154476,34
org.ws4d.osgi.mappingTool	8561	1,72	2753	28843	329582	126,04	41540654,28
org.ws4d.osgi.streamingAPI	17	1	16	33	88	2,07	182,76
org.ws4d.osgi.streamingService	76	1,42	71	173	1063	11,27	11995,54
org.ws4d.osgi.streamingClient	54	1,33	56	119	691	8,42	5822,67
org.ws4d.osgi.xmlAPI	59	1	48	144	804	8	6433,87
org.ws4d.osgi.xmlService	90	1	88	305	1970	19,86	39136,29
org.ws4d.osgi.xmlClient	57	1	94	247	1619	7,56	12244,72
org.ws4d.osgi.javaPersistenceAPI	44	1	34	98	498	8,03	4006,37
org.ws4d.osgi.javaPersistenceService	41	1,72	45	96	527	5,76	3041,64
org.ws4d.osgi.javaPersistenceClient	51	1	53	142	813	8,29	6745,62
org.ws4d.osgi.marshallingPatterns	352	1,56	167	852	6290	25,74	161970,66
org.ws4d.osgi.MathAPI	35	1	9	9	28	0	0
org.ws4d.osgi.MathService	206	1,14	180	816	6113	24,28	148477,58
org.ws4d.osgi.MathClient	407	1,36	289	1605	13120	40,25	528122,52
Bereich	***LOC***	***CC***	***n***	***N***	***V***	***D***	***E***
Middleware	23170	2,9	4571	82592	1004177	262,68	263778007,84
Werkzeuge	9689	1,79	3065	32148	372327	125,49	46725206,12
Beispiele	1489	1,29	625	4628	42983	62,99	2707534,87
Insgesamt	34348	2,33	7382	119368	1533854	234,22	359261057,01

Interpretation

Der Umfang der Comoros-Software wurde in einem ersten Schritt über die LOC vermessen. Eine Visualisierung der wichtigsten Messergebnisse ist in Abbildung 7.4 zu sehen.

Die der Middleware zugehörigen Bundles nehmen mit 23170 Zeilen Code den mit Abstand größten Part der gesamten Comoros-Software ein. Innerhalb der Middleware ist die *Distribution Software* mit über 13000 Zeilen die wichtigste Komponente. Hier wird der OSGi-Remote-Services-Standard implementiert. Ebenfalls gut zu erkennen ist, dass das Marshaling in Comoros eine wichtige Rolle spielt, insgesamt wurden hier 5859 Zeilen Code implementiert.

Die für Comoros entwickelten Werkzeuge, die den Komfort für den Entwickler erhöhen, nehmen mit insgesamt 9689 Zeilen Code ebenfalls eine wichtige Rolle ein. Hier ist gerade

Tabelle 7.10: Ergebnisse der Berechnung der Modularisierung

Bundle	*#Klassen*	*#Bundles*	*Bundle*	*#Klassen*	*#Bundles*
management	18	1	marshal	42	1
dpws	7	1	marshal.api	6	1
dsw	65	1	marshal.generic	38	1
ec	16	1			
configAgent	18	1	mappingTool	77	1
streamingAPI	1	1	javaPersistenceAPI	1	1
streamingService	3	1	javaPersistenceService	2	1
streamingClient	2	1	javaPersistenceClient	2	1
xmlAPI	1	1	MathAPI	2	1
xmlService	2	1	MathService	3	1
xmlClient	2	1	MathClient	12	1
marshallingPatterns	11	1			
Bereich	***#Klassen***	***#Bundles***			
Middleware	192	7			
Werkzeuge	95	3			
Beispiele	44	13			
Insgesamt	331	22			

das Comoros.MappingTool eine umfangreiche Entwicklung. Die Beispiele wurden bewusst klein und verständlich gehalten und sind somit mit gerade einmal 1489 Zeilen Code sehr übersichtlich.

Die gesamte Comoros-Umgebung kann mit seinen insgesamt 34348 Zeilen Programmcode durchaus als umfangreiches Softwareprodukt angesehen werden. McConnell gibt an, dass ein ein einzelner Entwickler pro Tag etwa 10–50 Zeilen Code erzeugen kann [McC04],

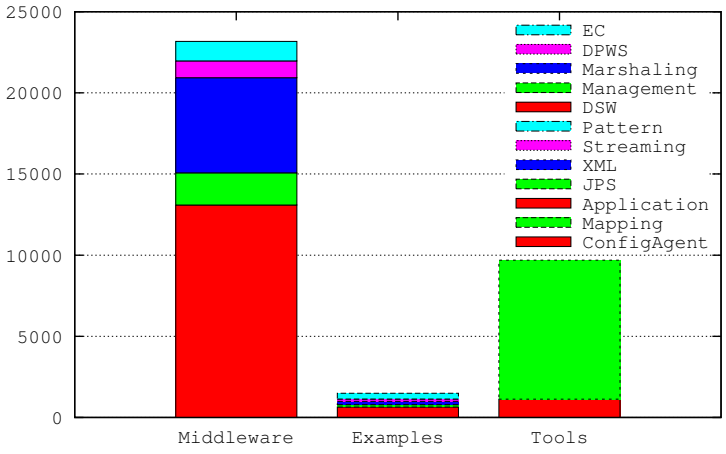


Abbildung 7.4: Comoros Lines of Code Verhältnisse

so dass sich bei einer Laufzeit von 5 Jahren 10000 bis 50000 Zeilen Code summieren. Comoros liegt innerhalb dieser Spanne.

Die zyklomatische Komplexität und die Halstead Metrik bewerten nicht den Umfang der Software, sondern deren Qualität und Wartbarkeit. Allerdings geben diese Werte keine exakten Auskünfte über diese beiden Merkmale, bieten aber einen guten Anhaltspunkt. Dennoch können die automatisch gewonnenen Werte keine manuelle Untersuchung ersetzen.

Mit Hilfe der zyklomatischen Komplexität kann ein Eindruck von der Komplexität der Comoros-Software und auch indirekt einen Eindruck über deren Wartbarkeit gewonnen werden. Um die gewonnenen Werte einordnen zu können, müssen diese mit Referenzwerten verglichen werden. Die Carnegie Mellon University definiert insgesamt vier Wertebereiche [Bra97]:

- **1–10:** Die Methode gilt als simpel und unproblematisch zu überschauen und zu testen.
- **11–20:** Die Methode beinhaltet deutlich komplexeren Code, der zwar weiterhin verständlich, aber die Testbarkeit deutlich erschwert sein kann.
- **20–50:** Die Methode kann eine unüberschaubare Anzahl von Code-Pfaden enthalten und lässt sich nur noch schwer verstehen und testen.
- **50+:** Die Methode gilt als unwartbar.

Die durchschnittliche zyklomatische Komplexität der Comoros-Software beträgt 2,33, wobei die Middleware-Bundles mit 2,9 deutlich komplexer als die einfach gehaltenen Beispiel-Bundles mit 1,29 sind. Da diese Metrik aber auf Ebene der Methoden angewendet wird, ist der durchschnittliche Wert über alle Methoden zwar ein Hinweis auf die Komplexität, dennoch müssen einzelne Ausreißer betrachtet werden. Tabelle 7.11 zeigt die Anzahl der Methoden in den aufgestellten Kategorien.

Tabelle 7.11: Ausreisser zyklomatische Komplexität

Bundle	*10–20*	*21–50*	*>50*	*Bundle*	*10–20*	*21–50*	*>50*
management	6	0	0	marshal	5	1	1
dpws	1	2	0	marshal.api	0	0	0
dsw	19	5	0	marshal.generic	5	1	0
ec	1	2	0				
configAgent	2	0	0	mappingTool	6	1	0
Insgesamt	45	12	1				

In der schlechtesten Kategorie der unwartbaren Methoden befindet sich genau eine Methode im Marshaling-Bundle. Diese Methode unterscheidet eingehende Datentypen und besteht aus einer großen `switch-case`-Anweisung. Diese Art von Codeblock verursacht zwar eine große zyklomatische Komplexität, wird von Entwicklern dagegen als nicht kompliziert empfunden. Weitere 12 Methoden sind der Kategorie der schwer verständlichen Methoden zugeordnet, und 45 in der Kategorie der verständlichen aber erschwert testbaren

Methoden. Insgesamt existieren 1847 Methoden, Abbildung 7.5 zeigt den Anteil aller Kategorien.

Gerade einmal 0,05% aller Methoden sind der vierten Kategorie zugeordnet, 0,6% der Dritten und 2,4% der zweiten Kategorie. Somit sind 96,95% aller Methoden in der ersten Kategorie und somit einfach zu verstehen. Ein Blick auf die Ausreißer-Methoden zeigt, dass es sich zumeist um Methoden handelt, die String-Ketten analysieren und auseinandernehmen. An diesen Stellen lässt sich die erhöhte Komplexität nicht vermeiden.

Abb. 7.5: Komplexität der Comoros-Methoden

Die zuletzt ermittelte Halstead-Metrik kombiniert Umfang- und Komplexitätsmessungen. Das Volumen `V` eines Programms beschreibt den Umfang, während die Maße Schwierigkeitsgrad `D` und Aufwand `E` die Komplexität vermessen. Für das Volumen gilt als Richtwert, dass eine Funktion ein Volumen von 20–1000 umfassen darf, eine Klasse zwischen 100 und 8000 [Lam07]. Bei einer Gesamtanzahl von 2123 Methoden und 331 Klassen ergibt sich ein durchschnittlicher Wert von 722 pro Methode und 4634 pro Klasse, Werte, die somit innerhalb der Referenzwerte liegen.

Für den Schwierigkeitsgrad liegen keine Referenzwerte vor. Die Analyse der Messergebnisse zeigt, dass die Distribution Software den höchsten Schwierigkeitsgrad hat, was sich auch mit den Messergebnisse der zyklomatischen Komplexität deckt. Die Beispiele haben allesamt einen sehr geringen Schwierigkeitsgrad und Aufwand, so dass diese dem Zweck der einfachen Einführung in die Software entsprechen.

Halstead definiert weiterhin noch die Anzahl der Programmierfehler. Diesem Wert liegt die in Experimenten von Halstead gewonnene Erkenntnis zugrunde, dass im Durchschnitt nach 3000 elementaren Operationen des Gehirns ein Programmierfehler auftritt. Somit kann die Fehleranzahl über die Formel $Beta = V/3000$ berechnet werden. Die Comoros-Software enthält nach dieser Berechnung somit 511 Fehler.

Die Modularisierung von Comoros wird anhand der Anzahl von Bundles und Klassen definiert. Insgesamt umfasst das Softwareprojekt 22 Bundles und 331 Klassen. Wie die LOC-Vermessung bereits gezeigt hat, schwankt die Größe der Bundles stark, von 13085 Zeilen für die Distribution Software bis gerade einmal 17 Zeilen für die API des Streaming-Beispiels. Ein Durchschnittswert ist hier also nicht von Nutzen. Für die Klassen gilt diese Einschränkung aber nicht. Es ergeben sich im Mittel insgesamt 107 Zeilen pro Klasse und somit ein ausreichender Modularisierungsgrad.

Der in Abbildung 7.3 aufgezeigte Abhängigkeitsgraph der Bundles wird mit Hilfe der *FanIn* und *FanOut* Metriken analysiert. Diese Werte beschreiben jeweils die eingehenden und ausgehenden Abhängigkeiten jedes Knotens. In Comoros wird maximal der Wert 4 erreicht. Für das DPWS-Bundle und das Marshaling-API-Bundle ergibt sich jeweils ein FanIn von 3 und ein FanOut von 1. Dieser geringe Wert belegt die verständliche und wenig komplexe Struktur von Comoros.

7.6 Konfigurierbarkeit

Die Konfigurierbarkeit steht in Bezug zu der Flexibilität der Software, stellt allerdings den Blickwinkel des Entwicklers dar. Dennoch existieren Überschneidungen zwischen diesen beiden Merkmalen. Die aus Sicht des Entwicklers abgeleiteten Metriken sind in Tabelle 7.12 aufgelistet.

Tabelle 7.12: GQM-Zerlegung des Zielfaktors Evaluierung der Konfigurierbarkeit

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Konfigurierbarkeit
Objekt	von Comoros
Perspektive	aus Sicht des Entwicklers
Frage	**Metrik**
Q1: In welchen Umfang kann Comoros konfiguriert werden?	**M1.1:** Anzahl der Konfigurationsvariablen
Q2: Über wie viele unterschiedliche Technologien kann die Konfiguration übermittelt werden?	**M2.1:** Anzahl der technologiespezifischen Schnittstellen
Q3: Sind die Konfigurationsmöglichkeiten dokumentiert?	**M3.1:** Dokumentation [Ja/Nein]

Messergebnisse

Die Anzahl der Konfigurationsvariablen (**M1.1**) wurde im Zuge der Vermessung der Wiederherstellbarkeit bereits mit 26 vermessen. Der Zugriff auf die Konfigurationsvariablen kann sowohl über OSGi, als auch über jedes unterstützte Kommunikationsprotokoll, wie das DPWS, erfolgen. Somit ergibt sich für die Anzahl der technologiespezifischen Schnittstellen ein Wert von $2 + n$, wobei 2 für die durch Comoros initial ausgelieferten Technologien steht und n für die durch den Entwickler zusätzlich hinzugefügten Technologien. Die Frage nach der Dokumentation der Konfigurationsvariablen (**M3.1**) kann mit `Ja` beantwortet werden.

Interpretation

Die innerhalb der Flexibilität analysierten Freiheitsgrade von Comoros werden auf insgesamt 26 Konfigurationsvariablen abgebildet. Diese können vom Entwickler durch die umfangreiche Dokumentation leicht verstanden werden, die eine semantische Beschreibung, den Datentyp und den Default-Wert enthält. Weitere Konfigurationsvariablen können nach diesem Vorbild einfach ins System eingepflegt werden. Der Zugriff auf die Konfigurationsvariablen kann über unterschiedliche Technologien erfolgen, so dass eine Integration von Comoros in umfassende Management-Systeme vereinfacht wird.

7.7 Effizienz

In diesem Abschnitt wird das Verhalten der Comoros-Software, in Bezug auf die Effizienz und Leistungsfähigkeit, in verschiedenen Klassen von Hardwaresystemen und unter Berücksichtigung relevanter Aspekte bestimmt. Diese Aspekte betreffen die unterschiedli-

chen Konfigurationen und Ausprägungen der Software, die relevante Auswirkungen auf die Effizienz haben. Im folgenden werden zunächst der Versuchsaufbau beschrieben und anschließend die Kernarchitektur und die erweiterte Architektur vermessen.

7.7.1 Versuchsaufbau

Der Versuchsaufbau beschreibt das generelle Evaluierungsumfeld, das aus der verwendeten Hardware, der Netzwerktopologie, der Evaluierungssoftware, dem Messverfahren und der Datenerhebung besteht. Die hier aufgegriffenen Aspekte sind für alle Messungen gültig, also sowohl für die Vermessung der Kernarchitektur, als auch für die Vermessung der erweiterten Architektur.

Hardware

Für die Vermessung der Effizienz und Leistungsfähigkeit des Comoros-Frameworks wird der Einsatz auf unterschiedlichen Zielsystemen analysiert. Dazu werden die Hardwareklassen `Personal Computer`, `Mobile Systeme` und `Eingebettete Systeme` betrachtet. Innerhalb dieser Klassen wurden die in Tabelle 7.13 aufgeführten Systeme ausgewählt.

Tabelle 7.13: Verwendete Hardwareklassen und deren konkrete Systeme

HW-Klasse	*Bezeichnung*	*Hersteller, Modell*	*CPU*	*RAM*	*BS*
PC	PC	–	Intel i7 Quad-Core, 2,8 GHz	8 GB	openSuse 12.3
PC	Laptop	Lenovo ThinkPad X61	Intel Core2 Duo 2 GHz	2 GB	opensuse 11.3
Mobiles System	SGT	Samsung, Galaxy Tab P7500	ARM Cortex-A9 Dual-Core, 1 GHz	1 GB	Android 3.2
Eingebettetes System	BBB	BeagleBone, Black BB-BBLK-000	TI Siatra AM335x ARM Cortex, 1 GHz	512 MB	Linux Angström
Eingebettetes System	RP	Raspberry Pi, B	ARM1176JZF-S core, 700 MHz	512 MB	Raspian Linux

Als Zielsysteme für die Klasse der PCs wurde ein Standard-Desktop-PC und ein Laptop mit den Daten aus Tabelle 7.13 gewählt. Zwar befinden sich beide Systeme in der selben Hardwareklasse, dennoch unterscheiden sie sich deutlich in ihrer Leistungsfähigkeit. Als mobiles System wurde ein Android-basiertes System gewählt, da diese derzeit den Markt anführen und diese Position laut einer Gartner Studie[1] auch in Zukunft einnehmen. In der Klasse der eingebetteten Systeme wurden, analog zur Klasse der PCs, zwei Systeme gewählt, die untereinander stark unterschiedliche Leistungsmerkmale besitzen. Alle Systeme erfüllen die Voraussetzung ein OSGi-System betreiben zu können und sind damit für den Einsatz von Comoros geeignet.

Versuchs-Topologie

Die ausgewählten Systeme können innerhalb eines durch Comoros aufgespanntes verteiltes Dienste- und Gerätesystem in zwei Rollen eingesetzt werden, in der Rolle des Clients und in

1 Ed.: Gartner, URL: `http://www.gartner.com/newsroom/id/2408515`, Abruf: 21.05.2015

der Rolle des Servers. Eine vollständige und umfassende Vermessung der Leistungsfähigkeit von Comoros auf allen Systemen erfordert den Einsatz jedes Systems in jeder Rolle in allen möglichen Kombinationen. Um den Umfang der erhobenen Daten in einem beherrschbaren Rahmen zu belassen und unnütze Datengräber – Daten ohne Mehrwert hinsichtlich zu untersuchender Aspekte – zu vermeiden, wurden drei der vier Systeme in der Rolle des Servers eingesetzt, und nur ein System in der Rolle des Clients. In den Experimenten werden auf allen Knoten jeweils die selben Operationen ausgeführt, so dass die Unterschiede der Hardwareklassen in Bezug auf die untersuchten Aspekte deutlich werden. Abbildung 7.6 zeigt die verwendete Topologie.

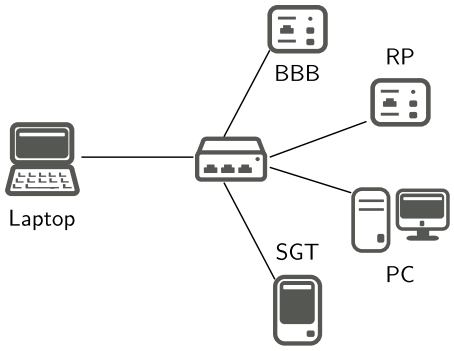


Abbildung 7.6: Netzwerktopologie für Evaluierungsexperimente

Der Laptop dient in der ausgewählten Topologie als Client, während die anderen Systeme in der Rolle des Servers agieren. Bis auf das SGT, das über eine WLAN-Verbindung ins System integriert ist, verwenden alle Knoten eine Ethernet-Verbindung.

Messzenario

Das Messzenario umfasst eine Client- und eine Server-Anwendung. Zum Start der Evaluierungssoftware kann auf jeder Seite zwischen dem dynamischen Ex- und Importieren von Services nach der RS-Spezifikation und dem statischem Ex- und Importieren anhand von Filterlisten für die Unterstützung von Legacy-Systemen gewählt werden.

Die Client-Evaluierungsanwendung generiert Nutzdaten in unterschiedlichen Datenstrukturen und übermittelt diese an die Service-Anwendung. Die Server-Anwendung empfängt die Daten, summiert die einzelnen Elemente auf und sendet das Ergebnis zurück zum Client. Dabei kann zwischen einer synchronen Kommunikation mittels eines Service-Aufrufs und einer asynchronen Event-basierten Kommunikation gewählt werden.

Innerhalb des Messzenarios werden zwei Arten von Messungen durchgeführt: `Analyse-Messungen` und `Last-Messungen`. Die Analyse-Messungen betrachten die einzelnen Messobjekte separat und verzichten auf eine Einordnung in den Gesamtprozess, der Weg von einer Anfrage an den Server bis zur Antwort. Auf diese Weise können die einzelnen Aspekte präzise analysiert werden und die Unterschiede zwischen den einzelnen Konfigurationen und Hardwareklassen aufgedeckt werden.

Die Last-Messungen betrachten den Gesamtprozess für verschiedene Anzahlen gleichzeitiger Anfragen an den Server. Durch die Analyse des Ressourcen-Verbrauchs kann die Skalierung der unterschiedlichen Konfigurationen und Hardwareklassen mit einer

steigenden Anzahl von Anfragen bestimmt werden.

Datenerhebung

Für die Durchführung von Benchmark-Messungen ist es nicht ausreichend lediglich eine einzelne Messung für jeden Versuch auszuführen. Eine einzelne Messung kann z. B. durch das JVM-Verhalten, durch Ressourcen-Reservierungen, Hintergrundprozesse, System-Interrupts oder durch die Garbage-Collection stark beeinflusst werden. Aus diesem Grund werden alle Messungen mit einer ausreichenden Anzahl an Wiederholungen durchgeführt, z. B. 1000 für Messungen bezüglich der Aufrufzeiten. Bei der Durchführung der Wiederholungen ist es wichtig, dass diese voneinander unabhängig durchgeführt werden, damit die Ergebnisse nicht korreliert sind. Abhängige Messungen können z. B. entstehen, wenn in einem ersten Durchlauf Datenstrukturen initial gefüllt werden, auf deren Inhalt im zweiten Durchlauf zugegriffen wird. Diese Art von Korrelation muss vermieden und eine Stichprobe mit voneinander unabhängigen Werten erzeugt werden.

Liegt eine Stichprobe mit n unkorrelierten Werten vor, so kann über diese die interessierende Statistik, also der Mittelwert, gebildet werden. Dieser Mittelwert ist in Bezug auf die Grundgesamtheit, aus der die Stichproben gezogen wurden allerdings nur eine Schätzung. Eine neue Stichprobe aus der Grundgesamtheit kann zu einem anderen Mittelwert führen. Mit Hilfe von Konfidenzintervallen kann nun die Zuverlässigkeit der berechneten Mittelwerte bestimmt werden. Über eine definierte Wahrscheinlichkeit P wird für jede Stichprobe ein Bereich von Schätzungen ermittelt. $P\%$ aller so errechneten Konfidenzintervalle beinhalten nun den wahren Mittelwert der Grundgesamtheit. Somit wird durch die Größe des Konfidenzintervalls direkt die Zuverlässigkeit indiziert. Je enger das Intervall ist, umso genauer ist der geschätzte Mittelwert. Üblicherweise wird in der Literatur P mit 95% gewählt, dieser Wert wird auch für die hier durchgeführten Analysen verwendet.

Konfidenzintervalle können über verschiedene Verfahren berechnet werden. Für Performanz-Messungen wird in der Literatur das Bootstrap-Verfahren empfohlen, in dem die Schätzung von Parametern in einem durch Efron [Efr93] entwickeltem Resampling-Verfahren erfolgt. Ein großer Vorteil des Bootstrap-Verfahrens ist, dass kein spezielles Verteilungsmodell (z. B. die Normalverteilung) für die Zufallsvariablen vorliegen muss. Weiterhin müssen die auch nicht sehr groß sein – lediglich $n > 20$ – um Folgerungen aus dem zentralen Grenzwertsatz anwenden zu können. Dieser Punkt ist gerade mit Blick auf kompliziert durchzuführende Messungen von entscheidendem Vorteil.

Die für Performanz-Tests wichtige Fragestellung, ob zwischen der Ausführungszeit eines `Task A` und der eines `Task B` statistisch signifikante Unterschiede bestehen, kann mit Hilfe von Konfidenzintervallen nicht vollständig beantwortet werden. Lediglich bei sich überlappenden Intervallen kann festgehalten werden, dass diese Signifikanz nicht besteht. Um eine verlässliche Aussage für einen signifikanten Unterschied treffen zu können wird der *t-Test* [Gos08] durchgeführt.

7.7.2 Vermessung der Kernarchitektur

Comoros unterscheidet abstrakt zwischen der Kernarchitektur, die den RS-Standard auf Basis des DPWS realisiert und der erweiterten Architektur, die weitergehende Anforde-

rungen implementiert. Soll die Comoros-Software strikt den RS-Standard umsetzten, so wird dies durch die Anwendung einer Basiskonfiguration erreicht. In diesem Abschnitt werden zunächst Messungen auf Basis dieser Konfiguration durchgeführt und die daraus entstehenden Ergebnisse als Vergleich für Messungen mit anderen Konfigurationen verwendet.

Die Basiskonfiguration verwendet eine anfragengesteuerte Bereitstellung von Services, dynamische Proxys, das RS-Marshaling, keine Event-basierte Kommunikation und SOAP-over-HTTP als Transportbindung.

Messziele

Das grundsätzliche Messziel bei der Evaluierung der Kernarchitektur ist die Analyse der Effizienz und Leistungsfähigkeit von Comoros im Rahmen seiner Basiskonfiguration und auf unterschiedlichen Rechnerklassen. Die daraus abgeleitete GQM-Zerlegung ist in Tabelle 7.14 ausgeführt.

Tabelle 7.14: GQM-Zerlegung des Zielfaktors Evaluierung der Effizienz und Leistungsfähigkeit

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Effizienz und Leistungsfähigkeit
Objekt	der Comoros-Kernarchitektur
Perspektive	aus Sicht des Nutzers
Frage	**Metrik**
Q1: Wie leistungsfähig ist das Daten-Marshaling?	**M1.1:** Zeit der (De-)Serialisierung auf dem Client
	M1.2: Zeit der (De-)Serialisierung auf dem Server
	M1.3: Größe der erzeugten XML-Dokumente
Q2: Wie effizient und leistungsfähig ist der entfernte Service-Aufruf?	**M2.1:** Zeit Aufruf im Client
	M2.2: Zeit Aufruf im Service
	M2.3: Zeit Round-Trip-Time
	M2.4: Größe TCP-Nachricht
	M2.5: Auslastung CPU
Q3: Welche Auswirkungen haben unterschiedlich strukturierte Daten?	**M3.1:** Zeit Aufruf im Client
	M3.2: Zeit Serialisierung im Client
	M3.3: Zeit Deserialisierung im Service

Das grundsätzliche Ziel einer Middleware ist der Austausch von Applikations-Daten zwischen Anwendungen auf verschiedenen Knoten eines Netzwerks, so dass Untersuchungen in diesem Bereich von besonderer Bedeutung sind (**Q1**). In diesem Zusammenhang ist die Strukturierung von Nutzdaten ein wesentlicher Aspekt der Effizienz-Analysen. Gleiche Nutzdaten können durch unterschiedliche Datenstrukturen repräsentiert werden, die in

einem verteilten Umfeld mit unterschiedlicher Performanz verwendet werden. Für die Untersuchungen dieser Fragestellung werden Messungen mit den folgenden Datenstrukturen vorgenommen:

- Ein-, Zwei-, Drei-, Vierdimensionale Arrays
- ArrayList
- HashMap
- HashSet

Die Datenstrukturen wurden in den einzelnen Experimenten jeweils mit 24, 96, 384 und 1536 Elementen befüllt. Neben der verwendeten Datenstruktur hat auch die Strukturierung der Daten innerhalb eines Datentyps Auswirkungen auf die Messergebnisse (**Q3**). Diese Auswirkungen wurden anhand eines vierdimensionalen-Arrays und der folgenden Strukturierungen vermessen:

- 1536, 1, 1, 1
- 1, 1536, 1, 1
- 1, 1, 1536, 1
- 1, 1, 1, 1536

Das Handling der Daten ist ein wesentlicher Teil der Effizienzanalysen für einen entfernten Service-Aufruf. Alle weiteren Aspekte, vor allem Last-Messungen und Round-Trip-Zeiten werden separat behandelt (**Q2**). Alle Messungen werden in der definierten Topologie mit allen Knoten im Netzwerk durchgeführt.

Messobjekte

Aufgrund der definierten Messziele wurden die Messobjekte abgeleitet. Dazu wurde der Pfad vom Aufruf des entfernten Service durch den lokalen OSGi-Client durchlaufen und hinsichtlich der aufgestellten Metriken untersucht. Es ergeben sich zwei unterschiedliche Objekttypen, Zeit-basierte und Größen-basierte.

Der übergeordnete, Zeit-basierte Messpunkt des entfernten Methodenaufruf (*C_Total*) unterteilt sich auf der Client-Seite in folgende Messobjekte:

- Serialisierung der Eingabedaten (*C_Serialize*)
- Methodenaufruf entfernt (*C_Invoke*)
- Deserialisierung der Rückgabedaten (*C_Deserialize*)

Auf der Service Seite ergeben sich folgende Objekte:

- Deserialisierung der Eingabedaten (*S_Deserialize*)
- Methodenaufruf lokal (*S_Invoke*)
- Serialisierung der Rückgabedaten (*S_Serialize*)

Der Zusammenhang der Objekte ist in Abbildung 7.7 illustriert. Jede dieser Messobjekte ist unabhängig von der Hardware und Konfiguration, so dass die Auswirkungen von

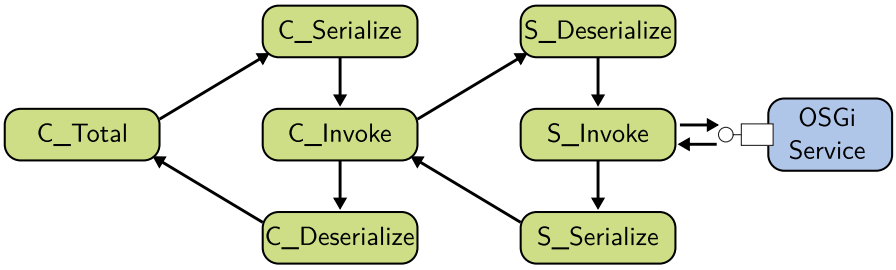


Abbildung 7.7: Abgeleitete Messobjekte aus den Messzielen und dem Ausführungspfad eines entfernten Serviceaufrufs

Änderungen an diesen Parametern auf die Performanz direkt ersichtlich sind und analysiert werden können.

Die Größen-basierten Messobjekte umfassen die Größe der erstellten XML-Dokumente (*C_XML*, *S_XML*) und die Größe der TCP-Segmente (*N_TCP*). Im Zuge der Last-Messungen wird zusätzlich noch die CPU-Auslastung gemessen.

Messergebnisse

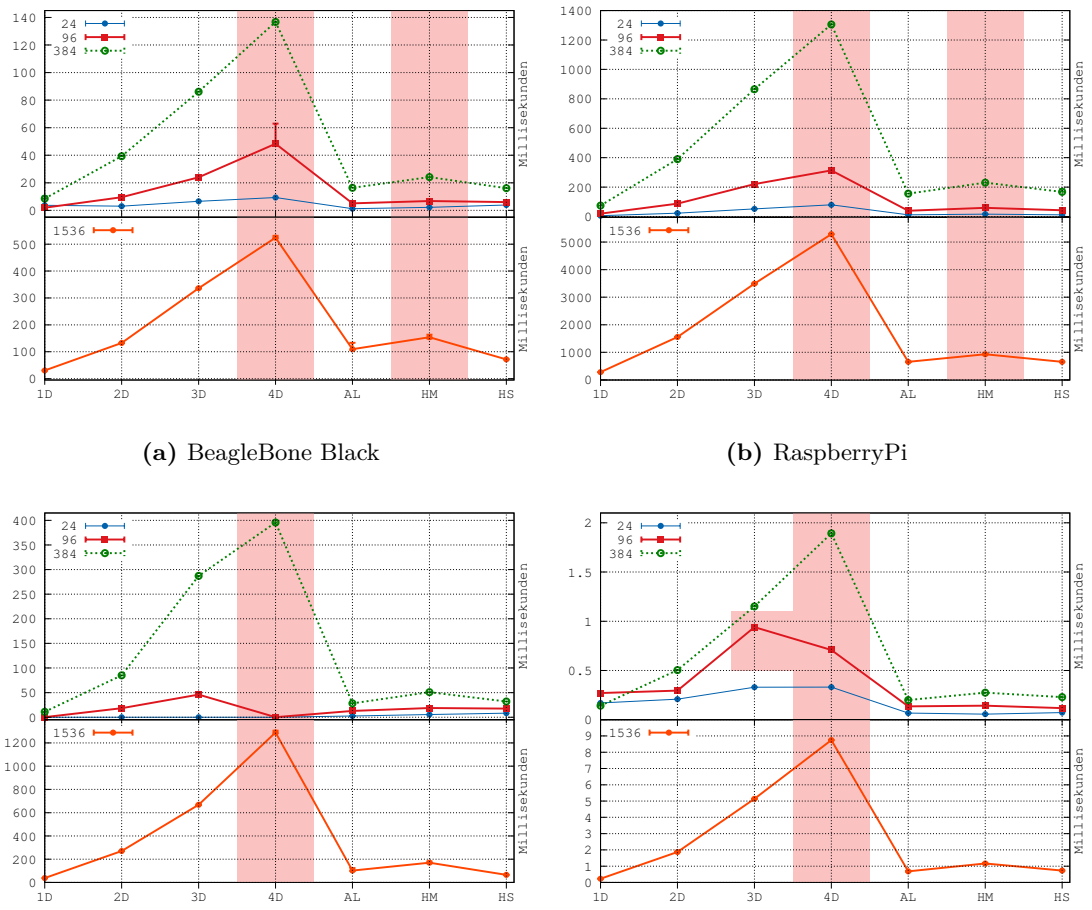
Die Strukturierung der Messergebnisse folgt den abgeleiteten Messobjekten. Zu Beginn werden die Ergebnisse der Server-Seite und anschließend die der Client-Seite präsentiert. Wie beschrieben sind die Messungen in Analyse- und Last-Messungen unterteilt, so dass die Messergebnisse entsprechend vorgestellt werden. Die Interpretation erfolgt anschließend im nachfolgenden Abschnitt.

S_Deserialize Das Messobjekt analysiert die Zeit der Deserialisierung der verschiedenen XML-Datenstrukturen in entsprechende Java-Objekte. Die Ergebnisse sind in Abbildung 7.8 dargestellt.

S_Invoke, S_Serialize Die gemessenen Zeiten des *S_Invoke* Messpunkts, innerhalb dessen der lokale OSGi-Service aufgerufen wird, betragen auf den jeweiligen Zielsystemen um die 0,001% des gesamten Prozesses. Ebenso verhält sich der Messpunkt *S_Serialize*, in dem eine einfache Integer-Zahl in ein XML-Dokument serialisiert wird. Auch hier ergeben sich Zeiten um 0,001%–0,08% des gesamten Prozesses. Somit haben diese beiden Messpunkte keine bedeutsame Auswirkungen auf die Gesamtzeit und die Ergebnisse werden dementsprechend nicht präsentiert.

C_Deserialize Analog zu *S_Invoke* und *S_Serialize* auf der Server-Seite nimmt die Zeit des Messpunkts *C_Deserialize* auf der Client -Seite nur einen sehr geringen Anteil an der Gesamtzeit ein. Hier sind es sogar lediglich um 0,0005%–0,002. Folglich werden auch hier keine Messergebnisse präsentiert.

C_Serialize Die Serialisierung der Eingabedaten in eine XML-Dokument findet auf der Client-Plattform statt und wird nicht von den Systemen auf der Server-Seite beeinflusst.



(a) BeagleBone Black

(b) RaspberryPi

(c) Samsung Galaxy Tab

(d) Personal Computer

Abbildung 7.8: Ergebnisse des Messobjekts *S_Deserialize*

Da im Versuchsaufbau die Client-Seite immer durch das selbe System repräsentiert wird, wird an dieser Stelle lediglich ein Messergebnis präsentiert, zu sehen in Abbildung 7.9.

C_Invoke Das Messobjekt *C_Invoke* enthält alle Messobjekte der Server-Seite, also im wesentlichen die Zeiten des Messpunkts *S_Deserialize* und die Zeit der Netzwerkkommunikation. *C_Invoke* korreliert damit eng mit dem Messobjekt *C_Total*, als relevanter Messpunkt ist lediglich *C_Serialize* nicht enthalten. Aus diesem Grund werden in Abbildung 7.10 nicht die Ergebnisse aller vier Systeme präsentiert, sondern nur die Ergebnisse zweier Systeme aus unterschiedlichen Leistungsklassen.

C_Total *C_Total* enthält alle anderen Messobjekte und die Netzwerkkommunikation. Der Messpunkt repräsentiert somit den kompletten entfernten Serviceaufruf, die Ergebnisse sind in Abbildung 7.11 dargestellt.

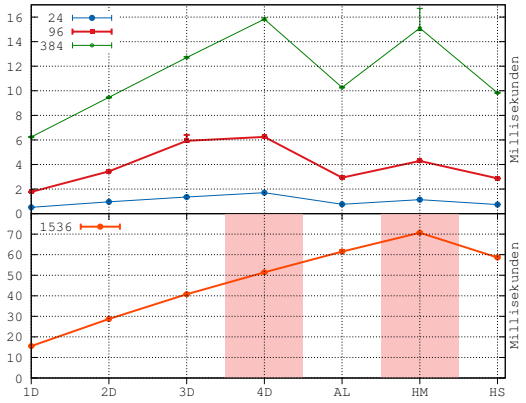
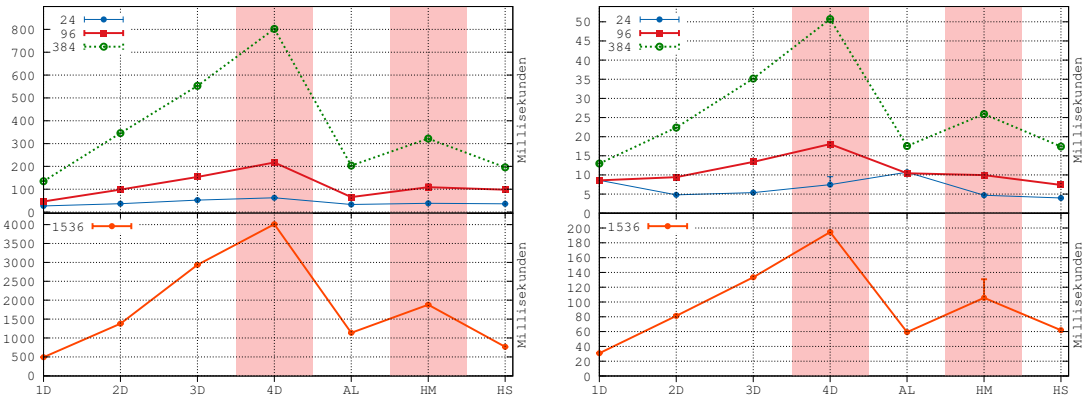


Abbildung 7.9: Ergebnisse des Messobjekts *C_Serialize*



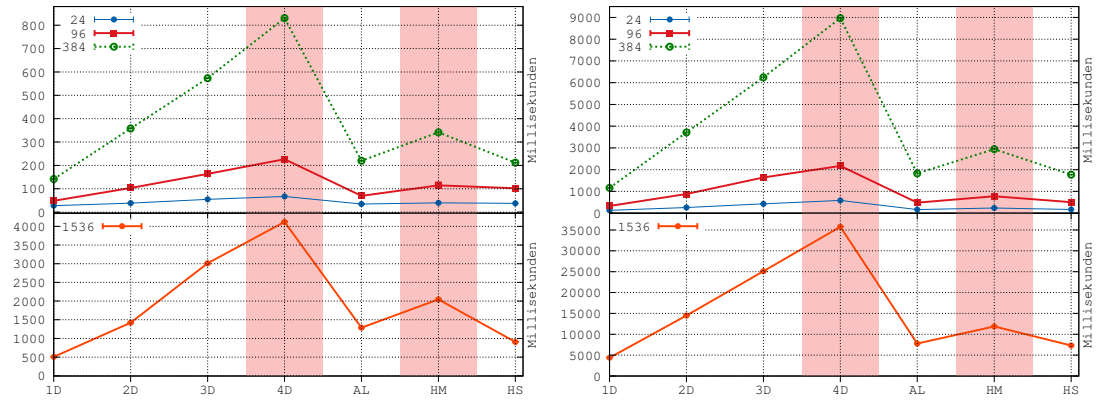
(a) BeagleBone Black **(b)** Personal PC

Abbildung 7.10: Ergebnisse des Messobjekts *C_Invoke*

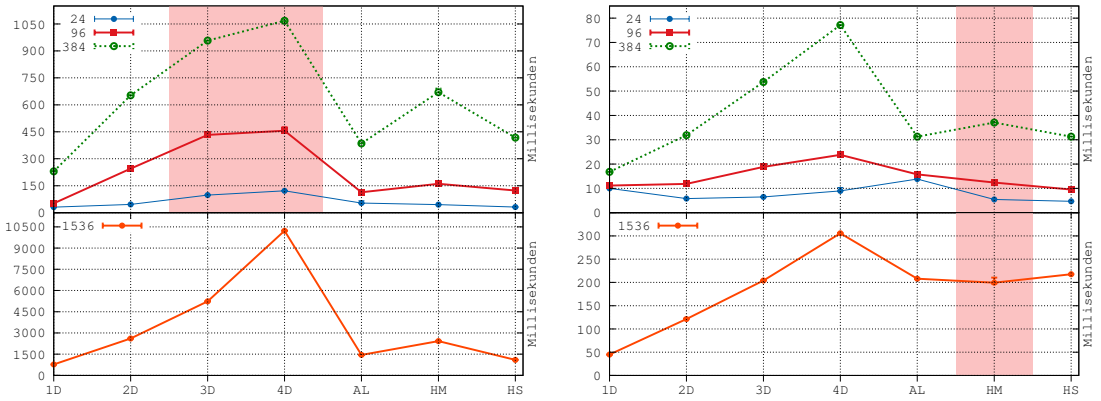
Nachrichtengröße Die Nachrichtengröße wurde für die Eingabedaten sowohl in der Anzahl von XML-Zeichen als auch in der Größe der TCP-Übertragung gemessen. Die Rückgabedaten wurden nicht vermessen, da diese sehr klein und in jedem Szenario gleich sind. Abbildung 7.12 zeigt die beiden Messergebnisse.

Datenstrukturierung Die Messergebnisse der Untersuchungen bezüglich der unterschiedlichen Daten-Strukturierung innerhalb eines Arrays ist in Abbildung 7.13 zu sehen.

Last-Messungen Im Zuge der Last-Messungen wurde die Skalierung in Bezug auf mehrere parallele Anfragen untersucht. So wurden für die Zielsysteme Personal Computer und RaspberryPi die Dauer pro Aufruf (Abbildung 7.14) und die CPU-Auslastung (Abbildung 7.15) vermessen. Für die anderen Systeme standen keine CPU-Profiler zur Verfügung.



(a) BeagleBone Black **(b)** RaspberryPi



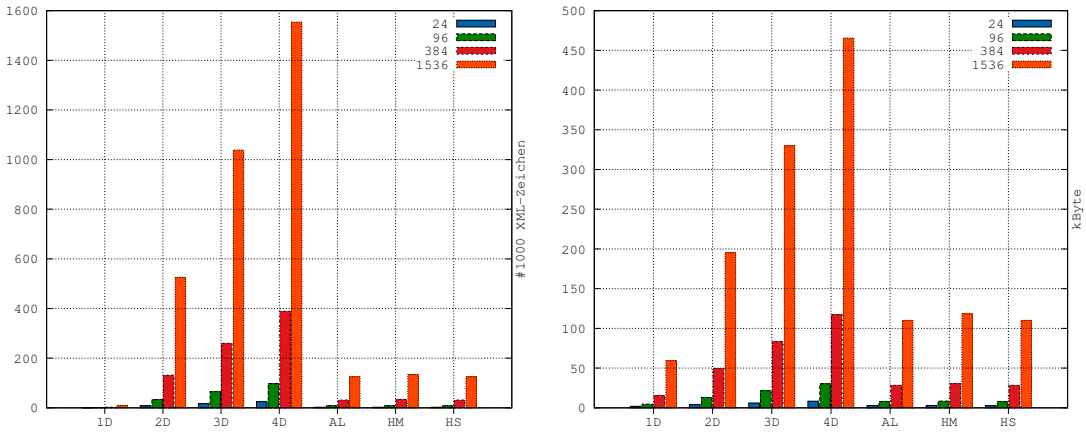
(c) Samsung Galaxy Tab **(d)** Personal Computer

Abbildung 7.11: Ergebnisse des Messobjekts *C_Total*

Interpretation der Messergebnisse

Wie in Unterabschnitt 7.7.1 erläutert, wurden bei der Auswertung der gemessenen Daten Konfidenzintervalle berechnet. Aufgrund des gewählten Verfahrens zur Berechnung der Intervalle und der Vielzahl an Messungen sind die Konfidenzintervalle sehr schmal, so dass diese kaum in den Diagrammen sichtbar sind.

S_Deserialize Bei der Analyse der Messdaten vermittelt das Messobjekt *S_Deserialize*, dargestellt in Abbildung 7.8, einen grundsätzlichen Eindruck über die Komplexität der Daten-Deserialisierung und beantwortet somit einen Teilaspekt (**M1.2**) der Fragestellung **Q1**. Es fällt direkt auf, dass auf allen Systemen die Deserialisierung von Arrays grundsätzlich aufwändiger ist als die Deserialisierung von Collections. Bereits ab zwei Array-Dimensionen ist das Niveau der Collections erreicht. Diese Ergebnisse sind nicht weiter verwunderlich, da die Komplexität eines multidimensionalen Array-XML-Dokuments



(a) XML Nachrichtengröße **(b)** TCP Nachrichtengröße

Abbildung 7.12: Ergebnisse der Messobjekte *S_XML_Size* und *S_TCP_Size*

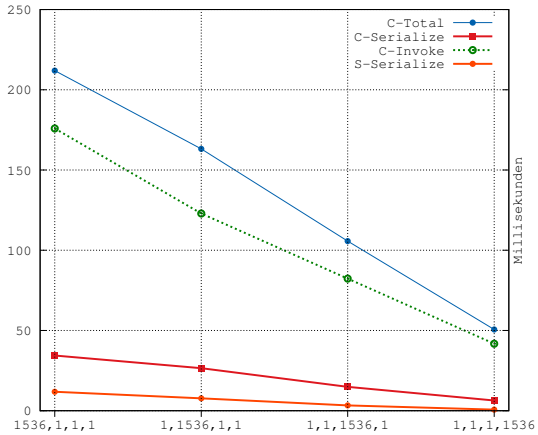
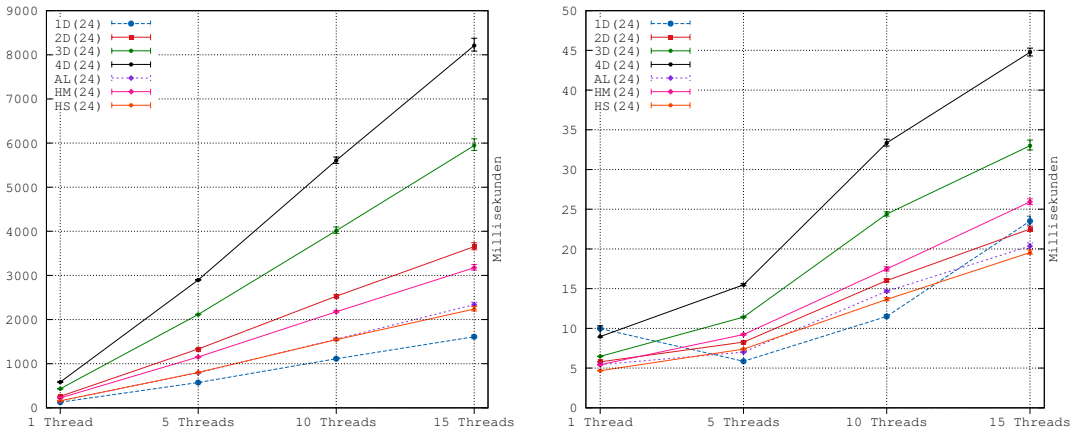


Abbildung 7.13: Ergebnisse des Messobjekts *DataStructure* für ein 4D-Array

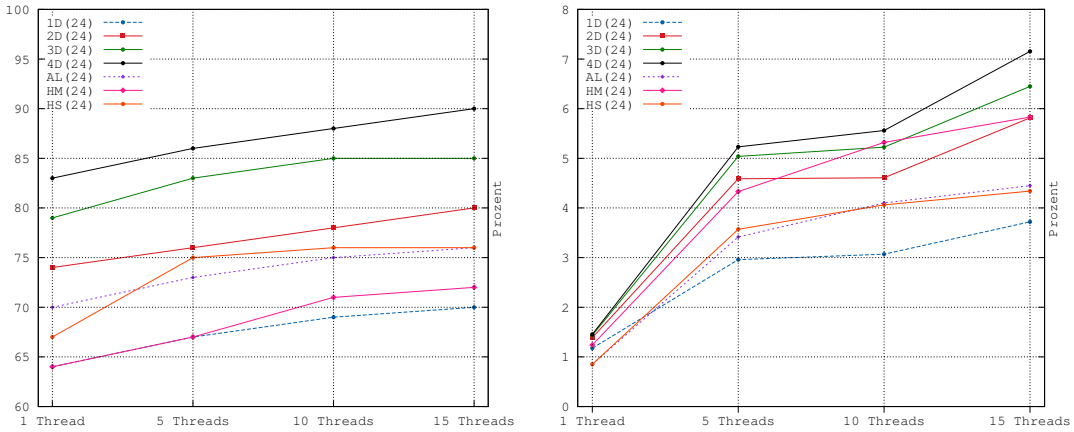
durch die notwendigen Verschachtelungen wesentlich höher ist. Innerhalb der Collections ist die `Map` auf allen Systemen aufwändiger zu deserialisieren als eine `List` oder ein `Set`, da in einer `Map` alle Werte durch einen `Key` ausgezeichnet werden.

Durch eine detaillierte Betrachtung der Messwerte für die Deserialisierung kann die Leistungsfähigkeit von Comoros für diese Aufgabe auf den unterschiedlichen Rechnerklassen bewertet werden. Die zusätzliche Rechenzeit für Arrays im Vergleich von 384 Elementen zu 1536 Elementen ergibt für das BeagleBone einen Zuwachs von 387%, für das Samsung Galaxy Tab (SGT) 279%, für das RaspberryPi 420% und für den PC 460%. Das SGT hat in diesem Punkt den geringsten Zuwachs. Diese Beobachtung gilt auch für Collections, hier sind es für das SGT 331% und für die anderen Rechnerklassen im Durchschnitt 485%. In der PC-Klasse wurde weiterhin gemessen, dass die Deserialisierung von 3D-Arrays aufwändiger als die Deserialisierung von 4D-Arrays ist. Eine Bewertung der Messwerte mit dem T-Test ergibt einen T-Wert von 0,39 und gemäß der T-Verteilung somit eine



(a) RaspberryPi **(b)** Personal Computer

Abbildung 7.14: Last-Messung: Dauer einer Anfrage



(a) RaspberryPi **(b)** Personal Computer

Abbildung 7.15: Last-Messung: CPU-Auslastung

Wahrscheinlichkeit für einen signifikanten Unterschied von gerade einmal 65%. Diese Beobachtung kann somit durch Messschwankungen verursacht worden sein.

C_Serialize Dieses Messobjekt beantwortet einen weiteren Teilaspekt (**M1.1**) der Fragestellung **Q1**. Die Ergebnisse wurden in Abbildung 7.9 illustriert. Im Gegensatz zu der Deserialisierung auf der Server-Seite fällt bei der Serialisierung auf der Client-Seite auf, dass bei steigender Anzahl der Objekte die Serialisierung der Collections immer mehr Zeit in Anspruch nimmt bis diese sogar bei 1536 Elementen langsamer als die Serialisierung eines 4D-Arrays ist. Dieses Verhalten kann nur durch den komplizierteren Zugriff auf Java-Collection-Datenstrukturen im Vergleich zu Java-Array-Datenstrukturen erklärt werden.

C_Invoke und C_Total Die Fragestellung **Q2** wird im wesentlichen durch diese beiden Messobjekte abgedeckt (**M2.1**, **M2.3**). Der Messpunkt *C_Invoke* enthält alle Server-seitigen Messobjekte und die Netzwerkkommunikation, beim Messobjekt *C_Total* wird zusätzlich noch die Client-seitige Serialisierung betrachtet. Ein interessanter Blickwinkel bei diesen Messobjekten ist die Frage, wie sich der Aufwand zwischen einem mehrdimensionalem Array und den Collections entwickelt. Auf der Server-Seite ist lediglich der bereits diskutierte Messpunkt *S_Deserialize* relevant. Hier ist der Mehraufwand für Arrays bereits deutlich sichtbar. Bei der Netzwerkkommunikation ist die Nachrichtengröße nun ein entscheidender Faktor. Abbildung 7.12 illustriert sowohl die Größe der XML-Dokumente als auch die Größe der TCP-Nachrichten für die einzelnen Datenstrukturen. Zur Analyse werden lediglich vierdimensionale-Arrays und Array-Listen im Detail verglichen. Bei 384 und 1536 Elementen ergibt sich ein prozentualer Unterschied von 413% beziehungsweise 422%. Dadurch resultiert, dass beim BeagleBone der Mehraufwand von 119 ms für das Messobjekt *S_Deserialize* (4D-Array zu ArrayList bei 384 Elementen) auf 577 ms für das Messobjekt *C_Invoke* ansteigt. Bei 1536 Elementen gibt es sogar einen Zuwachs von 415 ms zu 2648 ms. Für die anderen Rechnerklassen gelten ähnliche Werte. Durch diese Messergebnisse ist klar ersichtlich, dass bei großen Daten die Netzwerkkommunikation einen wesentlich größeren Anteil an der Gesamtlaufzeit hat als das Marshaling der Daten.

Das Messobjekt *C_Invoke* enthält nun zusätzlich die Client-seitige Serialisierung, in der, wie zuvor analysiert, Collections bei größeren Objekten langsamer als Arrays sind. Dadurch wird der für den Messpunkt *C_Invoke* beschriebene Effekt etwas abgeschwächt. Beim SGT fällt auf, dass sich 3D- und 4D-Arrays immer weiter annähern, so dass hier auf eine ineffiziente Netzwerkkommunikation bereits bei kleinen Objekten geschlossen werden kann.

Datenstrukturierung Eine Analyse hinsichtlich der Fragestellung **Q3** über die unterschiedlichen Strukturierungen der Daten innerhalb eines Arrays zeigt, dass diese große Auswirkungen auf die Effizienz des Systems hat. Es wurde ein 4D-Array untersucht, bei dem die Elemente, beginnend mit der ersten Dimension, jeweils eine Dimension weiter angeordnet wurden. Die Ergebnisse sind in Abbildung 7.13 zu sehen. Für die Client-seitigen Messobjekte steigen die gemessenen Zeiten beim Verschieben der Daten in die nächste Dimension um jeweils ca. 125%. Die Server-seitige Deserialisierung wächst jeweils um ca. 500%. Für die Nachrichtengröße ergibt sich jeweils eine Steigerung von ca. 5300% bei der Anzahl der XML-Zeichen und ca. 200% größere TCP-Nachrichten.

Die beobachteten Steigerungen resultieren aus der komplexeren Verschachtelung für den Fall, dass die Elemente in der letzten Dimension abgelegt werden. Listing 7.1 zeigt beispielhaft die unterschiedlichen XML-Instanzen für Daten in der ersten und Daten in der vierten Dimension. Der Unterschied in der Dateigröße und der Aufwand zur (De)-Serialisierung in Hinblick auf die Verschachtelung ist dabei gut zu erkennen.

Alle Messungen wurden mit der komplexeren Variante, das Ablegen der Elemente in der ersten Dimension, durchgeführt.

```
<n1:Array n1:type="[[[I">
  <n1:Array n1:type="[[I">
    <n1:Array n1:type="[I">
      <n1:Array n1:type="int">
        <n1:int>1</n1:int>
        <n1:int>2</n1:int>
      </n1:Array>
    </n1:Array>
  </n1:Array>
</n1:Array>
```

(a) Elemente in der letzten Dimension

```
<n1:Array n1:type="[[[I">
  <n1:Array n1:type="[[I">
    <n1:Array n1:type="[I">
      <n1:Array n1:type="int">
        <n1:int>1</n1:int>
      </n1:Array>
    </n1:Array>
  </n1:Array>
  <n1:Array n1:type="[[I">
    <n1:Array n1:type="[I">
      <n1:Array n1:type="int">
        <n1:int>2</n1:int>
      </n1:Array>
    </n1:Array>
  </n1:Array>
</n1:Array>
```

(b) Elemente in der ersten Dimension

Listing 7.1: 4D-Array: XML-Instanzen mit Elementen in den unterschiedlichen Dimensionen

Last-Messungen Mit Hilfe der Last-Messungen wurde ermittelt, wie die mittleren Antwortzeiten der eingesetzten Systeme mit einer unterschiedlichen Anzahl gleichzeitiger Anfragen skalieren und wie sich die CPU-Auslastung auf den verschiedenen Systemen verhält. Abbildung 7.14 und Abbildung 7.15 zeigen die Ergebnisse der Messungen.

Die CPU-Auslastung im PC steigt während aller Messungen nicht über 9%, während das RaspberryPi, mit der wesentlich leistungsschwächeren CPU, bei 15 Threads bereits zu 90% ausgelastet ist. Bei einer steigenden Anzahl gleichzeitiger Anfragen steigt auf den Systemen auch der Bedarf an Arbeitsspeicher, in dem die Daten aus den parallelen Anfragen abgelegt werden. Auch hier ist der PC mit 8 GB RAM gegenüber dem RaspberryPi mit gerade einmal 512 MB deutlich im Vorteil. Mit diesen Überlegungen lässt sich die schlechtere Skalierung auf dem RaspberryPi gut erklären. Bei 4D-Arrays ist beim RaspberryPi die mittlere Antwortzeit bei einer sequentiellen Abarbeitung um ca. 1500% schneller als bei einer gleichzeitigen Anfrage von 15 Threads. Der PC antwortet nur ca. 550% schneller.

7.7.3 Vermessung der erweiterten Architektur

Die erweiterte Architektur vergrößert den Funktionsumfang von Comoros über die Grenzen der RS-Spezifikation. Die zusätzlichen Bereiche können durch das Deployment von Komponenten und/oder eine passende Konfiguration aktiviert werden. Ziel ist nun die Evaluierung der Auswirkungen der unterschiedlichen Konfigurationen im Vergleich zur Basiskonfiguration.

Proxy-Generierung

Die unterschiedlichen Varianten der Proxy-Erzeugung wurden in Unterabschnitt 5.3.2 vorgestellt. Dabei wird zwischen Dynamischen Proxys und Proxy-Bundles mit generiertem statischen Code unterschieden, deren Effizienz untersucht werden soll. Die GQM-Zerlegung

für diese Fragestellung ist in Tabelle 7.15 dargestellt.

Tabelle 7.15: GQM-Zerlegung des Zielfaktors Evaluierung Proxy-Generierung

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Proxy-Generierung
Objekt	innerhalb von Comoros
Frage	**Metrik**
Q1: Wie unterscheidet sich die Generierungszeit für die Proxy-Varianten?	**M1.1:** Zeit der Generierung
Q2: Wie unterscheiden sich die Laufzeiten der Proxy-Varianten?	**M2.1:** Round-Trip-Time für einen entfernten Service-Aufruf

Die in Unterabschnitt 5.3.2 aufgestellte Vermutung, dass Dynamische Proxys schneller instantiiert werden können aber langsamer in der Verwendung sind, und die daraus resultierende Konfigurationsempfehlung in Tabelle 5.3 wird an dieser Stelle evaluiert. Dazu muss die Generierung der Proxys und die Laufzeit der beiden Varianten im Detail untersucht werden.

Messobjekte Die Messobjekte für die Fragestellung **Q1** ergeben sich aus der Implementierung zur Erstellung der beiden Proxy-Varianten. Die beiden Implementierungen sind in Abbildung 7.2 gegenübergestellt.

```
1 ProxyManager.handleNewDevice()
2   ...
3   ProxyManager.addService()
4     ProxyManager.registerProxies()
5       GenericProxyGenerator.createProxy()
```

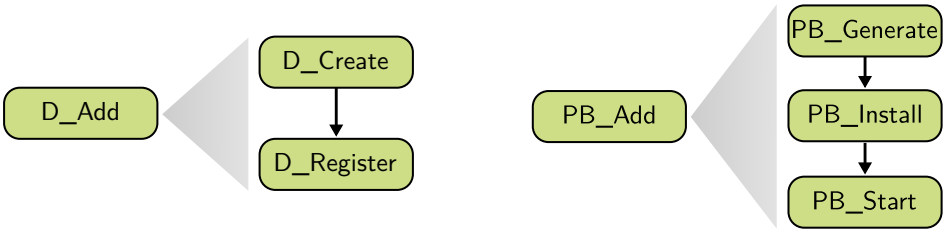
(a) Dynamische Proxys

```
1 ProxyManager.handleNewDevice()
2   ...
3   ProxyManager.addService()
4     ProxyManager.registerProxies()
5       ProxyBundleGenerator.generateProxyBundle()
6       ProxyManager.installBundle()
7       ProxyManager.startBundle()
```

(b) Proxy-Bundle

Listing 7.2: Gegenüberstellung der Implementierungen zur Proxy-Erzeugung

Aus den unterschiedlichen Implementierungen können direkt die Messobjekte zur Evaluierung der Generierungsdauer abgeleitet werden. Diese sind in Abbildung 7.18 dargestellt.

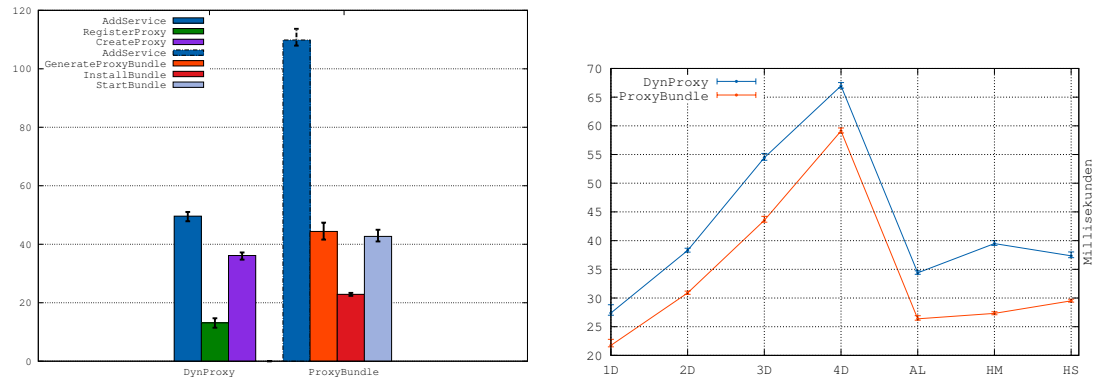


(a) Messobjekte für Dynamische Proxys **(b)** Messobjekte für Proxy-Bundles

Abbildung 7.18: Messobjekte zur Evaluierung der Proxy-Generierungsdauer

Die Vermessung der Laufzeitunterschiede für die Proxy-Varianten benötigt nur ein Messobjekt. Für die Round-Trip-Time wird wie bei der Vermessung der Kernarchitektur das Messobjekt *C_Total* verwendet.

Messergebnisse Die Ergebnisse der durchgeführten Messungen sind in Abbildung 7.19 abgebildet. Es werden lediglich Ergebnisse der Messungen mit dem BeagleBone präsentiert, da die Messergebnisse der anderen Systeme die gleichen Charakteristika aufweisen und somit keinen Mehrwert bieten. Die Laufzeiten wurden mit 24 Elementen vermessen um die Auswirkungen der (De-)Serialisierung-Zeiten möglichst gering zu halten.



(a) Proxy-Generierung **(b)** Laufzeiten der Proxy-Varianten

Abbildung 7.19: Vermessungen der unterschiedlichen Proxy-Varianten

Interpretation der Messergebnisse Die Analyse der Generierungs-Zeiten in Abbildung 7.19(a) ergibt einen signifikanten Unterschied zwischen den beiden Varianten. Proxy-Bundles benötigen ca. 60 ms mehr Zeit bis sie zur Verwendung zur Verfügung stehen. Das ist gegenüber den Dynamischen Proxys eine Steigerung von ca. 140%.

Bei den Laufzeiten ist, wie erwartet, die Proxy-Bundle-Variante effizienter als die Dynamischen Proxys. Im Durchschnitt ist ein Unterschied zwischen ca. 10% und ca. 25% zu erkennen. Je größer der Anteil der (De-)Serialisierung-Zeiten wird, umso geringer wird der Unterschied, da dieser Prozess in beiden Varianten identisch ist. Die Ergebnisse lassen den Schluss zu, dass, je nach verwendeter Datenstruktur, nach 6–11 Aufrufen die Verwendung von Proxy-Bundles zu empfehlen ist.

Anfrage vs. Konfigurationsgesteuerte Bereitstellung

Die beiden Varianten wurden in Unterabschnitt 5.3.1 ausführlich erläutert. Die Bereitstellung von Proxys kann entweder über Properties im Programmcode oder über externe Konfigurationen erfolgen. Die funktionalen Unterschiede und die Berechtigung beider Varianten wurden dargelegt, an dieser Stelle soll zusätzlich die Effizienz beider Bereitstellungsarten untersucht werden. Neben der Proxy-Bereitstellung auf der Client-Seite kann auch auf der Server-Seite im Zuge der Skeleton-Generierung zwischen den beiden Varianten unterschieden werden. Insgesamt ergibt sich die in Tabelle 7.16 aufgeführte GQM-Zerlegung.

Tabelle 7.16: GQM-Zerlegung des Zielfaktors Evaluierung Bereitstellungsphase

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Bereitstellungsphase
Objekt	innerhalb von Comoros
Frage	Metrik
Q1: Wie unterscheidet sich die Proxy-Bereitstellungszeit für die beiden Varianten?	**M1.1:** Zeit der Generierung
Q2: Wie unterscheidet sich die Skeleton-Bereitstellungszeit für die beiden Varianten?	**M2.1:** Zeit der Generierung

Aus Fragestellung **Q1** leitet sich indirekt die Zeit von der Anfrage bis zur ersten Benutzung des Service ab, die für die aufgestellten Empfehlungen in Tabelle 5.3 relevant ist. Natürlich spielt dafür auch noch die zuvor betrachtete Proxy-Art eine Rolle. Unterschiedliche Skeleton-Varianten existieren nicht.

Messobjekte Auf der Client-Seite wird die Zeit von der Service-Anfrage bis zur Absendung der Suchanfrage nach einem *Remote Service* in das Netzwerk gemessen. Für die anfragengesteuerte Bereitstellung ergibt sich so der Messpunkt *C_ReqDeploy*, der das Auslesen der Properties eines installierten Service-Trackers, deren Verarbeitung und das Absenden der Suchanfrage (hier: DPWS-Suchanfrage) enthält. Bei der konfigurationsgesteuerten Bereitstellung enthält der Messpunkt *C_ConfDeploy* die Zeit in der die übermittelte Konfiguration an den *OSGi Config Admin* übergeben wurde und ebenfalls die Verarbeitung der Konfiguration sowie das Versenden der DPWS-Suchanfrage.

Die Server-Seite kann auf gleiche Weise betrachtet werden, nur wird hier keine DPWS-Suchanfrage abgesendet, sondern die lokale OSGi-Service-Registry nach einem passenden Service befragt. Somit ergeben sich hier die Messobjekte *S_ReqDeploy* und *S_ConfDeploy*.

Messergebnisse Abbildung 7.20 zeigt die Ergebnisse aller Messobjekte – sowohl die Client-Seite als auch die Server-Seite – im Überblick.

Interpretation der Messergebnisse Die konfigurationsgesteuerte Bereitstellung ist insbesondere für die Verwendung von Legacy-Systemen unerlässlich. Gegenüber der anfragengesteuerten Bereitstellung müssen allerdings erhebliche Performanz-Einbußen hingenommen werden. Die Proxy-Bereitstellung wurde, wie im Messzenario vorgesehen, auf einem Laptop durchgeführt. Die Messungen für das Messobjekt *C_ReqDeploy* wurden im Mittel in 1,6 ms absolviert, die Ergebnisse für das Messobjekt *C_ConfDeploy* liegen im Mittel bei 31 ms und damit ca. 1800% langsamer. Auf der Server-Seite wurde die Systeme BBB, PC und RP betrachtet. Es fällt direkt auf, dass es in der Klasse der eingebetteten Systeme eine deutliche Diskrepanz in den jeweiligen Messobjekten existiert, während in der PC-Klasse der Unterschied wesentlich geringer ausfällt. Die absoluten Zahlen liegen

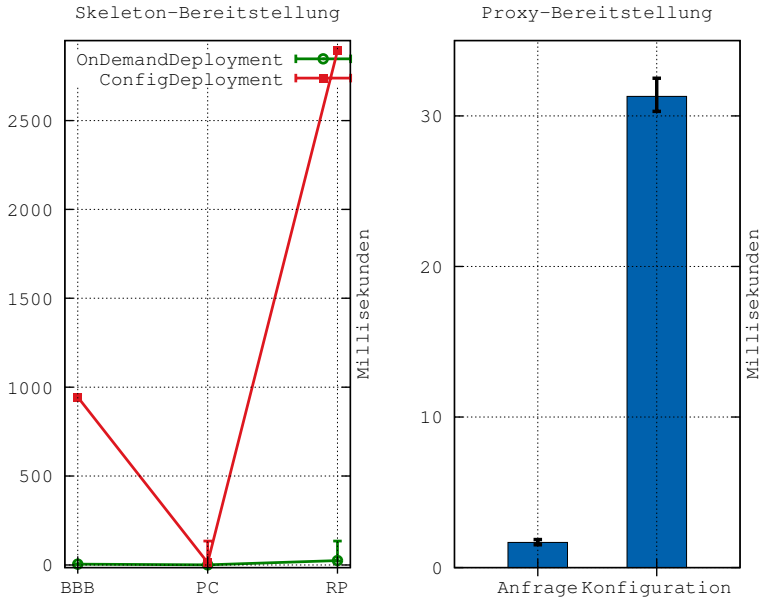


Abbildung 7.20: Ergebnisse der Vermessungen zur Bereitstellung von Skeleton und Proxy

beim BBB bei 943 ms gegenüber 4,2 ms (22000%), beim RP 2892 ms gegenüber 24 ms (11700%) und beim PC 12,6 ms gegenüber 0,5 ms (2600%).

Insgesamt wird so die Empfehlung aus Tabelle 5.3 untermauert. In einem dynamischen Umfeld wird durch die konfigurationsgesteuerte Bereitstellung der erste Service-Zugriff extrem verzögert. In statischen Umgebungen können aber durch zuvor über Konfigurationen definierte Kommunikationsmuster sogar die Bereitstellungszeiten der anfragengesteuerten Bereitstellung eingespart werden. Hier ist aber noch zu beachten, dass in diesem Falle eventuell ungenutzte Proxys im System existieren, die unnötig Arbeitsspeicher verbrauchen.

Event-basierte Kommunikation

In der Event-basierten Kommunikation (siehe Abschnitt 6.2) wird zwischen der Verwendung des Event-Converters und zwischen der impliziten asynchronen Kommunikation über die Verteilung von Event-Handlern unterschieden. Die Evaluierung der beiden Varianten analysiert, ob es zwischen dieses ein Unterschied bezüglich der Effizienz besteht. Es ergibt sich die in Tabelle 7.17 dargestellte GQM-Zerlegung.

Messobjekte Die Messobjekte zur Analyse der Event-basierten Kommunikation bewerten die Round-Trip-Time für beide Varianten (*EC_RTT* und *EH_RTT*). Das plattformübergreifendes Versenden eines Events wird durch folgenden Ablauf beschrieben:

1. `Bundle A` auf `Plattform 1` erstellt `Event e1`
2. `Event e1` wird an den lokalen EventAdmin übergeben
3. Das Event wird variantenspezifisch übermittelt

Tabelle 7.17: GQM-Zerlegung des Zielfaktors Evaluierung der Event-basierten Kommunikation

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	der Event-basierten Kommunikation
Objekt	innerhalb von Comoros
Frage	**Metrik**
Q1: Wie unterscheidet sich Effizienz der beiden Kommunikationsvarianten?	**M1.1:** Round-Trip-Time EventConverter
	M1.2: Round-Trip-Time EventHandler

4. `Bundle B` auf `Plattform 2` empfängt `Event e1`
5. `Bundle B` auf `Plattform 2` erstellt `Event e2`
6. `Event e2` wird an den lokalen EventAdmin übergeben
7. Das Event wird variantenspezifisch übermittelte
8. `Bundle A` auf `Plattform 1` empfängt `Event e2`

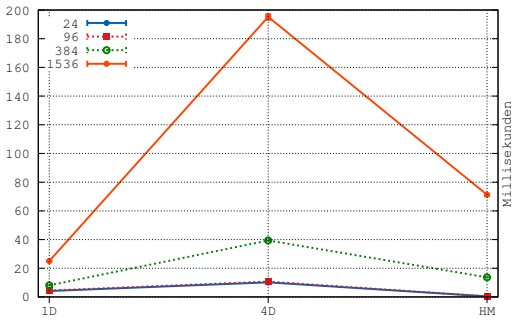
Zur Bestimmung der Round-Trip-Time werden Messpunkte vor dem ersten Schritt und nach dem letzten Schritt eingefügt.

Weitere Messobjekte sind zur Bewertung der aufgeworfenen Fragestellung nicht vorgesehen. Eine Analyse der Größe der versendeten Nachrichten ist nicht notwendig, da beide Varianten den gleichen (De-)Serialisierungsmechanismus verwenden und somit keine Unterschiede zu erwarten sind.

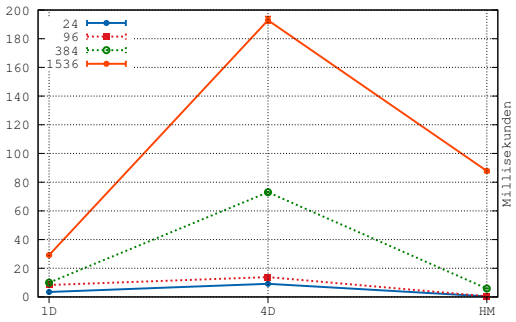
Messergebnisse Die Ergebnisse der beiden Messobjekte sind in Abbildung 7.21 abgebildet. Es wurden Messungen für das einfachste Array (eindimensional), das komplexeste Array (vierdimensional) und die komplexeste Collection (`HashMap`) jeweils auf dem RaspberryPi und dem BeagleBone durchgeführt.

Interpretation der Messergebnisse Die Analyse der Ergebnisse ergibt, dass sich die Verwendung des Event-Converters und des Event-Handler-Mechanismus ähnlich performant verhalten. Der Event-Converter ist geringfügig schneller als der Event-Handler-Mechanismus, was sich insbesondere bei vierdimensionalen Arrays mit 384 Elementen bemerkbar macht. Auf dem BeagleBone unterscheiden sich die beiden Varianten um den Faktor 1,9, auf dem RaspberryPi um den Faktor 1,4. Eine Überprüfung mit dem t-Test ergibt für das BeagleBone einen t-Wert von 1,2 und somit eine Wahrscheinlichkeit von 89% für einen signifikanten Unterschied zwischen den beiden Stichproben. Für das RaspberryPi ergibt sich eine Wahrscheinlichkeit von immer noch 70%.

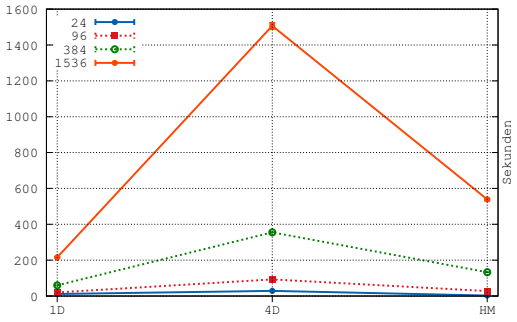
Bei 1536 Elementen ist hingegen kaum noch ein Unterschied zwischen den beiden Varianten zu erkennen. Hier dominiert einfach die Zeit zur (De-)Serialisierung. Auch bei einer kleinen Anzahl von Elementen (24 und 96) ist der Unterschied gering, der Event-Converter ist nur unwesentlich schneller. Die (De-)Serialisierungszeit ist hier gering,



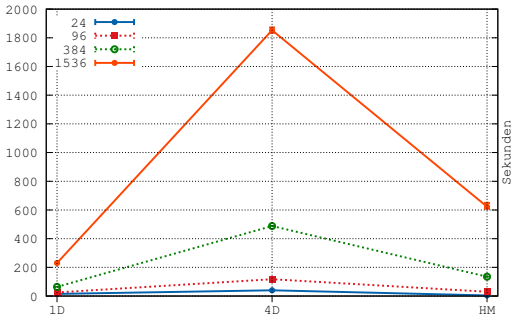
(a) BeagleBone Black – Event-Converter



(b) BeagleBone Black – Event-Handler



(c) RaspberryPi – Event-Converter



(d) RaspberryPi – Event-Handler

Abbildung 7.21: Ergebnisse der Messobjekte *EC_RTT* und *EH_RTT*

allerdings sind die Nachrichten auch so klein, dass eine effizientere Nachrichtenverarbeitung nur gering ins Gewicht fällt.

Zusammenfassend kann festgehalten werden, dass der Comoros.EventConverter insgesamt leichte Performanz-Vorteile gegenüber der impliziten Event-basierten Kommunikation mittels verteilter EventHandler hat. Zusammen mit den in Abschnitt 6.2 beschriebenen funktionalen Vorteilen ist die Verwendung des Event-Converters klar empfehlenswert.

Erweitertes Marshaling

In Abschnitt 6.1 wurde ein Prozess zum generischen Marshaling eingeführt, durch den der Benutzer von der Aufgabe zur Entwicklung eigener Marshaller befreit wurde. In diesem Abschnitt wurde bereits erläutert, dass ein generischer Marshaling-Prozess in der Mehrheit umfangreichere XML-Dokumente erzeugt als ein eigens für ein bestimmtes Java-Objekt entwickeltes XML-Schema. Aus diesem Grund wurde die Möglichkeit zum Hinzufügen von Transformationspattern geschaffen. Dieser Abschnitt untersucht nun die Performanz-Unterschiede zwischen dem Marshaling der RS-Spezifikation, dem erweiterten Marshaling und den Auswirkungen der Transformationspattern. Die GQM-Zerlegung ist in Tabelle 7.18 zu sehen.

Messobjekte Als Messobjekte wurden die aus der Vermessung der Kernarchitektur bekannten *C_Total*, *C_Serialize*, *S_Deserialize* und *XML_Size* verwendet, so dass ein

Tabelle 7.18: GQM-Zerlegung des Zielfaktors Evaluierung des erweiterten Marshaling

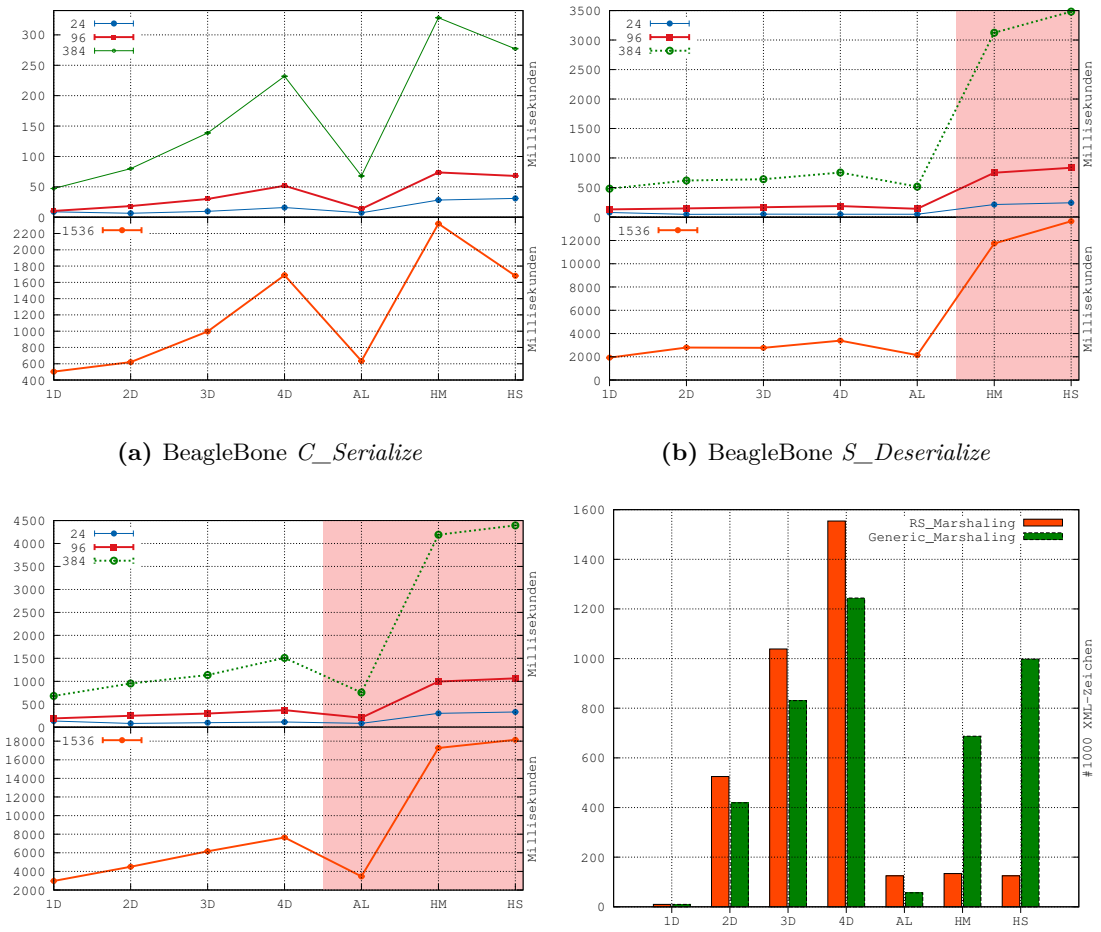
Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	des erweiterten Marshaling
Objekt	der erweiterten Comoros-Architektur
Frage	**Metrik**
Q1: Welche Unterschiede hinsichtlich der Effizient bestehen zwischen dem RS-spezifischen Marshaling und dem erweiterte Marshaling?	**M1.1:** Zeit der (De-)Serialisierung **M1.2:** Größe der Nachrichten
Q2: Wie beeinflusst der Einsatz von Transformationspattern die Performanz des Marshaling-Prozess?	**M2.1:** Zeit der (De-)Serialisierung **M2.2:** Größe der Nachrichten
Q3: Wie ist der Einfluss auf die Gesamtlaufzeit eines entfernten Serviceaufrufs?	**M3.1:** Round-Trip-Time

Quervergleich einfach möglich ist. *C_Serialize* und *S_Deserialize* bewerten die (De-)Serialisierungszeiten bezüglich Fragestellung **Q1** (**M1.1**). Die Nachrichtengröße (**M1.2**) wird durch *XML_Size* bestimmt. Alle diese Messobjekte beeinflussen die Laufzeit eines entfernten Serviceaufrufs (**Q3**), die schließlich in *C_Total* gemessen wird (**M3.1**). Für die Fragestellung **Q2**, die Auswirkungen der Transformationspattern, werden die selben Messobjekte verwendet.

Messergebnisse Die Ergebnisse der Messreihen sind in Abbildung 7.22 abgebildet. Innerhalb des Messzenarios wurde als Server-System das BeagleBone gewählt.

Im Gegensatz zu den Ergebnissen der Messungen mit dem RS-spezifischen Marshaling, präsentiert in den Abbildungen 7.9, 7.8 und 7.11, fällt direkt ins Auge, dass das vierdimensionale Array nicht mehr die schlechteste Performanz hat, sondern dass sowohl die `HashMap`, als auch das `HashSet` länger für die gestellten Aufgaben benötigen. Die `ArrayList` hingegen ist beim generischen Marshaling auf einem ähnlichen Niveau wie das eindimensionale Array.

Durch den Einsatz geeigneter Transformationspattern kann die Performanz gegenüber dem generischen Marshaling deutlich gesteigert werden. Abbildung 7.23 zeigt die Messergebnisse. In allen drei Messobjekten wird ein deutlicher Performanzgewinn sichtbar. Im alles umfassenden Messobjekt *C_Total* wird der Faktor 2,6 für die `HashMap` erreicht (t-Wert 7,8: entspricht Wahrscheinlichkeit von 99,9% für einen signifikanten Unterschied), für das `HashSet` sogar der Faktor 4,4 (t-Wert von 9,6: entspricht Wahrscheinlichkeit von 100% für einen signifikanten Unterschied).



(a) BeagleBone *C_Serialize*

(b) BeagleBone *S_Deserialize*

(c) BeagleBone *C_Total*

(d) Nachrichtengröße der XML-Instanz

Abbildung 7.22: Ergebnisse der Messobjekte zur Untersuchung des erweiterten Marshaling

Interpretation der Messergebnisse Die interessantesten Ergebnisse betreffen die `HashMap` und das `HashSet`, da hier die größten Unterschiede im Vergleich zum RS-spezifischen Marshaling entstehen. Für die `HashMap` wurde bereits in Abschnitt 6.1 und Unterabschnitt 6.1.3 erläutert, warum sich aufgrund des generischen Marshaling-Prozesses das XML-Schema – was zusätzlich eine größere WSDL und damit einen langsameren Austausch von Metadaten verursacht– und damit auch die Instanzen verkomplizieren. Für das `HashSet` ist die Auswirkung noch extremer, kann aber einfach anhand des internen Aufbaus des Java-`HashSet` erläutert werden. Die Werte innerhalb eines `HashSet` werden in Java in einer `HashMap` verwaltet. Dabei wird der Key zur Ablage des Wertes verwendet und der Value wird jeweils mit einer Instanz von `java.lang.Object` belegt. Das generische Marshaling setzt genau diese Struktur um und so entsteht die folgende beispielhafte, und zur Übersicht gekürzte XML-Instanz.

```
1 <n1:in_int_sumHashSet_java.util.HashSet>
```

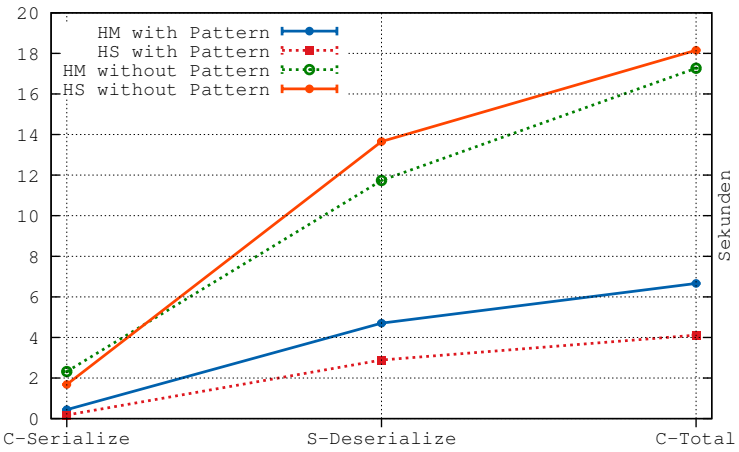


Abbildung 7.23: Performanz-Unterschiede durch den Einsatz von Transformationspattern

```
    <n1:arg_0 n2:type="n1:java.util.HashSet">
      <n1:map n2:type="n1:java.util.HashMap">
        <n1:entry n2:type="n1:HashMapEntryType">
          <n1:key n2:type="n3:int" n1:instanceof="java.lang.Integer">5</n1:key>
          <n1:value n2:type="n3:anyType" n1:instanceof="java.lang.Object"/>
        </n1:entry>
      </n1:map>
    </n1:arg_0>
</n1:in_int_sumHashSet_java.util.HashSet>
```

Diese Instanz hat mehrere gravierende Nachteile. Zum einen werden unnütze Informationen übertragen, wie der in jedem Element gleiche Value der inneren `HashMap`. Zum anderen müssen bei der Deserialisierung mit der `HashMap` eigentlich überflüssige Objekte erzeugt werden, die für den Aufbau des `HashSet` nicht benötigt werden. Zu allerletzt muss jeweils noch eine Instanz von `java.lang.Object` erzeugt werden, was innerhalb der *ObjectFactory* des generischen Marshaling besonders kompliziert ist.

Im Gegensatz zur `HashMap` und zum `HashSet` verringert sich bei allen anderen Datenstrukturen die Nachrichtengröße beim generischen im Vergleich zum RS-spezifischen Marshaling. Dies liegt vor allem daran, dass beim RS-spezifischen Marshaling zusätzlich Wert auf eine gute Menschenlesbarkeit der XML-Dokumente gelegt wurde und dadurch die XML-Dokumente teilweise etwas umfangreicher sind als notwendig. Dennoch ist insgesamt auch bei diesen Datenstrukturen ein höherer Aufwand zu erkennen. Für den Aufruf mit einem vierdimensionalen Array mit 1536 Elementen ergibt sich immer noch ein Performanzverlust von ungefähr Faktor 1,9, bei einem eindimensionalen Array mit 24 Elementen ergibt sich sogar ein Faktor 4,9. Diese Unterschiede sind darin begründet, dass bei einem zuvor bekannten XML-Schema, wie beim RS-spezifischen Marshaling der Fall, die (De-)Serialisierung deutlich effizienter durchgeführt werden kann, da hier direkt auf bekannte Strukturen zugegriffen werden kann. Durch den Einsatz von Transformationspattern kann dieser Nachteil, wie durch die Ergebnisse in Abbildung 7.23 belegt,

ausgeglichen werden.

Effiziente Transportbindungen

Um einen Performanzgewinn in der Kommunikation zu erzielen wurde in Unterabschnitt 6.5.2 ein von Moritz vorgestellter Mechanismus für Comoros implementiert. Hierbei wird die *SOAP-over-HTTP*-Bindung durch die UDP-basierte *SOAP-over-CoAP*-Bindung ersetzt. Zur Evaluierung des Performanzgewinns ergibt sich die in Tabelle 7.19 dargestellte GQM-Zerlegung.

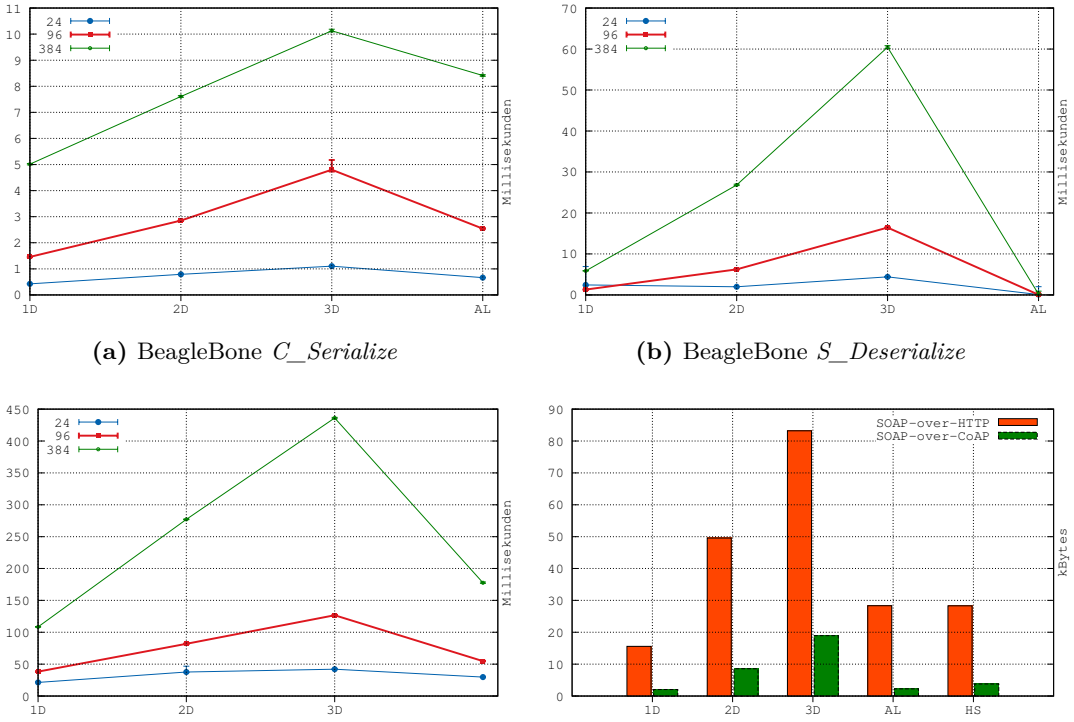
Tabelle 7.19: GQM-Zerlegung des Zielfaktors Evaluierung der SOAP-over-CoAP-Bindung

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	Effizienz der SOAP-over-CoAP-Bindung
Objekt	innerhalb von Comoros
Frage	**Metrik**
Q1: Welche Unterschiede hinsichtlich Effizienz und Leistungsfähigkeit bestehen zwischen der HTTP und der CoAP-basierten-Bindung?	**M1.1:** Zeit der (De-)Serialisierung
	M1.2: Round-Trip-Time
	M1.3: Größe der Nachrichten
	M1.4: RTT bei parallelen Anfragen

Messobjekte Die Untersuchung der Effizienz der SOAP-over-CoAP-Bindung innerhalb von Comoros besteht nur aus einer Fragestellung (**Q1**), die mit vier verschiedenen Metriken vermessen wird. Es wird erneut ein entfernter Serviceaufruf analysiert, im Detail die (De-)Serialisierungszeiten und die Gesamtlaufzeit. Dazu werden erneut die Messobjekte *C_Serialize*, *S_Deserialize* und *C_Total* verwendet. Zusätzlich werden im Zuge einer Lastmessung die *C_Total*-Zeiten für mehrere parallele Anfragen untersucht. Zur Unterstützung der Analysen werden abschließen noch die Nachrichtengrößen ermittelt.

Messergebnisse Viele Erkenntnisse zur Effizienz der CoAP-over-SOAP-Bindung können direkt in den Arbeiten von Avramov und Moritz [Avr13; Mor12] eingesehen werden. Die Messergebnisse für **M1.1**, **M1.2** und **M1.3** sind in Abbildung 7.24 abgebildet. Auf der Server-Seite wurde erneut das BeagleBone gewählt. Die Vermessungen mit den anderen Systemen bringen keine substantiellen weiteren Erkenntnisse. Da die CoAP-Implementierung für vierdimensionale Arrays eine hohe Instabilität zeigte, wurden an dieser Stelle auf Messungen verzichtet. Im Bereich der Collections werden lediglich die Ergebnisse der `ArrayList` präsentiert, da die Messungen von `HashSet` und `HashMap` zu den selben Erkenntnissen führten.

Durch die kompaktere Nachrichtenform von CoAP gegenüber HTTP und der verwendeten XML-Komprimierung EXI werden die Nachrichten, wie in Abbildung 7.24(d) zu erkennen, erheblich kleiner. Im Durchschnitt wird der Faktor 7,5 erreicht, wobei die



(a) BeagleBone *C_Serialize*

(b) BeagleBone *S_Deserialize*

(c) BeagleBone *C_Total*

(d) Nachrichtengröße der TCP-Pakete

Abbildung 7.24: Ergebnisse der Untersuchung der SOAP-over-CoAP-Bindung

`ArrayList` mit dem Faktor 12,5 am meisten profitiert. Die kleineren Nachrichten wirken sich direkt positiv auf die Nachrichtenübertragung und damit auf das Messobjekt *C_Total* aus.

Verglichen mit den Ergebnissen der SOAP-over-HTTP-Bidndung (siehe Abbildungen 7.8(a), 7.9 und 7.11(a)) ergibt sich ein signifikanter Performanzgewinn. Die *C_Serialize*-Zeiten verbessern sich um ca. 18%, die *S_Deserialize*-Zeiten um ca. 34%, was insgesamt zu einer um ca. 23% besseren *C_Total*-Performanz führt.

Die durchgeführte Last-Messung präsentiert ein weiteren deutlichen Unterschied zur HTTP-basierten Bindung. Wie in Abbildung 7.25 zu sehen, sinkt die durchschnittliche Zeit pro Anfrage mit steigender Anzahl an parallelen Anfragen. Bei 10 Threads sind alle getesteten Datentypen auf einem ähnlichen Niveau, so dass gerade Anfragen mit größerem Datenvolumina (hier z. B. 3D-Arrays) am meisten von parallelen Anfragen profitieren.

Auswirkungen sicherer Kommunikation

In Abschnitt 6.7 wurden die unterstützten Sicherheitsmechanismen für die Kommunikation erläutert. Die Auswirkungen einer abgesicherten Verbindung auf die Performanz wird in diesem Abschnitt evaluiert. Die GQM-Zerlegung für diese Fragestellung ist in Tabelle 7.20 zu sehen.

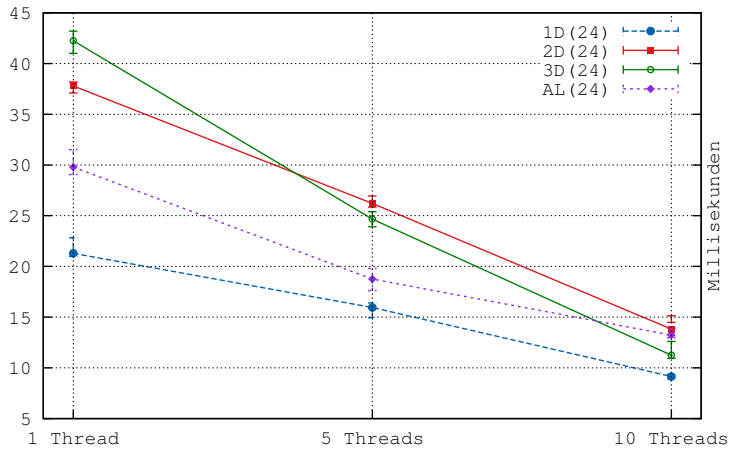


Abbildung 7.25: CoAP-Last-Messung: Dauer einer Anfrage

Tabelle 7.20: GQM-Zerlegung des Zielfaktors Evaluierung der Kommunikationssicherheit

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	Effizienz der Kommunikationssicherheit
Objekt	innerhalb von Comoros
Frage	**Metrik**
Q1: Welche Unterschiede hinsichtlich Effizienz und Leistungsfähigkeit entsteht durch die Umsetzung der Kommunikationssicherheit für entfernte Service-Aufrufe?	**M1.1:** Round-Trip-Time
	M1.2: CPU-Auslastung
	M1.3: Nachrichtengröße

Messobjekte Die Verwendung von abgesicherten HTTPS-Verbindungen hat einzig Auswirkungen auf die Kommunikation, so dass nur das Messobjekt *C_Total* von Interesse ist.

Messergebnisse Erneut wurde als Server-System das BeagleBone gewählt. Die Ergebnisse der *C_Total*-Messungen sind in Abbildung 7.26 abgebildet.

Im Vergleich zur ungesicherten HTTP-Verbindung, dargestellt in Abbildung 7.11(a), ist der Unterschied deutlich zu sehen, der durch den Mehraufwand beim Aufbau einer HTTPS-Verbindung entsteht. Beim kleinsten Datenobjekt, dem eindimensionalen Array mit 24 Elementen wirkt sich dieser Mehraufwand am deutlichsten aus (27,3 ms gegenüber 85 ms, also Faktor 3,1). Bei großen Datenobjekten ist hat die Übertragung der Daten ein größeres Gewicht und der Mehraufwand ist im Verhältnis etwas geringer. Beim vierdimensionalen Array mit 1536 Elementen steigert sich die mittlere Antwortzeit von 4118 ms auf 10304 ms, also um den Faktor 2,5.

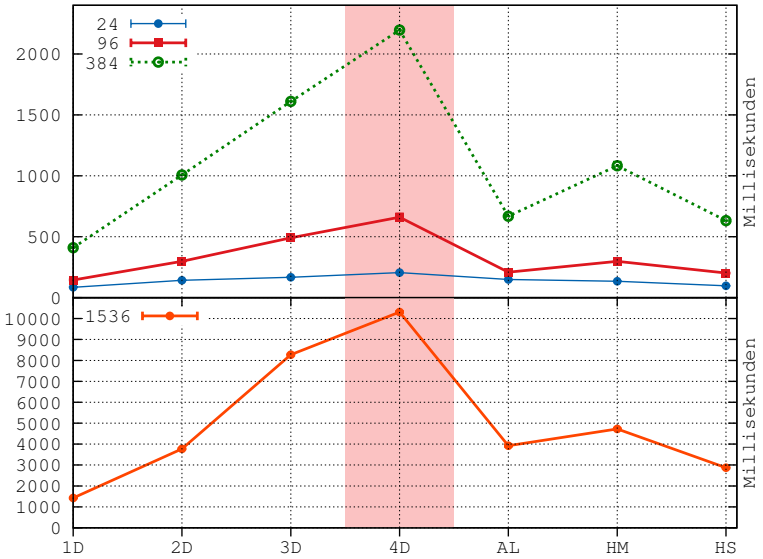


Abbildung 7.26: Round-Trip-Time mit HTTPS-Verbindung

Die höhere Round-Trip-Time lässt sich folgendermaßen erklären. Der reine Aufbau einer SSL-Verbindung ist bereits ca. 8 % langsamer als eine ungesicherte Verbindung. Hinzu kommen eine erhöhte Nachrichtengröße (hier wurde ein Faktor von durchschnittlich 3,2 gemessen) und ein erhöhter Rechenaufwand durch die Ver- und Entschlüsselung. Die CPU-Auslastung wurde nur auf dem PC vermessen, da auf dem RaspberryPi die Auslastung bei SSL-Verbindungen zumeist gegen 100 % lief und Unterschiede somit nicht gemessen werden konnten. Auf dem PC wurde ein Mehraufwand zwischen 59 % und 63 % gemessen.

7.8 Komfort

Ziel ist die Evaluierung des Entwicklungskomforts von Comoros. Dieser wurde ausführlich in der Arbeit von Krutzek und Rhein [Kru15] untersucht, deren wichtigste Ergebnisse an dieser Stelle zusammengefasst werden.

Für die Vermessung des Komforts und den Vergleich mit einem anderen Ansatz wurde eine verteilte Beispielapplikation entworfen und deren Kommunikation zwischen Applikations- und Datenbank-Server untersucht. Eine abstrakte Darstellung der Architektur ist in Abbildung 7.27 dargestellt.

Die Kommunikation zwischen den Servern wurde zum einen mittels der Comoros-Middleware und zum anderen mittels eines im Moment sehr aktuellen Architekturstils, dem REST-Prinzip, realisiert. Die GQM-Zerlegung für die Analyse des Entwicklungskomforts ist in Tabelle 7.21 einzusehen.

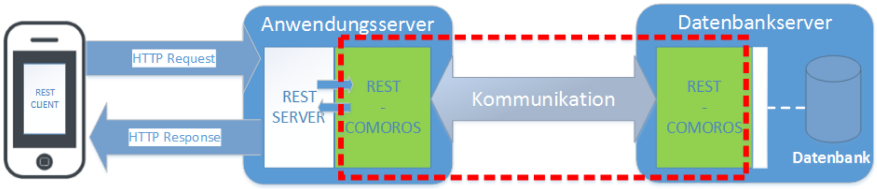


Abbildung 7.27: Einordnung der zu messenden Kommunikationselemente in eine Beispielapplikation

Tabelle 7.21: GQM-Zerlegung des Zielfaktors Evaluierung des Entwicklungskomforts

Ziel	
Zweck	Evaluierung
Betrachteter Aspekt	des Entwicklungskomforts
Objekt	des Entwicklungsprozess eines verteilten modularen Systems
Perspektive	aus Sicht des Entwicklers
Frage	**Metrik**
Q1: Wie Umfangreich ist die Integration der Kommunikationsschnittstelle?	**M1.1:** Messung der Lines of Code
Q2: Wie viel Zeit nimmt die Integration der Kommunikationsschnittstelle in Anspruch?	**M2.1:** Zeit der Einarbeitung **M2.2:** Zeit der Implementierung
Q3: Wie ist die subjektive Einschätzung des Entwicklers in Bezug auf die verwendete Technik?	**M3.1:** Fragebogen

Messobjekte

Der Umfang (**Q1**) beschreibt die Größe des geschriebenen Codes zur Umsetzung einer Komponente. Je weniger Code ein Entwickler implementieren muss, umso komfortabler ist die Entwicklung des Systems. Im beschriebenen Messzenario werden genau die Zeilen Code aufsummiert, die für die Entwicklung der Kommunikation zwischen Anwendungs- und Datenbank-Server notwendig sind (**M1.1**). Entwicklungen, die für Comoros und die REST-Umsetzung gleichermaßen benötigt werden, sind nicht Bestandteil der Messung.

Q2 deckt die Fragestellung nach dem zeitlichen Aufwand des Entwicklungsprozesses ab. Dieser wird in zwei Teile unterteilt. Zum einen in die benötigte Zeit zur Einarbeitung in die notwendigen Grundlagen (**M2.1**), und zum anderen in die Zeit für die Implementierung der eigentlichen Kommunikationsschnittstelle (**M2.2**). Für die Messung der Einarbeitungszeit ist die Qualifikation des Entwicklers zu berücksichtigen. Je qualifizierter der Entwickler ist, und je größer die Vorkenntnisse in den behandelten Bereichen ist, umso geringer fällt die Einarbeitungszeit aus.

Neben den objektiv messbaren Kriterien unterliegt der Entwicklungskomfort auch einer subjektiven Bewertung (**Q3**). Diese subjektive Einschätzung wird, vergleichbar mit Usability-Bewertungen einer Software, über Fragebögen (**M3.1**) erfasst. Der für die

Bewertung von Comoros verwendete Fragebogen evaluiert unter anderem die vorhandene Dokumentation, die Möglichkeiten weitergehende Informationen zu recherchieren, den Integrationskomfort, die Umsetzung der Funktionsanforderungen und die Erweiterbarkeit.

Der Integrationskomfort einer Technik in ein bestehendes Projekt wird z. B. durch die Notwendigkeit der Anpassung bisheriger Entwicklungen oder durch das Vorhandensein von Werkzeugen beeinflusst. Der Komfort zur Umsetzung der Funktionsanforderungen beinhaltet vornehmlich die Fragestellung, ob und wie Anforderungen an die Kommunikation, z. B. die Verwendung bestimmter Datentypen, von der jeweiligen Technologie umgesetzt werden. Die Erweiterbarkeit betrachtet unter anderem Aspekte, wie aufwändig das Hinzufügen neuer Schnittstellen in das System ist oder ob neue Clients integriert werden können die eine andere Form der Kommunikation nutzen.

Messergebnisse

Die Ergebnisse der Umfangsmessungen sind in Tabelle 7.22 zu sehen. Die beiden Komponenten unterteilen sich in den Anwendungsserver (Client-Seite) und in den Datenbankserver (Server-Seite). Die Basis-Implementierung beschreibt das Gesamtsystem, die zusätzlichen Schnittstellen beziehungsweise die zusätzlichen Ressourcen beschreiben die notwendigen Zeilen Code für die Erweiterung der Kommunikation um weitere Funktionen.

Tabelle 7.22: Ergebnisse der Umfangsmessungen

Comoros	***Anwendungsserver***	***Datenbankserver***	***Summe***
Comoros Basis	37	36	73
Zusätzliche Schnittstelle	1	4	5
REST	***Anwendungsserver***	***Datenbankserver***	***Summe***
REST-Basis	33	35	68
Zusätzliche Ressource	3	11	14

Die Ergebnisse der Aufwandsmessungen sind in Tabelle 7.23 zusammengefasst. Wie erwähnt, ist der Aufwand in Einarbeitungszeit und Implementierungszeit unterteilt. Die Einarbeitungszeit beinhaltet sowohl die Einarbeitung in grundlegende Technologien (Java, OSGi, Webservices) und in die speziellen Frameworks (Comoros, REST). Die Einarbeitungszeit korreliert stark mit den Vorkenntnissen des Entwicklers. Aus diesem Grund wurde zusätzlich die Erfahrung des Entwicklers erfasst, das auf vier Stufen abgebildet wird. Diese Stufen ergeben sich am Beispiel von Comoros aus den Kombinationen der Punkte Java-Erfahren/Neuling, OSGi-Erfahren/Neuling und Comoros-Erfahren/Neuling.

Die Messergebnisse in Tabelle 7.23 sind für Comoros über Java-Erfahrene und OSGi- und Comoros-Neulinge ermittelt worden. Für REST wurden Java- und REST-erfahrene Entwickler evaluiert. Neben der Einarbeitungszeit und der Zeit für die Implementierung des Gesamtsystems wurde auch der Aufwand für Funktionserweiterungen erfasst.

Die Messungen der subjektiven Komfort-Einschätzung sind in Tabelle 7.24 aufgelistet. Die einzelnen Fragen konnten von den Probanden mit einem Wert zwischen 0 und 10 bewertet werden.

Tabelle 7.23: Ergebnisse der Aufwandsmessungen

Arbeitsschritt Comoros	*Benötigte Zeit*
Einarbeitungszeit	86
Implementierung Basis	14
Zusätzliche Schnittstelle	0,5
Arbeitsschritt REST	***Benötigte Zeit***
Einarbeitungszeit	30
Implementierung Basis	11
Zusätzliche Schnittstelle	0,75

Tabelle 7.24: Ergebnisse der subjektiven Einschätzung

Frage	*Comoros*	*REST*
Dokumentation	9	10
Informations-Recherche	6	10
Integration	10	7
Anforderungsumsetzung	10	10
Erweiterbarkeit	8	5
Summe	43	42

Interpretation der Messergebnisse

Abbildung 7.28 zeigt den Programmierumfang in Bezug auf eine bestimmte Anzahl an Kommunikationsschnittstellen zwischen den Anwendungskomponenten.

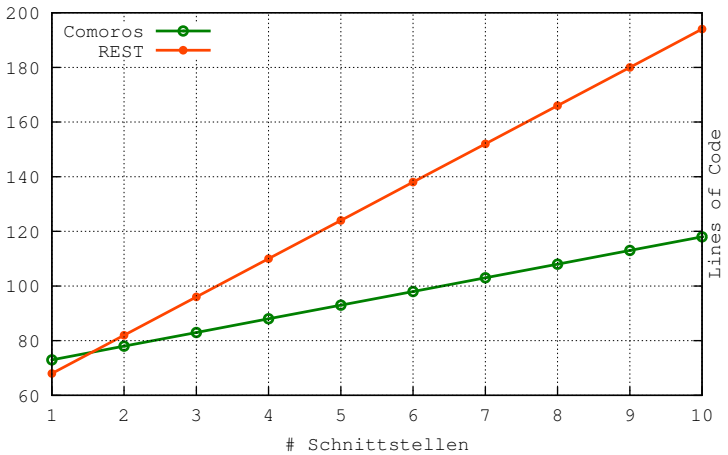
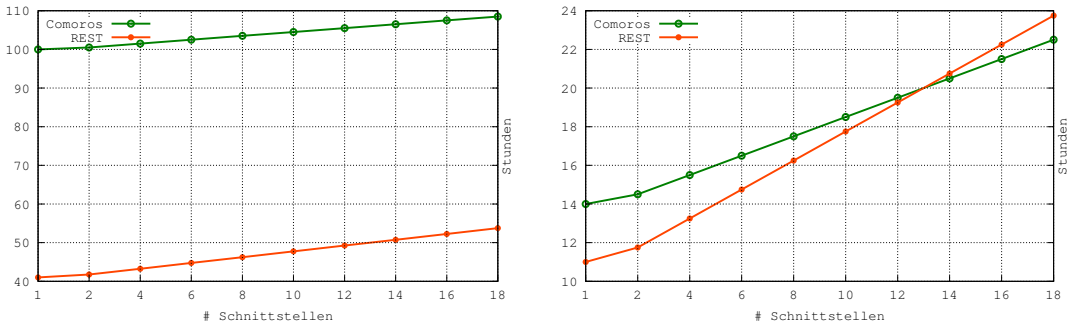


Abbildung 7.28: Implementierungsumfang für Kommunikationsschnittstellen

Ab einer Erstellung von zwei Schnittstellen ist der Aufwand der Comoros-basierten Implementierung bereits geringer. Der Abstand zwischen den beiden Vergleichsimplementierung nimmt kontinuierlich zu, bereits ab 20 Schnittstellen ist der Aufwand einer REST-basierten Implementierung doppelt so groß. Die Messungen beziehen sich auf die

anfragengesteuerte Bereitstellung der entfernten Services. Bei einer konfigurationsgesteuerten Bereitstellung würden Services über eine grafische Benutzeroberfläche verteilt werden, der Aufwand an zu schreibenden Code fällt dann noch geringer aus.

Die Ergebnisse der Aufwandsmessungen sind in Abbildung 7.29 visualisiert. Es ist zu erkennen, dass die Einarbeitungszeit für eine Comoros-basierte Entwicklung deutlich höher ist. Das liegt vor allem daran, dass OSGi, als Basistechnologie, das komplexere Themengebiet darstellt. Wie aus Tabelle 7.23 zu entnehmen, ist der Aufwand für die Erstellung weiterer Schnittstellen im Vergleich zu einer REST-basierten Implementierung aber geringer. Würde man die Zeiten hochrechnen, würde sich der zeitliche Aufwand nach 237 Schnittstellen gleichen.



(a) Aufwand mit Einarbeitungszeit **(b)** Aufwand ohne Einarbeitungszeit

Abbildung 7.29: Implementierungszeiten für die Kommunikation verteilter Komponenten

In Abbildung 7.29(b) wird die Einarbeitungszeit nicht berücksichtigt. Hier wird nur die Implementierung der Basis-Schnittstelle und weiterer Schnittstellen bewertet. In diesem Szenario ist bereits ab der 13. Schnittstelle der Aufwand der Comoros-basierten Entwicklung geringer. Wird das Szenario betrachtet, dass ein bestehendes Projekt in ein verteiltes System integriert werden soll, ist die Comoros-basierte Entwicklung von Anfang an in Vorteil. Durch das Legacy-Konzept können bestehende Komponenten ohne Quelltext-Anpassung als Teil eines verteilten Systems agieren.

Die Auswertung des Fragebogens in Tabelle 7.24 zur Bewertung des subjektiven Empfindens ergibt 43 Punkte für die Comoros-basierte und 42 Punkte für die REST-basierte Entwicklung. Der Vorteil einer REST-basierten Entwicklung liegt nach Analyse der Ergebnisse im einfachen Einstieg und gut zu recherchierenden Informationen. Die Stärken von Comoros liegen eindeutig in der Integration und der Erweiterbarkeit.

Insgesamt bleibt festzustellen, dass die Comoros-basierte Entwicklung in allen drei Fragestellungen einen besseren Wert als die Vergleichsimplementierung erreicht. Dadurch darf bereits auf einen höheren Entwicklungskomfort geschlossen werden. Die Auswertung zeigt aber auch, wie stark der Kenntnisstand eines Entwicklers den Entwicklungskomfort beeinflusst. Ohne eine Einarbeitung sind die Vorteile von Comoros in den Fragestellungen **Q2** und **Q3** deutlich höher.

7.9 Zwischenfazit

Für die Bewertung der Softwarequalität in diesem Kapitel wurde ein speziell auf Comoros angepasstes Qualitätsmodell auf Basis des Modells von Schweiggert [Sch85] entwickelt (siehe Abbildung 7.1). Die einzelnen Blätter des Baumes, die Funktionsabdeckung, die Wiederherstellbarkeit, die Sicherheit, der Komfort, die Flexibilität, die Effizienz, die Strukturierung und die Konfigurierbarkeit wurde anschließend detailliert nach dem GQM-Ansatz untersucht.

Die Messungen belegen insgesamt sowohl den erheblichen Umfang von Comoros auf der einen Seite, andererseits aber auch die gute Strukturierung, so dass die Weiterentwicklung und Wartung der Middleware leicht möglich ist. Die Entwicklungsfähigkeit wird zusätzlich durch eine umfassende Konfigurierbarkeit unterstützt. Weiterhin wurde eine große Zuverlässigkeit festgestellt.

Die Untersuchung der Leistungsfähigkeit und Effizienz von Comoros wurde unter Berücksichtigung verschiedener Hardwareklassen durchgeführt. Auf diese Weise wurden zwar der Performanz-Unterschied zwischen den einzelnen Klassen deutlich, es ist nach den Untersuchungen aber auch belegt, dass Comoros auch in der Klasse der eingebetteten Systeme anwendbar ist. Neben der Untersuchung der grundsätzlichen Effizienz wurden auch die Auswirkungen der unterschiedlichen Konfigurationen der erweiterten Architektur untersucht. Die Ergebnisse dieser Messungen haben die in Tabelle 5.3 aufgestellten Konfigurationsempfehlungen für die unterschiedlichen Anwendungsfälle nachgewiesen.

In der abschließenden Untersuchung der Benutzerfreundlichkeit wurde neben einer grundsätzlichen Bewertung von Comoros aus Nutzersicht auch ein Vergleich mit einer REST-basierten Entwicklung eines verteilten Systems absolviert. Die Messungen haben deutlich gezeigt, dass Comoros einen hohen Entwicklungskomfort besitzt, insbesondere bei der Integration in ein bestehendes Umfeld. Innerhalb der Analyse wurden auch die Teile der Comoros-Middleware identifiziert, die in besonderem Maße den Komfort erhöhen:

- Generisches Marshaling-Konzept: Keine manuelle Arbeit für die (De-)Serialisierung bei der Verwendung beliebiger Datentypen.
- Unterstützung von Altkomponenten: Komfortable Integration bestehender OSGi-Services in ein verteiltes Umfeld.
- Breiter Einsatzbereich: Unterstützung verschiedener Kommunikationsprotokolle (DPWS, CoAP, UPnP, etc.).
- Werkzeugunterstützung: Komforterhöhung in den Bereichen Marshaling-Aufgaben, Geräteintegration und Konfiguration.

Zusammenfassend kann nach Auswertung aller Punkte festgehalten werden, dass die Comoros-Middleware eine gute Softwarequalität besitzt.

8 Zusammenfassung & Ausblick

Die Dissertation leistet einen Beitrag zur effizienten Entwicklung verteilter ambienter Anwendungen im Umfeld des Internet der Dinge. Die Vermeidung von Insellösungen, die durch die Anwendung proprietärer Protokolle und Schnittstellen entstehen, sowie die Anwendbarkeit in hoch dynamischen und Ressourcen beschränkten Umgebungen sind dabei die wichtigsten Anforderungen. Dazu wurde mit Comoros eine Middleware entworfen, deren Anforderungen aus den genannten Anforderungen und den Anwendungsdomänen in Kapitel 4 detailliert abgeleitet wurden.

Die Comoros Middleware basiert dabei im wesentlichen auf den folgenden drei neuartigen Merkmalen:

- Durch die Kombination der OSGi-basierten Softwareentwicklung, mit der eine hohe Dynamik realisiert werden kann, mit einer verteilten Serviceorientierten Architektur auf Basis einer Webservice-Implementierung für eingebettete Geräte (DPWS), wurde eine Möglichkeit geschaffen verteilte ambiente Anwendungen konform zu wohl definierten und offenen Standards zu entwickeln. Dazu wurde in Kapitel 5 eine Architektur entworfen, die den OSGi-Remote-Service-Standard umsetzt und die Anforderungen in Bezug auf die Anwendung des DPWS löst.
- Die Remote-Service-Implementierung wurde um spezielle Fähigkeiten zur effizienten Entwicklung verteilter ambienter Anwendungen erweitert. Diese Erweiterungen werden von keiner anderen der aktuellen RS-Implementierungen angeboten. Hierbei handelt es sich insbesondere um ein erweitertes Marshaling, um eine Event-basierte Kommunikation, um die Integration von Geräten, um die Unterstützung von Altkomponenten und die Möglichkeit zu einem umfassenden Management.
- Alle Implementierungen berücksichtigen den Einsatz in Ressourcen beschränkten Umgebungen. Die Effizienz der Comoros-Implementierung wurde ausführlich belegt.

Ebenso wurde der für eine hohe Akzeptanz wichtige Benutzungskomfort evaluiert.

Die Comoros-Middelware wurde und wird in verschiedenen Forschungsprojekten eingesetzt. Ursprünglich wurde Comoros innerhalb des ITEA2-Projekts OSAmI entwickelt. Dort wurde Comoros sowohl im deutschen Teilprojekt in der Medizintechnik als auch im internationalen Teil in der Heimautomatisierung verwendet. Anschließend wurde Comoros im BaaS-Projekt, das sich mit der Gebäudeautomatisierung befasst, als Lösung für verteilte OSGi-Plattformen ins Projekt eingeführt. Auch im 2015 startenden Medolution-Projekt wird Comoros erneut als Lösung für die komfortable Entwicklung verteilter Anwendungen in der Medizintechnik eingesetzt. Durch diesen vielfältigen Projekteinsatz ist die Anwendbarkeit in unterschiedlichen Anwendungsdomänen hinreichend belegt. Durch die von Comoros überwundenen Insellösungen können nun sogar Domänen übergreifende Lösungen realisiert werden.

Im Feld der Unterstützung der Entwicklung verteilter Anwendungen bildet die Kommunikation zwischen den Anwendungsteilen einen der wichtigsten Schwerpunkte. Alle angesprochenen Domänen stellen dabei an diese Kommunikation unterschiedliche Anforderungen, welche die Eigenschaften der einzelnen Domänen abbilden. Diese Anforderungen sind aufgrund der dynamischen Entwicklung der Umgebungen teilweise auch unvorhersehbar. Somit sind Systeme, die sich selbst organisieren und die dynamisch zur Laufzeit ihre Struktur verändern, um so Nutzeranforderungen verlässlich zu erfüllen, ein notwendiges Kriterium. In diesem dynamischen Entwicklungsprozess kann es also zu Situationen kommen, in denen bestimmte Nutzeranforderungen in Konkurrenz zueinander stehen. Insbesondere, wenn ein System missionskritisch ist, also unter keinen Umständen ausfallen oder Schaden verursachen darf, muss sichergestellt sein, dass Funktionalität und Betriebssicherheit gewährleistet ist. Demgegenüber steht die Durchsetzung verschiedener nichtfunktionaler Anforderungen.

In einer medizintechnischen Anwendung z. B. ist eine der wichtigsten Anforderungen, dass patientenkritische Alarme zu jeder Zeit dem überwachenden Arzt mitgeteilt werden. Eine weitere nichtfunktionale Anforderung ist die Einhaltung von Datensicherheitskriterien. Das System kann aber einen solchen Zustand einnehmen, in dem durch Energieknappheit das Absenden eines Alarms unter Beibehaltung der Sicherheitskriterien unmöglich ist, da die Anwendung der Verschlüsselungsalgorithmen eine zu hohe CPU-Last und damit einen zu hohen Energieverbrauch verursacht. An dieser Stelle muss das System so adaptiert werden, dass das priorisierte Ziel, die Absendung des Alarms gesichert ist, und dafür sekundäre Ziele vernachlässigt werden. Konkret wird auf das Verschlüsseln der Alarmnachrichten verzichtet.

Comoros bietet auf der einen Seite viele Konfigurations-Alternativen in unterschiedlichen Bereichen an und auf der anderen Seite eine Management-Schnittstelle zur dynamischen Konfiguration zur Laufzeit. Die Integration in umfassende Management-Systeme ist somit vorbereitet und bietet zukünftigen Entwicklungen somit einen guten Einstiegspunkt.

In der Basis-Konfiguration, die Hauptbestandteil dieser Dissertation ist, verbindet Comoros die beiden Technologien OSGi und DPWS. Beide Technologien haben in der Forschung einen hohen Aktualitätsgrad, von dem die Comoros-Software direkt profitiert. Die kurz vorgestellte Forschungsarbeit von Moritz [Mor12] ermöglicht DPWS-Geräten

zum Beispiel die Integration in 6LoWPAN-Netzwerken, so dass durch Comoros auch OSGi-Clients diese Kleinstgeräte direkt nutzen können. Neben dem DPWS kann Comoros durch weitere Protokolle ergänzt werden, um die Anwendbarkeit in speziellen Domänen zu ermöglichen. Eine Erweiterung um das BACNet-Protokoll [Esd14] wurde bereits in einer studentischen Arbeit realisiert, so dass die Gebäudeautomatisierungs-Domäne weiter erschlossen werden kann. Durch die Ergänzung neuer Protokolle und Arbeiten an den einzelnen Protokollen bietet Comoros auch für weitere Forschungen ein interessantes Themenfeld.

Literaturverzeichnis

[Alo04] Gustavo Alonso, Fabio Casati, Harumi Kuno und Vijay Machiraju: „Web Services“. English. *Web Services*. Data-Centric Systems and Applications. Springer Berlin Heidelberg, 2004: S. 123–149 (siehe S. 33).

[Ami08] Venkat Amirisetty und Erkki Rysa: *Mobile Operational Management*. Java Specification Request (JSR) 232. Jan. 2008 (siehe S. 23).

[ANS89] ANSA: *The Advanced Network Systems Architecture (ANSA), Reference manual*. Techn. Ber. Cambridge: Architecture Project Management, 1989 (siehe S. 83).

[Atz10] Luigi Atzori, Antonio Iera und Giacomo Morabito: „The Internet of Things: A survey “. *Computer Networks* (2010), Bd. 54(15): S. 2787–2805 (siehe S. 1).

[Avr13] Kamen Avramov: „Resource-efficient Communication for OSGi Remote Services through Extension of the Comoros/JMEDS-platform by SOAP-over CoAP“. Magisterarb. TU Dortmund, 2013 (siehe S. 182, 239).

[Bac00] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord und Kurt Wallnau: *Volume II: Technical concepts of component-based software engineering*. Carnegie Mellon University, Software Engineering Institute, 2000 (siehe S. 16).

[Bal06] Keith Ballinger, David Ehnebuske, Christopher Ferris, Martin Gudgin, Canyang Kevin Liu, Mark Nottingham und Prasad Yendluri: *WS-I Basic Profile Version 1.1*. 2006 (siehe S. 36).

[Bar06] Alistair Barros, Marlon Dumas und Phillipa Oaks: „Standards for Web Service Choreography and Orchestration: Status and Perspectives“. *Business Process Management Workshops*. Hrsg. von ChristophJ. Bussler und Armin Haller. Bd. 3812. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006: S. 61–74 (siehe S. 33).

[Bas07] V. Basili, J. Heidrich, M. Lindvall, J. Munch, M. Regardie und A. Trendowicz: „GQM+Strategies – Aligning Business Strategies with Software Measurement“. *Empirical Software Engineering and Measurement,*

2007. ESEM 2007. First International Symposium on. Sep. 2007: S. 488–490 (siehe S. 49).

[Bas92] Victor R. Basili: *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. Techn. Ber. College Park, MD, USA: University of Maryland, 1992 (siehe S. 49).

[Beh11] Fabian Behr: „Implementierung und Evaluierung des Efficient XML Interchange (EXI) Standards für das JMEDS Framework“. Magisterarb. Fachhochschule Südwestfalen, 2011 (siehe S. 181).

[Bei07] Michael Beisiegel, Dave Booz, Adrian Colyer, Hal Hildebrand, Jim Marino und Ken Tam: *SCA Service Component Architecture*. März 2007 (siehe S. 97).

[Ber96] Philip A. Bernstein: „Middleware: A Model for Distributed System Services“. *Commun. ACM* (Feb. 1996), Bd. 39(2): S. 86–98 (siehe S. 27).

[Boe76] B. W. Boehm, J. R. Brown und M. Lipow: „Quantitative Evaluation of Software Quality“. *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976: S. 592–605 (siehe S. 201).

[Boh92] B. W. Bohem und W. L. Scherlis: „Megaprogramming“. *DARPA Software Technology Conference*. 1992 (siehe S. 15).

[Boo91] G. Booch: *Object Oriented Design with applications*. Bejamin/Cummings, Redwood City, 1991 (siehe S. 30).

[Boo04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris und David Orchard: *Web Services Architecture*. W3C Note NOTE-ws-arch-20040211. World Wide Web Consortium, Feb. 2004 (siehe S. 34, 39).

[Bor12] C. Bormann, A.P. Castellani und Z. Shelby: „CoAP: An Application Protocol for Billions of Tiny Internet Nodes“. *Internet Computing, IEEE* (März 2012), Bd. 16(2): S. 62–67 (siehe S. 38, 39).

[Bot08] A. Bottaro, E. Simon, S. Seyvoz und A. Gerodolle: „Dynamic Web Services on a Home Service Platform“. *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*. März 2008: S. 378–385 (siehe S. 25, 63).

[Bou08] Carine Bournez: *Efficient XML Interchange Evaluation*. World Wide Web Consortium, Working Draft WD-exi-evaluation-20080728. Juli 2008 (siehe S. 181).

[Bov14] Gérôme Bovet, Antonio Ridi und Jean Hennebert: „Toward Web Enhanced Building Automation Systems“. English. *Big Data and Internet of Things: A Roadmap for Smart Environments*. Hrsg. von Nik Bessis und Ciprian Dobre. Bd. 546. Studies in Computational Intelligence. Springer International Publishing, 2014: S. 259–283 (siehe S. 24).

[Box06] Don Box, Luis Felipe Cabrera, Craig Critchley, Francisco Curbera, Donald Furguson, Steve Graham und David Hull: *Web Services Eventing (WS-Eventing)*. Techn. Ber. W3C, 2006 (siehe S. 159).

[Bra97] Michael Bray, Kimberly Brune, David Fisher, John Foreman, Mark Gerken, John Gross, Gary Haines und Elizabeth Kean: *Software Technology Reference Guide*. Techn. Ber. Carnegie Mellon University - Software Engineering Institute, 1997 (siehe S. 214).

[Bra08] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler und François Yergeau: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. World Wide Web Consortium, Recommendation REC-xml-20081126. Nov. 2008 (siehe S. 3, 37, 102).

[Bru02] Eric Bruneton, Romain Lenglet und Thierry Coupaye: „ASM: a code manipulation tool to implement adaptable systems“. *Adaptable and extensible component systems* (2002), Bd. 30 (siehe S. 58).

[But08] Russell Butek: *Web services hints and tips: Design reusable WSDL faults*. Techn. Ber. IBM Technical Library, März 2008 (siehe S. 107).

[Cam11] M. Emilia Cambronero, Gregorio Díaz, Valentín Valero und Enrique Martínez: „Validation and verification of Web services choreographies by using timed automata “. *The Journal of Logic and Algebraic Programming* (2011), Bd. 80(1). The 2nd Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'08): S. 25–49 (siehe S. 33).

[Car13] Yudith Cardinale, Joyce Haddad, Maude Manouvrier und Marta Rukoz: „Web Service Composition Based on Petri Nets: Review and Contribution“. *Resource Discovery*. Hrsg. von Zoé Lacroix, Edna Ruckhaus und Maria-Esther Vidal. Bd. 8194. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013: S. 83–122 (siehe S. 33).

[Che11a] Min Chen, Sergio Gonzalez, Athanasios Vasilakos, Huasong Cao und Victor C. Leung: „Body Area Networks: A Survey“. *Mob. Netw. Appl.* (Apr. 2011), Bd. 16(2): S. 171–193 (siehe S. 11).

[Che12] Sheng-Tzong Cheng, Chi-Hsuan Wang und Gwo-Jiun Horng: „OSGi-based smart home architecture for heterogeneous network “. *Expert Systems with Applications* (2012), Bd. 39(16): S. 12418–12429 (siehe S. 25).

[Che11b] Stuart Cheshire und Marc Krochmal: *Multicast DNS*. Internet-Draft draft-cheshire-dnsext-multicastdns-14.txt. Fremont, CA, USA: IETF Secretariat, 14. Feb. 2011 (siehe S. 68).

[Chr03] Mary Beth Chrissis, Mike Konrad und Sandy Shrum: *CMMI Guidlines for Process Integration and Product Improvement*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (siehe S. 49).

[Chr01] Erik Christensen, Francisco Curbera, Greg Meredith und Sanjiva Weerawarana: *Web Services Description Language (WSDL) 1.1*. W3C Note. World Wide Web Consortium, März 2001 (siehe S. 101, 102).

[Cle04] Luc Clement, Andrew Hately, Claus von Riegen und Tony Rogers: *UDDI Spec Technical Committee Draft 3.0.2*. OASIS Committee Draft. OASIS, 2004 (siehe S. 34).

[Cou05] Coulouris, Jean Dollimore und Tim Kindberg: *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005 (siehe S. 27, 86).

[Cow04] John Cowan und Richard Tobin: *XML Information Set (Second Edition)*. World Wide Web Consortium, Recommendation REC-xml-infoset-20040204. Feb. 2004 (siehe S. 36).

[Cro06] Douglas Crockford: *The application/json Media Type for JavaScript Object Notation (JSON)*. Techn. Ber. RFC 4627. IETF, Juli 2006 (siehe S. 36).

[Dai11] Naci Dai, Jesus Bermejo, Felix Cuadrado Latasa, Alejandra Ruiz López, Isaac Agudo, Elmar Zeeb, Jan Krüger, Wolfgang Thronicke, Oliver Dohndorf, Christoph Fiehe und Anna Litvina: „OSAMI COMMONS - An Open Platform for Dynamic Services for Ambient Intelligence“. *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2011)*. Toulouse, France: IEEE Computer Society, 2011 (siehe S. 178).

[Day83] J.D. Day und H. Zimmermann: „The OSI reference model“. *Proceedings of the IEEE* (Dez. 1983), Bd. 71(12): S. 1334–1340 (siehe S. 2, 8).

[Dei07] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert und J.-F. Girard: „An Activity-Based Quality Model for Maintainability“. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. Okt. 2007: S. 184–193 (siehe S. 48).

[Deu06] S. de Deugd, R. Carroll, K.E. Kelly, B. Millett und J. Ricker: „SODA: Service Oriented Device Architecture“. *Pervasive Computing, IEEE* (Juli 2006), Bd. 5(3): S. 94–96 (siehe S. 26).

[Deu96] P. Deutsch: *GZIP File Format Specification Version 4.3*. United States, 1996 (siehe S. 45).

[Deu86] Deutsche Gesellschaft für Qualität: *Software-Qualitätssicherung: Aufgaben, Möglichkeiten, Lösungen*. Techn. Ber. Berlin, Köln, Offenbach: Nachrichtentechnische Gesellschaft im VDE, Arbeitsgruppe 143, 1986 (siehe S. 201).

[Doh09] O. Dohndorf, C. Fiehe, A. Litvina, I. Luck, J. Kattwinkel, F.-J. Stewing, J. Kruger und H. Krumm: „Location-Transparent Integration of Distributed OSGi Frameworks and Web Services“. *Advanced Information Networking and Applications Workshops, 2009. WAINA '09. International Conference on*. Mai 2009: S. 464–469 (siehe S. 120).

[Doh10] Oliver Dohndorf, Jan Krüger, Heiko Krumm, Christoph Fiehe, Anna Litvina, Ingo Lück und Franz-Josef Stewing: „Lightweight Policy-Based Management of Quality-Assured, Device-Based Service Systems“. *24th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2010, Perth, Australia, 20-13 April 2010*. IEEE Computer Society, 2010: S. 526–531 (siehe S. 185).

[Don08] Andrew Donoho, Jose Costa-Requena, John Fuller, Tom McGee und Alan Messer: *UPnP Device Architecture 1.1*. Techn. Ber. UPnP Forum, 2008 (siehe S. 39).

[Dur13] Jacques Durand und Gershon Janssen: *OASIS Web Services Basic Reliable and Secure Profiles (WS-BRSP)*. 2013 (siehe S. 36).

[Ede06] A.H. Eden und T. Mens: „Measuring software flexibility“. *Software, IEE Proceedings* - (Juni 2006), Bd. 153(3): S. 113–125 (siehe S. 207).

[Edw01] W.Keith Edwards und Rebecca E. Grinter: „At Home with Ubiquitous Computing: Seven Challenges“. English. *Ubicomp 2001: Ubiquitous Computing*. Hrsg. von Gregory D. Abowd, Barry Brumitt und Steven Shafer. Bd. 2201. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001: S. 256–272 (siehe S. 2).

[Efr93] B. Efron und R. Tibshirani: *An Introduction to the Bootstrap*. Miscellaneous. 1993 (siehe S. 219).

[Ehr06] Paul Ehrlich und Toby Considine: *Open Building Information Exchange (oBIX) version 1.0*. OASIS Committee Specification. Dez. 2006 (siehe S. 26).

[Erl05] Thomas Erl: *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005 (siehe S. 2, 26, 33).

[Esd14] Joern Esdohr: „OSGi-basierte Middleware zur Kommunikation mit BACnet-Komponenten in der Gebäudeautomatisierung“. Magisterarb. TU Dortmund, 2014 (siehe S. 251).

[Fie99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee: *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. 1999 (siehe S. 41).

[Fie00] Roy Thomas Fielding: „Architectural Styles and the Design of Network-based Software Architectures“. AAI9980887. Diss. Irvine: University of California, 2000 (siehe S. 32).

[For04] Ira R. Forman und Nate Forman: *Java Reflection in Action (In Action Series)*. Greenwich, CT, USA: Manning Publications Co., 2004 (siehe S. 123).

[Fra12] D. Franke, S. Kowalewski und C. Weise: „A Mobile Software Quality Model“. *Quality Software (QSIC), 2012 12th International Conference on*. Aug. 2012: S. 154–157 (siehe S. 48).

[Gar84] David Garvin: „What does product quality really mean?“ *Sloan Management Review* (1984), Bd. 26: S. 25–45 (siehe S. 48).

[Gei08] Walter Geiger und Willi Kotte: „Die Fachbegriffe Qualität und Fähigkeit“. German. *Handbuch Qualität: Grundlagen und Elemente des Qualitätsmanagements: Systeme—Perspektiven*. Springer, 2008: S. 67–82 (siehe S. 48).

[Gei95] Kurt Geihs: *Client-Server-Systeme: Grundlagen und Architekturen*. Hrsg. von Bernd Mahr, Alexander Schill und Gottfried Vossen. Thomson's aktuelle Tutorien. Internat. Thomson Publ., 1995 (siehe S. 30).

[Gel11] A Gelibert, W. Rudametkin, D. Donsez und S. Jean: „Clustering OSGI Applications Using Distributed Shared Memory“. *New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on*. Mai 2011: S. 1–8 (siehe S. 61, 62).

[Gil77] T. Gilb: *Software metrics*. Winthrop computer systems series. Winthrop Publishers, 1977 (siehe S. 49).

[Gir03] Marc Girardot und Neel Sundaresan: „Millau: an encoding format for efficient representation and exchange of XML over the Web.“ *Computer Networks* (27. Nov. 2003), Bd. 33(1-6): S. 747–765 (siehe S. 46).

[Gir13] Michele Girolami, Filippo Palumbo, Francesco Furfari und Stefano Chessa: „The Integration of ZigBee with the GiraffPlus Robotic Framework“. *Evolving Ambient Intelligence*. Hrsg. von MichaelJ. O'Grady, Hamed Vahdat-Nejad, Klaus-Hendrik Wolf, Mauro Dragone, Juan Ye, Carsten Röcker und Gregory O'Hare. Bd. 413. Communications in Computer and Information Science. Springer International Publishing, 2013: S. 86–101 (siehe S. 25).

[Goe05] Brian Goetz: *Java theory and practice: Decorating with dynamic proxies*. Techn. Ber. IBM Technical Library, 2005 (siehe S. 99).

[Gor10] Torsten Gorath, Marco Eichelberg, Andreas Hein, Elmar Zeeb, Frank Golatowski und Dirk Timmermann: „Technologieunabhängige Geräteintegration des OSAmI–Projekts“. *PETRA '10*. 2010 (siehe S. 178).

[Gos08] William Sealy Gosset: „The probable error of a mean“. *Biometrika* (1908), Bd.: S. 1–25 (siehe S. 219).

[Gru05] D. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault und T. Watson: „The Eclipse 3.0 platform: Adopting OSGi technology“. *IBM Systems Journal* (2005), Bd. 44(2): S. 289–299 (siehe S. 23, 61).

[Gru00] V. Gruhn und A. Thiel: *Komponentenmodelle: DCOM, JavaBeans, EnterpriseJavaBeans, CORBA*. Professionelle Softwareentwicklung. Addison Wesley Verlag, 2000 (siehe S. 14, 15).

[Gud07] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar und Yves Lafon: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. World Wide Web Consortium, Recommendation REC-soap12-part1-20070427. Apr. 2007 (siehe S. 34, 40, 43, 182).

[Gut99] E. Guttman: „Service location protocol: automatic discovery of IP network services“. *Internet Computing, IEEE* (Juli 1999), Bd. 3(4): S. 71–80 (siehe S. 58).

[Ham99] S. Hamilton: „Taking Moore's law into the next century“. *Computer* (Jan. 1999), Bd. 32(1): S. 43–48 (siehe S. 27).

[Ham05] Ulrike Hammerschall: *Verteilte Systeme und Anwendungen - Architekturkonzepte, Standards und Middleware-Technologien*. Pearson Education, 2005: S. 1–208 (siehe S. 28, 29).

[Har13] Klaus Hartke: *Observing Resources in CoAP*. Techn. Ber. draft-ietf-core-observe-08.txt. Fremont, CA, USA: IETF Secretariat, 25. Feb. 2013 (siehe S. 39).

[Hei09] A. Hein, M. Eichelberg, O. Nee, A. Schulz, A. Helmer und M. Lipprandt: „A Service Oriented Platform for Health Services and Ambient Assisted Living“. *Advanced Information Networking and Applications Workshops, 2009. WAINA '09. International Conference on*. Mai 2009: S. 531–537 (siehe S. 23).

[Hei05] Lutz J. Heinrich und Franz Lehner: „Informationsmanagement : Planung, Überwachung und Steuerung der Informationsinfrastruktur“. 8., vollst. überarb. u. erg. Aufl. München, Wien: Oldenbourg, 2005. Kap. IT-Governance: S. 63–72 (siehe S. 33).

[Hof01] K. Hofrichter: „The residential gateway as service platform“. *Consumer Electronics, 2001. ICCE. International Conference on*. 2001: S. 304–305 (siehe S. 23).

[Ibr09] Ali Ibrahim, Yang Jiao, Eli Tilevich und WilliamR. Cook: „Remote Batch Invocation for Compositional Object Services“. English. *ECOOP 2009 – Object-Oriented Programming*. Hrsg. von Sophia Drossopoulou. Bd. 5653. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009: S. 595–617 (siehe S. 59).

[Ill06] Stefan Illner, Andre Pohl, Heiko Krumm, Ingo Lück, Andreas Bobek, Hendrik Bohn und Frank Golatowski: „Model-based Management of Embedded Service Systems – An Applied Approach“. *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA 2006)*. Vienna, Austria, 2006: S. 519–523 (siehe S. 185).

[Int05] International Organization for Standardization, Hrsg.: *ISO/IEC 24824-1 Generic Applications of ASN.1: Fast Infoset*. 2005 (siehe S. 46).

[ISO09a] ISO: *International Standard ISO/IEC 10746, Information technology: Open distributed processing - Reference model*. Techn. Ber. International Standard Organization, 2009 (siehe S. 83).

[ISO01] ISO: *International Standard ISO/IEC 9126, Information technology - Product Quality - Part1: Quality Model*. Techn. Ber. International Standard Organization, 2001 (siehe S. 48).

[ISO89] ISO: *ISO/IEC 7498-4: Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework*. Techn. Ber. International Standard Organization, 1989 (siehe S. 50, 186).

[ISO09b] ISO/IEC: *ISO/IEC 15408-2:2009 Information technology – Security techniques – Evaluation criteria for IT security*. Techn. Ber. ISO/IEC 15408-2:2008. Committee JTC1/SC27, 2009 (siehe S. 205).

[ISO02] ISO/IEC: *ISO/IEC 15939: Software Engineering - Software Measurement Process*. International Standard 15939. International Organization for Standardization und International Electrotechnical Commission, 2002 (siehe S. 49).

[Joh91] Brad Curtis Johnson: *A Distributed Computing Environment Framework: An OSF Perspective*. Techn. Ber. DEV-DCE-TP6-1. The Open Software Foundation, Juni 1991 (siehe S. 30).

[Jov05] E. Jovanov: „Wireless Technology and System Integration in Body Area Networks for m-Health Applications“. *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*. Jan. 2005: S. 7158–7160 (siehe S. 10).

[Jun12a] M. Jung, C. Reinisch und W. Kastner: „Integrating Building Automation Systems and IPv6 in the Internet of Things“. *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*. Juli 2012: S. 683–688 (siehe S. 25).

[Jun12b] M. Jung, J. Weidinger, C. Reinisch, W. Kastner, C. Crettaz, A. Olivieri und Y. Bocchi: „A Transparent IPv6 Multi-protocol Gateway to Integrate Building Automation Systems in the Internet of Things“. *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*. Nov. 2012: S. 225–233 (siehe S. 25).

[Kit97] Barbara Kitchenham, Steve Linkman, Alberto Pasquini und Vincenzo Nanni: „The SQUID approach to defining a quality model“. English. *Software Quality Journal* (1997), Bd. 6(3): S. 211–233 (siehe S. 49).

[Kor10] G. Kortuem, F. Kawsar, D. Fitton und V. Sundramoorthy: „Smart objects as building blocks for the Internet of things“. *Internet Computing, IEEE* (Jan. 2010), Bd. 14(1): S. 44–51 (siehe S. 1).

[Kru15] Artur Krutzek und Stefan Rhein: „Evaluation des Entwicklungskomforts des Remote-OSGi-Service-Ansatzes Comoros anhand einer verteilten und mobilen Planungs- und Organisationsanwendung“. Magisterarb. TU Dortmund, 2015 (siehe S. 242).

[Kwo10] Young-Woo Kwon, E. Tilevich und W.R. Cook: „An Assessment of Middleware Platforms for Accessing Remote Services“. *Services Computing (SCC), 2010 IEEE International Conference on*. Juli 2010: S. 482–489 (siehe S. 59).

[Lam07] Klaus Lambertz und Xavier-Noel Cullmann: „Komplexität und Qualität von Software“. *MSCoder* (2007), Bd. (siehe S. 215).

[Lew10] Scott Lewis: *Blog for the Eclipse Communication Framework Project (ECF)*. `http://eclipseecf.blogspot.de/2010/01/osgi-remote-services-from-ecf.html`. Jan. 2010 (siehe S. 72).

[Lie00] Hartmut Liefke und Dan Suciu: „XMill: An Efficient Compressor for XML Data“. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. Dallas, Texas, USA: ACM, 2000: S. 153–164 (siehe S. 45).

[Lin99] Tim Lindholm und Frank Yellin: *Java Virtual Machine Specification*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (siehe S. 106).

[Loc94] Harold W. Lockhart Jr.: *OSF DCE: Guide to Developing Distributed Applications*. New York, NY, USA: McGraw-Hill, Inc., 1994 (siehe S. 30).

[Mar01] D. Marples und P. Kriens: „The Open Services Gateway Initiative: an introductory overview“. *Communications Magazine, IEEE* (Dez. 2001), Bd. 39(12): S. 110–114 (siehe S. 23).

[Mar08] J.M. Marquez, J. Alamo und J.A Ortega: „Distributing OSGi Services: The OSIRIS Domain Connector“. *Networked Computing and Advanced Information Management, 2008. NCM '08. Fourth International Conference on*. Bd. 1. Sep. 2008: S. 341–346 (siehe S. 61).

[Mar99] B. Martin und B. Jano: „WAP Binary XML Content Format“. *W3C NOTE*. Juni 1999 (siehe S. 46).

[Mar09] Jaime Martín, Ralf Seepold, NatividadMartínez Madrid, JuanAntonio Álvarez, Alejandro Fernández-Montes und JuanAntonio Ortega: „A Home E-Health System for Dependent People Based on OSGI“. *Intelligent Technical Systems*. Hrsg. von Natividad Martínez Madrid und RalfE.D. Seepold. Bd. 38. Lecture Notes in Electrical Engineering. Springer Netherlands, 2009: S. 117–130 (siehe S. 23).

[McA05] Jeff McAffer und Jean-Michel Lemieux: *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005 (siehe S. 23).

[McC76] T.J. McCabe: „A Complexity Measure“. *Software Engineering, IEEE Transactions on* (Dez. 1976), Bd. SE-2(4): S. 308–320 (siehe S. 49, 86).

[McC77] J McCall: *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*. Bd. 1-3. ADA049055. General Electric, Nov. 1977 (siehe S. 48, 49, 201, 207, 208).

[McC04] Steve McConnell: *Code Complete: A Practical Handbook of Software Construction*. 2. Aufl. Redmond, WA: Microsoft Press, 2004 (siehe S. 213).

[Med03] Brahim Medjahed, Athman Bouguettaya und Ahmed K. Elmagarmid: „Composing Web Services on the Semantic Web“. *The VLDB Journal* (Nov. 2003), Bd. 12(4): S. 333–351 (siehe S. 33).

[Mel10] Ingo Melzer: „Service-orientierte Architektur“. German. *Service-orientierte Architekturen mit Web Services*. Spektrum Akademischer Verlag, 2010: S. 9–31 (siehe S. 32).

[Moo65] G. E. Moore: „Cramming More Components onto Integrated Circuits“. *Electronics* (19. Apr. 1965), Bd. 38(8): S. 114–117 (siehe S. 27).

[Moo98] G.E. Moore: „Cramming More Components Onto Integrated Circuits“. *Proceedings of the IEEE* (Jan. 1998), Bd. 86(1): S. 82–85 (siehe S. 27).

[Mor12] Guido Moritz: „Web Services in stark ressourcenlimitierten Umgebungen“. Diss. Universität Rostock, 2012 (siehe S. 41–43, 181, 239, 250).

[Mot07] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera und Fabio Casati: „Semi-automated Adaptation of Service Interactions“. *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007: S. 993–1002 (siehe S. 171).

[Mül12] Frerk Müller, Myriam Lipprandt, Marco Eichelberg, Axel Helmer, Clemens Busch, Detlev Willemsen und Andreas Hein: „AAL in Cardiac Rehabilitation“. *Handbook of Ambient Assisted Living - Technology for Healthcare, Rehabilitation and Well-being*. Hrsg. von Juan Carlos Augusto, Michael Huch, Achilles Kameas, Julie Maitland, Paul J. McCullagh, Jean Roberts, Andrew Sixsmith und Reiner Wichert. Bd. 11. Ambient Intelligence and Smart Environments. IOS Press, 2012: S. 512–534 (siehe S. 179).

[Mun02] John C. Munson: *Software Engineering Measurement*. Boca Raton, FL, USA: CRC Press, Inc., 2002 (siehe S. 49).

[New08] Eric Newcomer, David Bosschaert und Tim Diekmann: *RFC 119 - Distributed OSGi*. 2008 (siehe S. 63).

[Nix09a] T. Nixon und A. Regnier: *SOAP-over-UDP Version 1.1*. Techn. Ber. OASIS, Jan. 2009 (siehe S. 133).

[Nix09b] T. Nixon, A. Regnier, D. Discroll und A. Mensch: *Devices Profile for Web Services Version 1.1*. Techn. Ber. OASIS, Juli 2009 (siehe S. 3, 39, 111, 160).

[07] *Nyota Online Documentation*. http://eclipse.compeople.eu/wiki/index.php/Nyota:Main. 2007 (siehe S. 60).

[Oak98] Scott Oaks: *Java security - Java 1.2*. O'Reilly, 1998 (siehe S. 121).

[Obj06] Object Management Group: *CORBA Component Model 4.0 Specification*. Specification Version 4.0. Object Management Group, Apr. 2006 (siehe S. 16).

[Obj95] Object Management Group: *The Common Object Request Broker (CORBA): Architecture and Specification*. Techn. Ber. Object Management Group, 1995 (siehe S. 31).

[Ort13] Ed Ort und Bhakti Mehta: *Java Architecture for XML Binding (JAXB), 2003*. Techn. Ber. Sun Developer Network, 2013 (siehe S. 66).

[OSA11] OSAmI Consortium: *ÖSAmI Device Integration"v1.0*. ITEA2 Project ip0701 Deliverable. 2011 (siehe S. 179).

[Par72] D. L. Parnas: „On the Criteria to Be Used in Decomposing Systems into Modules". *Commun. ACM* (Dez. 1972), Bd. 15(12): S. 1053–1058 (siehe S. 14).

[Pau09] Cesare Pautasso: „{RESTful} Web service composition with {BPEL} for {REST} ". *Data & Knowledge Engineering* (2009), Bd. 68(9). Sixth International Conference on Business Process Management (BPM 2008) – Five selected and extended papers: S. 851–866 (siehe S. 32).

[Pel03] C. Peltz: „Web services orchestration and choreography". *Computer* (Okt. 2003), Bd. 36(10): S. 46–52 (siehe S. 33).

[Pos80] J. Postel: *User Datagram Protocol*. RFC 768. Internet Engineering Task Force, Aug. 1980: S. 3 (siehe S. 112).

[Ray12] A. Rayes, M. Morrow und D. Lake: „Internet of things implications on ICN". *Collaboration Technologies and Systems (CTS), 2012 International Conference on*. Mai 2012: S. 27–33 (siehe S. 2).

[Ree94] Carol A. Reeves und David A. Bednar: „DEFINING QUALITY: ALTERNATIVES AND IMPLICATIONS". *Academy of Management Review* (1994), Bd. 19(3): S. 419–445 (siehe S. 48).

[Rel07a] Jan S. Rellermeyer und Gustavo Alonso: „Concierge: A Service Platform for Resource-constrained Devices". *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007: S. 245–258 (siehe S. 71).

[Rel07b] Jan S. Rellermeyer, Gustavo Alonso und Timothy Roscoe: „Building, Deploying, and Monitoring Distributed Applications with Eclipse and R-OSGI". *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*. eclipse '07. Montreal, Quebec, Canada: ACM, 2007: S. 50–54 (siehe S. 75, 200).

[Rel07c] Jan S. Rellermeyer, Gustavo Alonso und Timothy Roscoe: „R-OSGi: Distributed Applications Through Software Modularization“. *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*. Middleware '07. Newport Beach, California: Springer-Verlag New York, Inc., 2007: S. 1–20 (siehe S. 58, 120).

[Ric07] Leonard Richardson und Sam Ruby: *Restful Web Services*. First. O'Reilly, 2007 (siehe S. 32).

[Rig96] Roger Riggs, Jim Waldo, Ann Wollrath und Krishna Bharat: „Pickling State in the Java System.“ *Computing Systems* (1996), Bd. 9(4): S. 291–312 (siehe S. 103).

[Rob05] George G. Robertson, Mary P. Czerwinski und John E. Churchill: „Visualization of Mappings Between Schemas“. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. Portland, Oregon, USA: ACM, 2005: S. 431–439 (siehe S. 172).

[Ros07] Marcel-Catalin Rosu: „A-SOAP: Adaptive SOAP Message Processing and Compression“. Salt Lake City, UT, USA, Juli 2007: S. 200–207 (siehe S. 46).

[Ruh00] William Ruh, Thomas Herron und Paul Klinker: *IIOP Complete: Understanding CORBA and Middleware Interoperability*. Essex, UK, UK: Addison-Wesley Longman Ltd., 2000 (siehe S. 32).

[Saa03] Alexandre Saad: „Java-based Functionality and Data Management in the automobile—Prototyping at BMW Car IT GmbH“. *JavaSPEKTRUM. SIGS Datacom* (2003), Bd. 2: S. 49–53 (siehe S. 23).

[Sal06] David Salomon: *Data Compression: The Complete Reference*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006 (siehe S. 45).

[Sam06] Yacine Sam, Omar Boucelma und Mohand-Saïd Hacid: „Web Services Customization: A Composition-based Approach“. *Proceedings of the 6th International Conference on Web Engineering*. ICWE '06. Palo Alto, California, USA: ACM, 2006: S. 25–31 (siehe S. 171).

[Sch07] A. Schill und T. Springer: *Verteilte Systeme: Grundlagen und Basistechnologien*. Springer-Verlag Berlin Heidelberg, 2007 (siehe S. 28, 30).

[Sch08] John Schneider und Takuki Kamiya: *Efficient XML Interchange (EXI) Format 1.0*. World Wide Web Consortium, Working Draft WD-exi-20080919. Sep. 2008 (siehe S. 46).

[Sch96] Roy W. Schulte und Yefim V. Natis: *SSA Research Note SPA-401-068, Service Oriented Architectures, Part 1*. Techn. Ber. the Gartner Group, 1996 (siehe S. 32).

[Sch85] Franz Schweiggert: „SOFTWARE-QUALITÄT: Eine Standortbestimmung“. *Köls85* (1985), Bd.: S. 9–30 (siehe S. 201, 247).

[She11] Zach Shelby, Klaus Hartke, Carsten Bormann und Brian Frank: *Constrained Application Protocol (CoAP)*. Techn. Ber. draft-ietf-core-coap-07.txt. Fremont, CA, USA: IETF Secretariat, 8. Juli 2011 (siehe S. 26, 37).

[She06] Limin Shen und Shangping Ren: „Analysis and measurement of software flexibility based on flexible point“. *3th Software Measurement European Forum, Italy*. 2006: S. 331–341 (siehe S. 209).

[She31] W.A. Shewhart: *Economic Control of Quality of Manufactured Product*. Bell Telephone Laboratories series Bd. 509. American Society for Quality Control, 1931 (siehe S. 48).

[Sin12] N.D. Singpurwalla und S.P. Wilson: *Statistical Methods in Software Engineering: Reliability and Risk*. Springer Series in Statistics. Springer London, Limited, 2012 (siehe S. 49).

[Sno89] Richard Snodgrass: *The Interface Description Language: Definition and Use*. New York, NY, USA: Computer Science Press, Inc., 1989 (siehe S. 31).

[Sri95] R. Srinivasan: *XDR: External Data Representation Standard*. RFC Editor. United States, 1995 (siehe S. 36).

[Sta90] James W. Stamos und David K. Gifford: „Remote Evaluation“. *ACM Trans. Program. Lang. Syst.* (Okt. 1990), Bd. 12(4): S. 537–564 (siehe S. 59).

[Sun88] Sun Microsystems: *RPC: Remote Procedure Call Protocol specification: Version 2*. RFC 1057 (Informational). Internet Engineering Task Force, Juni 1988 (siehe S. 29).

[Svo85] Liba Svobodova: „Client/Server Model of Distributed Processing“. English. *Kommunikation in Verteilten Systemen I*. Hrsg. von Dirk Heger, Gerhard Krüger, Otto Spaniol und Werner Zorn. Bd. 95. Informatik-Fachberichte. Springer Berlin Heidelberg, 1985: S. 485–498 (siehe S. 30).

[Szy02] Clemens Szyperski: *Component Software: Beyond Object-Oriented Programming*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002 (siehe S. 14).

[Tan06] Andrew S. Tanenbaum und Maarten van Steen: *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006 (siehe S. 29, 86, 93).

[Ter08] Terracotta Inc.: *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Springer Science & Business, 2008 (siehe S. 62).

[The09] The Apache Software Foundation: *Apache CXF - An Open Source Service Framework*. `http://cxf.apache.org/`. 2009 (siehe S. 66).

[The10a] The Apache Software Foundation: *ZooKeeper*. `http://www.osgi.org/Specifications`. 2010 (siehe S. 66).

[The04] The OSGi Alliance: *Listener Pattern Considered Harmful: The "Whiteboard"Pattern*. Technical Whitepaper. OSGi, 2004 (siehe S. 71).

[The12a] The OSGi Alliance: *OSGi Compendium Release 5*. http://www.osgi.org/Specifications. 2012 (siehe S. 22, 25, 129, 177, 186, 191–193).

[The12b] The OSGi Alliance: *OSGi Core Release 5*. http://www.osgi.org/Specifications. 2012 (siehe S. 3, 17, 18, 21, 22, 127).

[The10b] The OSGi Alliance: *OSGi Enterprise Release 4.2*. http://www.osgi.org/Specifications. 2010 (siehe S. 65, 103, 110).

[Tho97] Dean Thompson, Chris Exton, Leah Garrett, A.S.M. Sajeev und Damien Watkins: *Distributed Component Object Model (DCOM)*. 1997 (siehe S. 16).

[Tho04] Henry S. Thompson, David Beech, Murray Maloney und Noah Mendelsohn: *XML Schema Part 1: Structures Second Edition*. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028. Okt. 2004 (siehe S. 37).

[Tom04] Vojtěch Toman: „Syntactical compression of XML data“. *Presented at 16th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'04*. 2004 (siehe S. 46).

[Tre14] Andreas Trenkmann: „Grafische Komposition verteilter Software aus OSGi-Clients und DPWS-Services mit Generierung der Stub- und Marshalling-Funktionen“. Magisterarb. TU Dortmund, 2014 (siehe S. 167).

[Vin97] S. Vinoski: „CORBA: integrating diverse applications within distributed heterogeneous environments“. *Communications Magazine, IEEE* (Feb. 1997), Bd. 35(2): S. 46–55 (siehe S. 32).

[Wal09] Craig Walls: *Modular Java: Creating Flexible Applications with Osgi and Spring*. 1st. Pragmatic Bookshelf, 2009 (siehe S. 57).

[Wal05] Joe Walnes, Mauro Talevi, Jason van Zyl, Nat Pryce und Dan North: *XStream*. http://xstream.codehaus.org/. Apr. 2005 (siehe S. 103).

[09] *Web-Based Enterprise Management (WBEM)*. http://www.dmtf.org/standards/wbem. Distributed Management Task Force. 2009 (siehe S. 50).

[Wei91] Mark Weiser: „The Computer for the 21st Century“. *Scientific American* (Jan. 1991), Bd. 265(3): S. 66–75 (siehe S. 2).

[Wer06a] Christian Werner: „Optimierte Protokolle für Web Services mit Begrenzten Datenraten“. *Ausgezeichnete Informatikdissertationen 2006*. Hrsg. von Dorothea Wagner. Bd. D-7. LNI. GI, 2006: S. 209–218 (siehe S. 181).

[Wer06b] Christian Werner, Carsten Buschmann, Ylva Brandt und Stefan Fischer: „Compressing SOAP Messages by using Pushdown Automata“. *2006 IEEE International Conference on Web Services (ICWS 2006), 18-22 September 2006, Chicago, Illinois, USA*. IEEE Computer Society, 2006: S. 19–28 (siehe S. 181).

[Wie92] Gio Wiederhold, Peter Wegner und Stefano Ceri: „Toward Megaprogramming“. *Commun. ACM* (Nov. 1992), Bd. 35(11): S. 89–99 (siehe S. 15).

[Wie94] René Wies: „Policies in Network and System Management – Formal Definition and Architecture“. *Journal of Network and System Management* (1994), Bd. 2(1): S. 63–83 (siehe S. 51).

[Wil95] John Wiley: *Object Management Architecture Guide: Revision 3.0*. Hrsg. von Richard Mark Soley. Third Edition. New York: Object Management Group, 1995 (siehe S. 31).

[Bea09] John Beatty, Gopal Kakivaya, Devon Kemp, Thomas Kuehnel, Brad Lovering, Bryan Roe, Christopher St. John, Guillaume Simonnet, Doug Walter, Jack Weast, Yevgeniy Yarmosh und Prasad Yendluri: *Web Services Dynamic Discovery (WS-Discovery) Version 1.1*. OASIS Standard 1 July 2009. Organization for the Advancement of Structured Information Standards, 1.2009 (siehe S. 112, 196).

[06] *WSDM 1.1 OASIS Standard Specifications*. OASIS Web Services Distributed Management (WSDM) TC. 2006 (siehe S. 51).

[Wue11] Mathias Wuebbeling: „Komfortable Webservice-basierte Lösung zum OSGi-Standard "Remote Services"mit erweitertem dynamischen Marshalling“. Magisterarb. TU Dortmund, 2011 (siehe S. 141).

[Zee10] E. Zeeb, G. Moritz, W. Thronicke, M. Lipprandt, A. Hein, F. Müller, J. Krüger, O. Dohndorf, A. Litvina, C. Fiehe, I. Lück, F. Golatowski und D. Timmermann: „Generic platform for advanced E-health applications“. *e-Health Networking Applications and Services (Healthcom), 2010 12th IEEE International Conference on*. Juli 2010: S. 201–208 (siehe S. 23).

Abbildungsverzeichnis

Publikationen

Beiträge auf internationalen Konferenzen

1. Christoph Fiehe, Anna Litvina, Ingo Lück, **Oliver Dohndorf**, Jens Kattwinkel, Franz-Josef Stewing, Jan Krüger, Heiko Krumm „Location-Transparent Integration of Distributed OSGi Frameworks and Web Services“ *IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA 2009), Workshop Service Oriented Architectures in Converging Networked Environments (SOCNE 2009)*, Bradford, Großbritannien, 2009
2. Christoph Fiehe, Anna Litvina, Ingo Lück, Franz-Josef Stewing, **Oliver Dohndorf**, Jan Krüger and Heiko Krumm „Towards the Web of Things: Using DPWS to Bridge Isolated OSGi Platforms“ *IEEE 8th International Conference on Pervasive Computing and Communications (PerCom 2010), Workshop Web of Things (WoT 2010)*, Mannheim, Deutschland, 2010
3. **Oliver Dohndorf**, Jan Krüger, Heiko Krumm, Christoph Fiehe, Anna Litvina, Ingo Lück, Franz-Josef Stewing „Lightweight Policy-Based Management of Quality-Assured, Device-Based Service Systems“ *IEEE 24th International Conference on Advanced Information Networking and Applications (AINA 2010), Workshop Service Oriented Architectures in Converging Networked Environments (SOCNE 2010)*, Perth, Australien, 2010
4. Elmar Zeeb, Guido Moritz, Wolfgang Thronicke, Myriam Lipprandt, Andreas Heinz, Frerk Müller, Jan Krüger, **Oliver Dohndorf**, Anna Litvina, Christoph Fiehe, Ingo Lück, Frank Golatowski, Dirk Timmermann „Generic Platform for Advanced E-Health Applications“ *IEEE 12th International Conference on e-Health Networking, Application & Services (Healthcom2010)*, Lyon, Frankreich, 2010
5. **Oliver Dohndorf**, Jan Krüger, Heiko Krumm, Christoph Fiehe, Anna Litvina, Ingo Lück, F.-J. Stewing „Policy-Based Management for Resource-Constrained Devices and Systems“ *11th IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2010)*, Fairfax, USA, 2010
6. **Oliver Dohndorf**, Jan Krüger, Heiko Krumm, Christoph Fiehe, Anna Litvina, Ingo Lück, Franz-Josef Stewing „Tool-Supported Refinement of High-Level Requirements and Constraints into Low-Level Policies“ *12th IEEE In-*

ternational Symposium on Policies for Distributed Systems and Networks (POLICY 2011), Pisa, Italien, 2011

7. Naci Dai, Jesus Bermejo, Felix Cuadrado Latasa, Alejandra Ruiz López, Isaac Agudo, Elmar Zeeb, Jan Krüger, Wolfgang Thronicke, **Oliver Dohndorf**, Christoph Fiehe, Anna Litvina „OSAMI COMMONS - An Open Platform for Dynamic Services for Ambient Intelligence“ *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2011), Workshop Service Oriented Architectures in Converging Networked Environments (SOCNE 2011)*, Toulouse, Frankreich, 2011
8. **Oliver Dohndorf**, Jan Krüger, Heiko Krumm, Christoph Fiehe, Anna Litvina, Ingo Lück, Franz-Josef Stewing „Adaptive and Reliable Binding in Ambient Service Systems“ *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2011), Workshop Service Oriented Architectures in Converging Networked Environments (SOCNE 2011)*, Toulouse, Frankreich, 2011
9. Israfil Akman, Rafael Bielen, Henning Brümmer, Egor Kudrjaschow, Stefan Tschentscher, **Oliver Dohndorf**, Heiko Krumm, Jan-Dirk Hoffmann, Anke Workowski, Detlev Willemsen „CordiAAL: Enhanced Motivation for Cardiological Ergometer Training through Virtual Groups in Virtual Worlds“ *7th international Conference on Health Informatics (HEALTHINF 2014)* Angers, Frankreich, 2014
10. Michael Pantförder, Hasan Simsek, Patrick Baron, Theodor Schnitzler, Krystian Kensy, **Oliver Dohndorf**, Heiko Krumm, Jan-Dirk Hoffmann, Anke Workowski, Detlev Willemsen „Gamification in endurance sports - Possible use in coronary secondary prevention to increase motivation and compliance“ *European Congress on e-Cardiology & e-Health* Bern, Schweiz, 2014

Beiträge auf nationalen Konferenzen

1. Christoph Fiehe, Anna Litvina, Ingo Lück, Franz-Josef Stewing, **Oliver Dohndorf**, Jan Krüger, Heiko Krumm „Policy-gesteuertes Management adaptiver und gütegesicherter Dienstesysteme im Projekt OSAMI“ *Informatik 2009 Im Focus das Leben: Jahrestagung der Gesellschaft für Informatik, Workshop Ambient Assisted Living (AAL)*, Lübeck, Deutschland, 2009
2. **Oliver Dohndorf**, Andre Göring, Heiko Krumm, Andre Schneider, Aike Sommer, Stephan Sladek, Clemens Busch, Jan-Dirk Hoffmann, Detlev Willemsen „RehaWeb - An Information System for Cardiologic Rehabilitation Assistance in the Third Phase“ *5. AAL-Kongress 2012* Berlin, Deutschland, 2012
3. Tim Janus, Torben Kohlmeier, Viktor Marinov, Janina Marks, Christian Mikosch, Michael Nimbs, Thorsten Panke,Jörn Störling, **Oliver Dohndorf**, Heiko Krumm, Jan-Dirk Hoffmann, Anke Workowski, Detlev Willemsen „GlobalSensing: A Supervised Outdoor-Training in Cardiological Secondary Prevention“ *6. AAL-Kongress 2013* Berlin, Deutschland, 2013

Betreute Arbeiten

1. Romuald Rasel „Distributed OSGi unter Anwendung des Standards OASIS WS-DD zur Webservice-gestützten Interaktion verteilter Geräte und Dienste“ *2010*
2. Jens Krause „Policies für das automatisierte Management von gerätebasierten Serviceensembles und ihren Assoziationen am Beispiel einer medizinischen Applikation“ *2010*
3. Daniel Saltmann „Dynamische Service-Anbindung zur Unterstützung zuverlässiger vernetzter Geräte und Anwendungen“ *2010*
4. Matthias Wübbeling „Komfortable Webservice-basierte Lösung zum OSGi-Standard „Remote Services“ mit erweitertem dynamischen Marshalling“ *2011*
5. Aike Jan Sommer „Rollenbasierte Autorisierung in einem kombiniertem Community-Portal und AAL-System zur kardiologischen Rehabilitation“ *2012*
6. Oskar Opalinski, Andre Schneider „Eine verteilte, REST-basierte Android Applikation zur mobilen Supervisionsunterstützung von Wandergruppen für die kardiovaskuläre Rehabilitation“ *2012*
7. Stephan Sladek „Integration eines kardiologischen AAL-Systems mit klinischer Vitalparameter-Überwachung in ein Web2.0-Portal eines Social-Networks für Rehabilitationspatienten“ *2013*
8. Nadine Hodroj, Semra Top „Usability und Barrierefreiheit eines Web-2.0-Social-Network-Portals zur kardiologischen Rehabilitation mit Nutzerevaluierung“ *2013*
9. Kamen Avramov „Resource-efficient communication for OSGi Remote Services through extension of the Comoros/JMEDS-platform by SOAP-over-COAP“ *2013*
10. Andreas Trenkmann „Grafische Komposition verteilter Software aus OSGi-Clients und DPWS-Services mit Generierung der Stub- und Marshalling-Funktionen“ *2014*
11. Viktor Marinov „Effizienz und Leistungsfähigkeit DPWS- und COAP-basierter Implementierungen des OSGi-Remote-Service-Standards in praktischen Anwendungsszenarien mobiler und eingebetteter Systeme“ *2014*