

Studiengang: Softwaretechnik
Prüfer: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer: Dipl. inf. Gregor Schiele
Beginn: 01. August 2003
Ende: 31. Januar 2004
CR-Nummer: C.2.4, C.5.3

Diplomarbeit Nr. 2130

**Entwicklung eines
Middleware-Teilsystems zur
Unterstützung
nichtfunktionaler Parameter in
ubiquitären Rechnersystemen**

Alexander Rau

Institut für Parallele und Verteilte
Systeme
Abteilung Verteilte Systeme
Universitätsstr. 38
70569 Stuttgart

Zusammenfassung

Im Zuge der Weiterentwicklung wird das alltägliche Leben mehr und mehr von kleinen und tragbaren Rechnern geprägt. Bei steigender Leistung und sinkenden Preisen ist eine zukünftige Allgegenwärtigkeit dieser elektronischen Geräte absehbar. Die Entwicklung verteilter Anwendungen für ubiquitäre Rechnernetze ist im Vergleich zu klassischen Rechnernetzen mit erheblichem Mehraufwand verbunden, da auf Grund der starken Heterogenität und der unzuverlässigen Kommunikation in ubiquitären Rechnersystemen zusätzliche Probleme behandelt werden müssen. Eine neuartige und bereits existierende Verteilungsinfrastruktur BASE der Abteilung Verteile Systeme adressiert diese Probleme und realisiert einen flexiblen und ressourceneffizienten Mechanismus zur Kommunikation zwischen beliebigen Rechereinheiten.

Eine weitere Entwicklung der Abteilung VS ist ein Komponentensystem namens PCOM. Dieses erlaubt auf der Basis von BASE die einfache Entwicklung verteilter Anwendungen durch Kapselung einzelner Funktionalitäten in Form einzelner Komponenten. Sofern die Schnittstellenbeschreibungen der Komponenten identisch sind, können diese beliebig zur Laufzeit untereinander ausgetauscht werden. Neben der erhöhten Flexibilität wird die Wiederverwendbarkeit existierender Komponenten dadurch erheblich verbessert. Allerdings hat sich herausgestellt, dass in einigen Anwendungsszenarien neben den Komponentenschnittstellen, auch funktionalen Eigenschaften genannt, unter anderem nichtfunktionale Eigenschaften eine wichtige Rolle spielen, die etwas über die Qualität der erbrachten Funktionalität aus Sicht des Komponentennutzers aussagen. Damit letztere wirkungsvoll realisiert werden können, ist eine geeignete Unterstützung dieser auch in BASE erforderlich. Das Thema dieser Arbeit stellt deshalb eine Erweiterung der Verteilungsinfrastruktur BASE zur Unterstützung beliebiger nichtfunktionaler Eigenschaften dar.

Meinen Eltern

zum Dank für ihre langjährige und liebevolle Unterstützung

Inhaltsverzeichnis

1	Einführung	5
1.1	Ubiquitous Computing	5
1.2	Konventionelle Verteilungsinfrastrukturen	7
1.3	Die Verteilungsinfrastruktur BASE	8
1.4	Das Komponentensystem PCOM	9
1.5	Nichtfunktionalität	10
1.6	Ziel dieser Arbeit	11
1.7	Überblick	11
2	Anforderungen und Vorgaben	13
2.1	Vorgaben	13
2.2	Anforderungen	13
3	Analyse verwandter Arbeiten	15
3.1	Nichtfunktionale Domänen	15
3.1.1	Domänen-spezifische Arbeiten	15
3.1.2	Domänen-übergreifende Arbeiten	17
3.2	Ende-zu-Ende Betrachtungen	18
3.2.1	Verteilungsinfrastrukturebene	18
3.2.2	Anwendungsebene	20
3.3	Architektonische Ansätze	20
3.3.1	Applikations-basierter Ansatz	21
3.3.2	Verteilungsinfrastruktur-basierter Ansatz	23
3.3.3	Kooperativer Ansatz	24
3.4	Technische Ansätze	24
3.4.1	Reservierungen	24
3.4.2	Adaption	26
4	Analyse	27
4.1	Flexibilisierung der Plugin-Architektur	29

4.1.1	Originäre Plugin–Architektur	29
4.1.2	Nachteile der originären Plugin–Architektur	31
4.1.3	Restrukturierung	32
4.1.4	Konzeptionelle Integration der Plugin–Auswahl	36
5	Formalisierung	39
5.1	Nichtfunktionale Informationen	39
5.1.1	Formulierung von Bedürfnisgraden	42
5.1.2	Wertzuweisungen	44
5.1.3	Lokalität	45
5.2	Nichtfunktionale Anforderungen	45
5.2.1	Adaptive Erweiterungen	47
6	Erfüllung von Anforderungen	51
6.1	Selektionsstrategien	51
6.1.1	Interoperabilität	51
6.1.2	Nichtfunktionale Teilangebote	52
6.1.3	Veränderlichkeit nichtfunktionaler Teilangebote	52
6.1.4	Erhebung nichtfunktionaler Teilangebote	54
6.1.5	Feststellung nichtfunktionaler Angebote	56
6.1.6	Erfüllbarkeit nichtfunktionaler Anforderungen	58
6.1.7	Interoperable Erfüllbarkeit	58
6.2	Kooperierende Selektionsstrategien	59
6.2.1	Äquivalenz zwischen Selektionsstrategien	60
6.2.2	Kooperation	61
6.2.3	Aushandlung nichtfunktionaler Angebote	63
7	Entwurf	67
7.1	Restrukturierte Plugin–Architektur	67
7.1.1	Nichtfunktionale Beschreibungen	74
8	Evaluation	76
8.1	Proof of Concept	76
8.2	Performanz	78
8.3	Nutzwert	86
9	Ausblick und zukünftige Erweiterungen	88
9.1	Fazit	88
9.2	Zukünftige Erweiterungen	89

Kapitel 1

Einführung

Diese Diplomarbeit entstand im Rahmen des Projekts *Peer to Peer Pervasive Computing* [29] an der Abteilung für *Verteilte und Parallele Systeme* der Universität Stuttgart.

1.1 Ubiquitous Computing

How computers will be used will be determined in part by technology trends, and in part by trends in the needs of people for computation, and by changes in living and activities.

Mark Weiser [36]

Verteilte Systeme und speziell die Formen der Rechnernetzung unterliegen der Gesetzmäßigkeit, wie sie Mark Weiser, der Urvater des *Ubiquitous Computing*, schon vor einigen Jahren erkannt hat. Im Vergleich zu den klassischen Rechnernetzen wie *Wide-* und *Local Area Networks*, die bereits vor Jahrzehnten einen Meilenstein darstellten, erleben neuerdings gerade *Adhoc-Rechnernetze* einen enormen Aufschwung. Neben der hohen Nachfrage sind die stetig steigende Leistung heutzutage verfügbarer Rechnersysteme, die Miniaturisierung existierender Technologien, die drahtlose Kommunikation und fallende Preise die wesentlichen Faktoren, die diesen Boom unterstützen. Bereits heute begleiten und erleichtern unzählige tragbare Rechnersysteme das alltägliche Leben. Ein Ende dieses Trends ist bisher nicht in Sicht.

Friedemann Mattern greift diese technologische Entwicklung auf [21] und geht vor allem auf die tiefgreifende Informatisierung und Vernetzung von Rechnersystemen ein. Die Vision des Ubiquitous Computing (kurz: UC) umfasst im Wesentlichen die Durchdringung der Umwelt mit unzähligen Rechereinheiten, die zur Bewältigung von Aufgaben untereinander Informationen austauschen

und kooperieren. Mark Weiser hält es bereits seit Anfang der 90er Jahre für erforderlich, dass aus Sicht des Benutzers die Bedienung und Verwendung solcher Systeme möglichst einheitlich und vereinfacht präsentiert wird und im optimalen Fall für diesen vollkommen unsichtbar arbeiten [37]. Hierzu hat sich Weiser folgendermaßen geäußert:

The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it [38].

Nach Meinung Matterns' sind die Konsequenzen des Ubiquitous Computing nur schwer abzuschätzen. Allerdings vermutet Mattern, dass diese sich sehr stark in wirtschaftlicher und sozialer Hinsicht bemerkbar machen werden.

Die Realisierung des Ubiquitous Computing ist schwierig. Abgesehen von der bisher nur ansatzweise zu erkennenden Vereinfachung von Benutzungsschnittstellen müssen künftig weitere Probleme behandelt werden. Diese sind auf unterschiedlichen Ebenen angesiedelt [18]:

- Benutzerebene
- Dienstebene
- Netzwerkebene

Auf Benutzerebene besteht die Gefahr, dass Anwender auf Grund der Vielzahl unterschiedlicher Rechnersysteme mit unzähligen anwendungsspezifischen Benutzungsschnittstellen konfrontiert und überfordert werden, was die Aufgabenerfüllung eines Benutzers in den Hintergrund treten lässt. Vielmehr wird der Computer, eigentlich als Hilfsmittel und Werkzeug und zur Unterstützung für die tägliche Arbeit gedacht, in den Vordergrund gestellt.

Auf Dienstebene besteht das Problem, dass Rechneinheiten anwendungsspezifische Schnittstellen anbieten, die dem Kommunikationspartner potenziell nicht zur Entwicklungszeit und erst recht nicht zur Laufzeit bekannt sind, weshalb die Interaktion zwischen beiden nicht ohne weiteres möglich ist. Man spricht in diesem Zusammenhang von einer *Open World Assumption* [32]. Die sogenannte *Spontane Vernetzung* behandelt diese Probleme, indem Dienst und Dienstanutzer jeweils Beschreibungen ihrer selbst bereitstellen. Auf diese Weise können sich diese mittels *Selbstkonfiguration* an die jeweiligen Schnittstellen und Verhaltensweisen des Interaktionspartner anpassen.

Auf Netzwerkebene sind die auftretenden Probleme technischer Natur. Die eingesetzten Rechnersysteme in ubiquitären Szenarien weisen eine hohe Heterogenität auf, da jedes Rechnersystem über bestimmte Übertragungstechniken

und Protokolle verfügt, die auf anderen Rechnersystemen unter Umständen nicht unterstützt werden. Diese Tatsache erschwert die Realisierung der Kommunikation und Interaktion zwischen diesen. Unabhängig davon führt die Mobilität von Rechnersystemen zu instabiler Kommunikation, zwangsläufig verursacht durch die begrenzte Reichweite drahtloser Technologien. Somit muss stets damit gerechnet werden, dass Rechnereinheiten über kurz oder lang nicht (mehr) verfügbar sind. Die Entwicklung zuverlässiger verteilter Anwendungen in Adhoc-Netzwerken erweist sich aus diesen Gründen als verhältnismäßig aufwendig und schwierig. Kindberg [16] folgert, dass zur Entwicklung ubiquitärer Anwendungen entsprechende Invarianten bekannt sein müssen, um der Dynamik in Adhoc-Netzen begegnen zu können.

You should design ubicomp systems on the assumption that the set of participating users, hardware and software is highly dynamic and unpredictable. Clear invariants that govern the system's execution should exist.

1.2 Konventionelle Verteilungsinfrastrukturen

Die auftretenden Probleme auf Netzwerkebene wurden in klassischen, heterogenen Rechnernetzen in der Vergangenheit durch sogenannte *Verteilungsinfrastrukturen*¹ behandelt. Sie stellen eine Abstraktionsschicht zwischen Betriebssystem und Anwendung dar. Dabei werden betriebssystemabhängige Kommunikationsmechanismen auf plattformunabhängige Kommunikationsmechanismen abgebildet und somit die Portabilität verteilter Anwendungen gewährleistet. Die erforderliche Interoperabilität zwischen heterogenen Rechnersystemen wird mit Hilfe standardisierter Interoperabilitätsprotokolle hergestellt, die einheitliche Nachrichtenformate festlegen. Allerdings wurde das Problem der Interoperabilität bisher nicht vollständig aus der Welt geschafft. Vielmehr existieren inzwischen eine ganze Menge an unterschiedlichen Implementierungen dieser Verteilungsinfrastrukturen, die teilweise ihre eigenen Interoperabilitätsprotokolle mit sich bringen. Derzeit stellt CORBA [24] zwar einen Quasi-Standard dar, aber faktisch wurden gerade im wissenschaftlichen Umfeld neue Verteilungsplattformen konzipiert. Dies liegt maßgeblich daran, dass der Einsatz der bereits existierenden Verteilungsinfrastrukturen in neuen Anwendungsszenarien, wie sie beispielsweise im Ubiquitous Computing vorkommen, tatsächlich nicht angemessen ist [1, 2].

¹auch Verteilungsplattform oder Middleware genannt

1.3 Die Verteilungsinfrastruktur Base

Stark heterogene, mobile Rechnernetze erfordern innerhalb der Verteilungsinfrastruktur Mechanismen, die die flexible Kommunikation über mehrere Protokolle und Übertragungstechniken erlauben. Auf diese Weise kann zum einen die Heterogenität verschiedener Rechnersysteme behandelt werden, zum anderen kann die Kommunikation zwischen Rechnersystemen bei gleichzeitiger Unterstützung mehrerer Protokolle oder Übertragungstechniken je nach Anforderungen angepasst werden. Beispielsweise ist die Kommunikation über Infrarot eventuell unerwünscht, wenn diese auch über die schnellere Variante *Wireless LAN* durchgeführt werden kann.

Ein weiteres Problem stellt die unzuverlässige Verfügbarkeit stabiler Infrastrukturen dar. Die Kommunikation zwischen beliebigen Teilnehmern muss auf dezentraler Basis stattfinden, da zentrale Dienste wie beispielsweise *Namens-* und *Verzeichnisdienste* in der bisherigen Form auf Grund mangelnder Verfügbarkeit nicht mehr eingesetzt werden können. Vielmehr sind dezentral organisierte Varianten solcher Dienste erforderlich, deren Nutzung auch während eines Kommunikationsabbruchs möglich sind.

Weiterhin handelt es sich bei einem Großteil der betrachteten Rechnersysteme in mobilen Rechnernetzen um tragbare und somit sehr kleine Geräte. Folglich sind die verfügbaren Ressourcen der jeweiligen Rechnersysteme mehr oder weniger begrenzt. Dies führt dazu, dass eine für Ubiquitous Computing geeignete Verteilungsinfrastruktur auf geringen Ressourcenverbrauch ausgelegt sein muss.

Diese Anforderungen wurden durch die Entwicklung einer neuen Verteilungsinfrastruktur mit Namen BASE [1, 2, 12] behandelt. Neuartig ist der Verzicht auf zentrale Dienste, die durch lokale, auf jedem Rechner verfügbare Instanzen ersetzt wurden. Dementsprechend wurde der Vorgang der zentralen Registrierung lokal verfügbarer Dienste auf eine dezentrale Variante mit Hilfe von Multi- und Broadcasts umgestellt.

Ein angemessener Kommunikationsmechanismus wurde mit Hilfe einer Architektur realisiert, die es ermöglicht, sogenannte *Plugins* zu laden und zu verwenden. Ein Plugin realisiert in diesem Zusammenhang jeweils die Kommunikation über ein bestimmtes Protokoll und Übertragungsmedium. Im Ganzen kann auf diese Weise je nach Bedarf und aktuellen Umgebungsbedingungen während der Kommunikation zwischen Plugins „umgeschaltet“ werden. Sogar bei komplexeren Interaktionsmustern ist zwischen Aufruf und Antwort der Wechsel auf alternative Plugins möglich.

1.4 Das Komponentensystem PCom

In einer weiteren Diplomarbeit innerhalb des CANU Projekts [13] wurde ein Komponentensystem entwickelt, das die in ubiquitären Anwendungsszenarien auftretenden Probleme auf Dienstebene behandelt und die Realisierung verteilter Anwendungen für ubiquitäre Verteilungsinfrastrukturen vereinfacht. PCom stellt somit eine Realisierung der Spontanen Vernetzung dar.

Komponenten stellen Programmeinheiten dar, die eine klar definierte Funktionalität realisieren und unter anderem in Hinblick auf Wiederverwendung konzipiert worden sind. Im Vergleich zu den Diensten einer Verteilungsinfrastruktur sind sie deshalb wesentlich stärker gekapselt, wodurch die Wiederverwendbarkeit garantiert wird. Auf Grund dieses Ansatzes wird die Anwendungsentwicklung dahingehend verändert, dass im Vergleich zur konventionellen Anwendungsentwicklung wesentlich weniger Aufwand betrieben werden muss, um eine vollständige, verteilte Anwendung zu realisieren. Komponenten müssen für die Erstellung einer neuen Anwendung lediglich sinnvoll miteinander verknüpft werden.

Ein wesentlicher Vorteil, den PCom bietet, sind die in einer Komponente jeweils enthaltenen Komponentenbeschreibungen. Eine solche Beschreibung erfasst einerseits die angebotene Funktionalität einer Komponente und zum anderen die Anforderungen, die zur Verwendung einer Komponente erfüllt sein müssen. Diese Beschreibungen werden als *Kontrakte* bezeichnet. Mit Hilfe von Kontrakten besteht die Möglichkeit, dass Abhängigkeiten zwischen Komponenten automatisch erkannt und vom Komponentensystem aufgelöst werden. Abhängigkeiten zwischen Komponenten werden hierzu nicht explizit über einen Namen oder die Identität einer Komponente bestimmt, sondern vielmehr anhand der Funktionalität einer Komponente. Dadurch können Komponenten zur Laufzeit ersatzweise durch neue Komponenten ausgetauscht werden, die zwar eine andere Implementierung beinhalten, aber laut Kontrakt die gleiche Funktionalität bieten. Eine explizite und statische Verknüpfung von Komponenten, wie es in konventionellen verteilten Anwendungen bisher der Fall war, ist nicht mehr erforderlich.

Im weiteren Verlauf dieser Arbeit werden an Stelle allgemeiner Dienste nur noch PCom-Komponenten betrachtet. Prinzipiell wäre auch der Bezug auf (BASE)-Dienste möglich. Sowohl Dienste als auch Komponenten stellen Implementierungen mit klar definierter Schnittstelle dar. Allerdings handelt es sich bei PCom-Komponenten aus den bereits aufgeführten Gründen um die komfortablere Alternative.

1.5 Nichtfunktionalität

Neben der Funktionalität einer Komponente spielen bei der Auswahl und Nutzung von Komponenten häufig auch nichtfunktionale Eigenschaften eine wichtige Rolle. Sie beschreiben im Wesentlichen das Laufzeitverhalten von Komponenten und die Art und Weise, wie Komponenten gegenüber ihren Nutzern bestimmte Dienste erbringen können [3]. Je nach Anwendungsfall konzentrieren sich nichtfunktionale Eigenschaften auf einzelne Aspekte während der Kommunikation zwischen Komponenten. Beispielsweise erfordern sicherheitskritische Anwendungen unter anderem, dass Komponenten eine hohe Verfügbarkeit aufweisen und die Funktionalität dieser auch in problematischen Situationen ausreichend realisiert und genutzt werden kann. Echtzeitkritische Anwendungen wiederum erheben zum Beispiel Ansprüche an das Kommunikationsverhalten in Bezug auf minimale Verzögerungen und ausreichend Datendurchsatz.

Da die Verteilungsinfrastruktur konzeptionell einen Kommunikationsbus für Komponenten und Komponentennutzer darstellt, spielt das realisierte Kommunikationsverhalten der Verteilungsinfrastruktur im Rahmen einer verteilten Anwendung eine wichtige Rolle. Die Verteilungsinfrastruktur ist mitverantwortlich für die „Qualität“ der Kommunikation und kann unter Umständen das Laufzeitverhalten einer verteilten Anwendung insgesamt negativ beeinflussen. Die Ursache hierfür ist im Vergleich zu klassischen Anwendungen der erheblich höhere Aufwand bei der Kommunikation, da dieser in den meisten Fällen ² nicht im gleichen Adressraum stattfinden kann und auf Grund dessen Zeit und Ressourcen für den Versand von Nachrichten investiert werden müssen. Dieser Aufwand wird in erster Linie durch die notwendigen Verarbeitungsschritte im Kommunikationsteil einer Verteilungsplattform hervorgerufen, der die Interoperabilität zwischen Rechnersystemen gewährleistet. Im Normalfall werden eingehende Aufrufe einer Komponente in ausgehende Nachrichten und eingehende Nachrichten in Aufrufe einer lokalen Komponente transformiert, stets in Hinsicht auf die beidseitig verfügbaren Protokolle. Weitere Verzögerungen bei der Auslieferung von Nachrichten können entstehen, wenn Engpässe oder Übertragungsfehler auf Netzwerkebene entstehen. Können Nachrichten nicht zugestellt werden, weil der entfernte Kommunikationspartner nicht mehr erreichbar ist, ist der getätigte Aufruf einer Anwendung nicht durchführbar. Das Laufzeitverhalten der Anwendung wird in beiden Fällen stark beeinflusst. Aus diesem Grund ist offensichtlich das Laufzeitverhalten der Verteilungsinfrastruktur

²Eine Ausnahmen stellt beispielsweise die Optimierung eines lokalen Aufrufes dar. In diesem Fall kann die Nachricht direkt an das Zielobjekt weitergeleitet werden ohne den Protokollstack zu durchlaufen.

tur im Vergleich zu konventionellen Anwendungen in einer ganzheitlichen Betrachtung nicht zu vernachlässigen.

Problematisch ist die Tatsache, dass das Kommunikationsverhalten der Verteilungsplattform oftmals nicht transparent ist. Ohne zusätzliche Mechanismen sind Komponenten zur Laufzeit weder in der Lage, das derzeitige Kommunikationsverhalten zu erfahren und auf Veränderungen im Laufzeitverhalten der Verteilungsplattform zu reagieren, noch Anforderungen bezüglich des Kommunikationsverhaltens zu stellen. Infolgedessen müssen sich Komponenten zur Laufzeit darauf verlassen, dass ausreichend Ressourcen zur Verfügung stehen.

1.6 Ziel dieser Arbeit

Auf Grund dessen, dass verteilte Anwendungen unter Umständen Ansprüche an das Laufzeitverhalten einer Verteilungsinfrastruktur stellen, ist eine Unterstützung nichtfunktionaler Eigenschaften innerhalb der Verteilungsinfrastruktur in vielen Fällen unabdingbar. Die Unterstützung beliebiger nichtfunktionaler Eigenschaften innerhalb von BASE ist deshalb das Ziel dieser Diplomarbeit. Eine Erweiterung für BASE soll die Integration anwendungsspezifischer Strategien ermöglichen, die im Einzelnen nichtfunktionale Eigenschaften für bestimmte Anwendungsgebiete unterstützen.

1.7 Überblick

In Kapitel 2 werden die Anforderungen und Vorgaben beschrieben, die an die in dieser Arbeit zu entwickelnde Erweiterung der Verteilungsinfrastruktur gestellt werden. Kapitel 3 behandelt existierende Arbeiten und Verteilungsinfrastrukturen, die nichtfunktionale Eigenschaften unterstützen. Den Betrachtungen folgend werden die Unterschiede nichtfunktionaler Eigenschaften analysiert und die Konzepte der existierenden Arbeiten zur Unterstützung nichtfunktionaler Eigenschaften vorgestellt. In Kapitel 4 werden anhand der bekannten Anforderungen und der erarbeiteten Erkenntnisse aus Kapitel 3 die erforderlichen Punkte zur Entwicklung der geplanten Erweiterung hergeleitet. Kapitel 5 diskutiert die Umsetzung nichtfunktionaler Beschreibungen, die zur Spezifikation gewünschter Kommunikationsverhalten benötigt werden. Dabei wird neben der eigentlichen Formalisierung nichtfunktionaler Anforderungen der Aspekt der Adaption näher beleuchtet. Kapitel 6 geht auf die Erfüllbarkeit und Einhaltung nichtfunktionaler Anforderungen während der Kommunikation zwischen Komponenten ein. Kapitel 7 beschreibt den Entwurf der Erweiterung. Zur Bewertung der Erweiterung werden in Kapitel 8 die Ergebnisse der Evaluation

präsentiert und diskutiert. Kapitel 9 rundet diese Arbeit mit einem Fazit und Ausblick in die Zukunft ab.

Kapitel 2

Anforderungen und Vorgaben

In diesem Kapitel werden die Anforderungen und Vorgaben beschrieben, die hinsichtlich der Erweiterung gestellt werden.

2.1 Vorgaben

Die folgenden Vorgaben werden in der Aufgabenbeschreibung explizit formuliert.

Plattform und Technologien

Die Unterstützung nichtfunktionaler Eigenschaften soll als Erweiterung der Verteilungsinfrastruktur BASE realisiert werden. Die Portabilität von BASE wird durch den Einsatz der plattform-unabhängigen Entwicklungssprache *Java* erreicht. Die vorgesehene Erweiterung von BASE um eine Unterstützung nichtfunktionaler Eigenschaften soll deshalb aus Konformität zur existierenden Verteilungsinfrastruktur ebenfalls mit Hilfe der *Java 2 Microedition* [35] realisiert werden. Diese ist speziell für ressourcenarme Rechnersysteme ausgelegt.

2.2 Anforderungen

Die im Folgenden aufgeführten Anforderungen wurden während der Problemanalyse festgelegt:

Generizität und Flexibilität

Die Erweiterung sollte generisch sein und auf flexible Art und Weise beliebige nichtfunktionale Eigenschaften unterstützen. Dies ist aus folgenden Gründen

sinnvoll: Erstens ist eine gleichzeitige Unterstützung aller möglichen nichtfunktionalen Eigenschaften kaum zu realisieren. Zweitens werden zukünftig weitere nichtfunktionale Eigenschaften relevant sein, die derzeit nicht bekannt sind oder eine untergeordnete Rolle spielen. Drittens gestatten ressourcenarme Rechereinheiten, die die eigentliche Zielgruppe von BASE darstellen, keine aufwendigen Implementierungen in Bezug auf Speicher- und Prozessorauslastung. Somit ist es wesentlich sinnvoller, eine generische Erweiterung zu entwickeln, die die Integration nichtfunktionaler Eigenschaften je nach Anwendungsszenario und auch für zukünftig relevante nichtfunktionale Eigenschaften ermöglicht.

Einfache Handhabung

Da komplexe Mechanismen tendenziell zu einem hohen Aufwand bei der Einarbeitung führen und unrentabel für einfache Anwendungsfälle sind, sollte die Unterstützung nichtfunktionaler Eigenschaften Komponentenentwicklern möglichst einfache Schnittstellen und Mechanismen anbieten, die von der Umsetzung nichtfunktionaler Eigenschaften innerhalb der Verteilungsinfrastruktur abstrahieren. Darüber hinaus sollten klar definierte Eintrittspunkte in der Verteilungsinfrastruktur bereitgestellt werden.

Minimalität

Viele der in Betracht kommenden Rechnersysteme verfügen lediglich über relativ schwach ausgeprägte Ressourcen. Damit der Betrieb der Verteilungsinfrastruktur und der geplanten Erweiterung wie bisher auch auf diesen Rechnersystemen möglich ist, liegt der Schwerpunkt auf einer ressourcenschonenden Realisierung der Erweiterung. In dieser Hinsicht ist neben einer entsprechend geeigneten Implementierung vor allem die Auswahl einfacher, aber effektiver Konzepte erforderlich.

Portabilität

Bereits bei der Entwicklung von BASE war die Portabilität der Verteilungsinfrastruktur eine ausdrückliche Anforderung. Heterogene Rechnersysteme erfordern zwangsläufig eine plattform-unabhängige Entwicklung verteilter Anwendungen, da andernfalls mit erheblichem Mehraufwand für die manuelle Portierung und Anpassung von Anwendungen auf einzelne Plattformen gerechnet werden muss. Dementsprechend muss auch die Erweiterung zur Unterstützung nichtfunktionaler Eigenschaften entsprechend portabel sein.

Kapitel 3

Analyse verwandter Arbeiten

Im Rahmen der vorgesehenen Erweiterung der Verteilungsinfrastruktur BASE werden im weiteren Verlauf dieses Kapitels existierende Arbeiten betrachtet, die nichtfunktionale Eigenschaften in irgendeiner Form unterstützen. Dazu werden einige bekannte Systeme vorgestellt und genauer differenziert. Im Anschluß daran werden existierende Konzepte zur Unterstützung nichtfunktionaler Eigenschaften genauer betrachtet und diskutiert.

3.1 Nichtfunktionale Domänen

Einige der sowohl in der Literatur beschriebenen Arbeiten als auch der sich im praktischen Einsatz befindlichen Systeme konzentrieren sich hinsichtlich Nichtfunktionalität auf einzelne, spezielle Anwendungsszenarien, auch *nichtfunktionale Domänen* genannt. Andere wiederum bieten eine generische Unterstützung für nichtfunktionale Eigenschaften und sind in beliebigen Domänen einsetzbar. In diesem Abschnitt werden beide Varianten genauer betrachtet.

3.1.1 Domänen-spezifische Arbeiten

Systeme, die jeweils nur eine einzige nichtfunktionale Domäne unterstützen, werden als *domänen-spezifisch* bezeichnet. Einige der wichtigsten dieser Art werden nun kurz präsentiert. Es wird dabei kein Anspruch auf Vollständigkeit erhoben. Vielmehr sollen die folgenden Beispiele einen groben Überblick geben, welche nichtfunktionalen Domänen derzeit eine gewisse Bedeutung haben.

Echtzeit

Die *Object Management Group*, bekannt für den weit verbreiteten *Corba*-Standard, stellt für diesen einige erweiterte Spezifikationen bereit, die im Einzelnen

unterschiedliche nichtfunktionale Domänen behandeln. Unter anderem zählt hierzu *Realtime-Corba* [27], das die nichtfunktionale Domäne der Echtzeit adressiert. Die echtzeit-kritische Ausführung von Anwendungen impliziert, dass Informationen in einem vorher bestimmten Zeitrahmen erfolgreich verarbeitet werden müssen. Im Kontext verteilter Systeme konzentrieren sich diese Systeme auf die Auslieferung von Nachrichten unter Einhaltung bestimmter zeitkritischer Eigenschaften. Eine typische Eigenschaft dieser Domäne ist zum Beispiel die *Latenz*. Sie beschreibt die Zeitdauer vom Zeitpunkt der Aktivierung bis hin zur erfolgreichen Auslieferung eines Aufrufes. Eine weit bekannte und praxis-taugliche Implementierung der Realtime-Corba-Spezifikation stellt *The Ace ORB* [17] dar.

Sicherheit

In verteilten Systemen spielt bekanntermaßen Sicherheit im Sinne von Geheimhaltung, Authentizität und Integrität eine nicht zu vernachlässigende Rolle. Verteilungsinfrastrukturen setzen beispielsweise Zugriffskontrollen für Objekte beziehungsweise Dienste ein, um die Nutzung dieser auf bestimmte Benutzergruppen einzuschränken oder Verschlüsselungsmechanismen, um die geheime Übertragung von Nachrichten zwischen Instanzen einer Verteilungsinfrastruktur zu ermöglichen. Auch zu dieser Domäne stellt die OMG eine Erweiterung der Corba-Spezifikation bereit [25]. Bekannte Verteilungsinfrastrukturen wie .NET (Remoting) [7] und J2EE [34] behandeln zumindest teilweise diese Domäne innerhalb einer Verteilungsinfrastruktur [6].

Zuverlässigkeit und Verfügbarkeit

Eine dritte hier genannte Erweiterung der Corba-Spezifikation adressiert die Zuverlässigkeit verteilter Anwendungen [26]. Diese kann beispielsweise durch Objektreplikation, Multicastgruppen und Recoveryprotokolle realisiert werden. Implementiert wurde diese Spezifikation zum Beispiel in Electra [33].

Energieeffizienz

Die Domäne Energieeffizienz stellt gerade für mobile und tragbare Rechnersysteme einen wichtigen Bereich der Nichtfunktionalität dar. Da mobilen Rechereinheiten in den meisten Fällen nur eine begrenzte Energieversorgung zur Verfügung steht, müssen verteilte Anwendungen auf niedrigen Energieverbrauch optimiert sein, um eine möglichst lange Betriebsdauer gewährleisten zu können. Mohapatra et. al behandeln in [22] diese nichtfunktionale Domäne, indem eine

Verteilungsinfrastruktur entworfen wurde, die das Ziel verfolgt, den tatsächlichen Energieverbrauch verteilter Anwendungen zu erfassen und diesen durch selektive Rekonfiguration einzelner Teile zu optimieren.

3.1.2 Domänen–übergreifende Arbeiten

Die zuvor exemplarisch aufgezeigten Domänen behandeln jeweils unabhängige Anwendungsbereiche nichtfunktionaler Eigenschaften. Ein wesentlicher Nachteil domänen–spezifischer Verteilungsinfrastrukturen ist, dass die gleichzeitige Unterstützung mehrerer nichtfunktionaler Domänen nicht angeboten wird. Für Anwendungsszenarien, die eine gleichzeitige Unterstützung mehrerer nichtfunktionaler Domänen voraussetzen, ist der Einsatz domänen–spezifischer Verteilungsinfrastrukturen praktisch ungeeignet.

Diesen Nachteil beheben zu einem gewissen Grad Verteilungsinfrastrukturen mit domänen–übergreifender Unterstützung [20, 4, 10]. Sie ermöglichen prinzipiell die Integration eigener Verfahren zur Unterstützung nichtfunktionaler Eigenschaften je nach Anwendungsszenario. Insofern sollte in diesem Zusammenhang nicht damit gerechnet werden, dass diese Systeme „out of the box“ beliebige nichtfunktionale Domänen von vornherein unterstützen. Dies ist praktisch nicht möglich, da eine endgültige und allumfassende Lösung für mehrere Domänen sehr komplex, aufwendig und nur schwer zu realisieren ist. Unterschiedliche Domänen wie beispielsweise Echtzeit und Energieeffizienz können sogar unter Umständen zu widersprüchlichen Vorgehensweisen bei der Unterstützung nichtfunktionaler Eigenschaften führen. Gerade auf ressourcenarmen Rechereinheiten lassen sich diese kaum vereinigen. Darüber hinaus ist auf Grund der Generizität domänen–übergreifender Systeme die Effizienz maßgeschneiderter domänen–spezifischer Verteilungsinfrastrukturen mit Sicherheit nicht zu erreichen.

Folglich stellen domänen–übergreifende Verteilungsinfrastrukturen lediglich generische Mittel zu Verfügung, die die Integration eigener Verfahren zur Unterstützung bestimmter Domänen in die Verteilungsinfrastruktur ermöglichen. Diese generischen Mittel bestehen in der Regel aus einer allgemein anwendbaren Beschreibungssprache, mit deren Hilfe nichtfunktionale Anforderungen explizit formuliert werden können, und einem oder mehreren klar definierten Einstiegspunkten innerhalb der Verteilungsinfrastruktur, die die Integration eigener Verfahren erlaubt.

Je nach Grad der erforderlichen Unterstützung kann dies natürlich einen enormen Aufwand für den jeweiligen Anwendungsentwickler bedeuten. Zum einen müssen nichtfunktionale Anforderungen mit Hilfe der bereitgestellten

Spezifikationsprache formuliert werden, was im Vergleich dazu in domänen-spezifischen Systemen normalerweise direkt mit Hilfe proprietärer Schnittstellen möglich ist. Zum anderen muss auf Grund einer „nur“ generischen Unterstützung der Code zur Realisierung und Erfüllung nichtfunktionaler Anforderungen erst implementiert und integriert werden. Die Verwendung generischer Systeme bedeutet deshalb stets einen zusätzlichen Aufwand für Anwendungsentwickler, auch wenn damit gerechnet werden kann, dass langfristig existierende und etablierte Implementierungen angepasst und wiederverwendet werden können.

3.2 Ende-zu-Ende Betrachtungen

Unabhängig davon, zu welcher Domäne nichtfunktionale Eigenschaften gehören und ob diese durch eine spezifische oder generische Lösung unterstützt werden, existieren grundlegende Unterschiede zwischen einzelnen nichtfunktionalen Eigenschaften (siehe z.B. [3, 31, 23]). Im Folgenden werden diese Unterschiede genauer betrachtet. In diesem Zusammenhang spielt vor allem der sogenannte *Kommunikationspfad* eine wichtige Rolle, der in Abbildung 3.1 dargestellt wird. Der Kommunikationspfad repräsentiert diejenigen Objekte der verteilten Anwendung, die für die Übertragung von Aufrufen zwischen Komponenten im Einzelnen verantwortlich sind. Da diese Objekte prinzipiell sequenziell durchlaufen werden, entsteht eine Art Kette. Die Endpunkte dieser Kette bilden die jeweiligen miteinander kommunizierenden Komponenten. Zwischen den Endpunkten des Kommunikationspfades verarbeiten primär Objekte der Verteilungsinfrastruktur die transferierten Aufrufe bzw. Nachrichten. Die Reihenfolge, in der die Objekte miteinander verknüpft sind, hängt im Wesentlichen von den erforderlichen Verarbeitungsschritten ab. Eine Instanz des Kommunikationspfades besitzt immer ein Pendant auf der gegenüberliegenden Seite, das dieselbe Funktionalität bereitstellt. Insofern existieren entsprechend der betrachteten Funktionalität der Objekte mehrere unterschiedliche Verarbeitungsschichten.

3.2.1 Verteilungsinfrastrukturebene

Die Verteilungsinfrastruktur stellt im Wesentlichen einen Kommunikationskanal zwischen Komponenten dar. Gerade nichtfunktionale Eigenschaften wie beispielsweise *Energiebedarf*, *Latenz* und *Zuverlässigkeit*, die unter anderem vom Kommunikationsverhalten abhängen, erfordern unbedingt eine Unterstützung durch die Verteilungsinfrastruktur. Solche nichtfunktionale Eigenschaften werden zukünftig auf Grund der Indirektion zwischen Komponenten über die Ver-

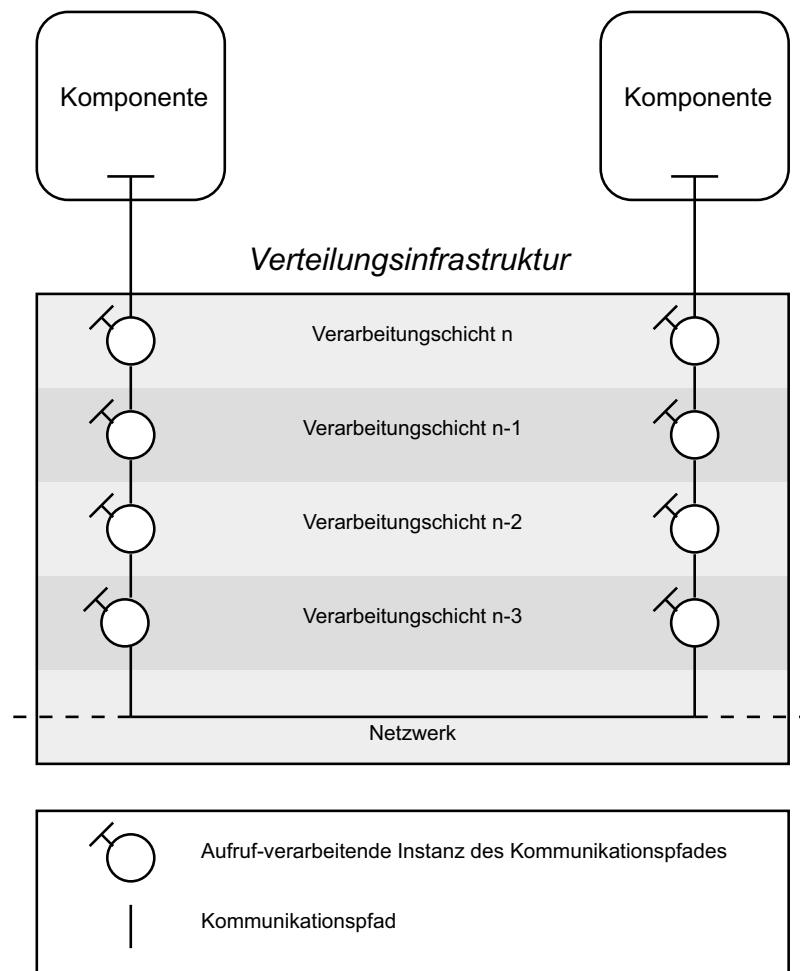


Abbildung 3.1: Die Kommunikationskette zwischen Komponenten

teilungsinfrastruktur als *indirekte nichtfunktionale* Eigenschaften bezeichnet.

Eine genauere Betrachtung indirekter nichtfunktionaler Eigenschaften zeigt in Zusammenhang mit dem Kommunikationspfad einen weiteren Unterschied. In einigen Fällen ist die Realisierung einer nichtfunktionalen Eigenschaft ausschließlich durch Objekte des Kommunikationspfades in einer einzigen Verarbeitungsschicht realisierbar. Folglich bleiben Objekte aus anderen Verarbeitungsschichten bei der weiteren Verarbeitung von Aufrufen und Nachrichten in Bezug auf diese Eigenschaft unberührt. Eine solche nichtfunktionale Eigenschaft stellt beispielsweise die Eigenschaft *Abhörsicherheit* dar. Sie nimmt lediglich die Verarbeitungsschicht in Anspruch, die zur Verschlüsselung von Nachrichten vorgesehen ist. Eigenschaften dieser Art werden kurz als *horizontale* Eigenschaften bezeichnet.

Vertikale, nichtfunktionale Eigenschaften zeichnen sich dadurch aus, dass

sie ganz im Gegensatz zu den horizontalen Eigenschaften grundlegend nur dann umgesetzt werden können, wenn jedes verarbeitende Objekt des Kommunikationspfades diese Eigenschaft unterstützt. Im Vergleich zu den horizontalen Eigenschaften müssen vertikale Eigenschaften über sämtliche Schichten hinweg realisiert werden, was letztlich bedeutet, dass vertikale Eigenschaften zwangsläufig immer indirekte Eigenschaften sind. Zu diesen Eigenschaften gehören beispielsweise *Energieverbrauch* und *Latenz*. Auf Grund der Tatsache, dass Komponenten die Endpunkte des Kommunikationspfades darstellen, sind natürlich auch sie für die Umsetzung dieser nichtfunktionalen Eigenschaften mitverantwortlich.

3.2.2 Anwendungsebene

Viele nichtfunktionale Eigenschaften können ohne Unterstützung der Verteilungsinfrastruktur realisiert werden. Dies ist genau dann der Fall, wenn die jeweiligen Eigenschaften keine Beziehung zu den verarbeitenden Objekten des Kommunikationspfades innerhalb der Verteilungsinfrastruktur aufweisen. Genaugenommen sind lediglich die Endpunkte des Kommunikationspfades, die Komponenten, für ihre Umsetzung verantwortlich. Da auch Komponenten eine gesonderte Verarbeitungsschicht der Kommunikationskette darstellen und die Unterstützung direkt zwischen Komponenten realisiert werden kann, werden diese Eigenschaften als *direkte, horizontale* Eigenschaften bezeichnet. Beispiele solcher Eigenschaften sind Versionierungsinformationen von Komponenten, Sprachunterstützung und viele weitere anwendungs-spezifische nichtfunktionale Eigenschaften.

3.3 Architektonische Ansätze

In der Literatur existieren unterschiedliche Ansätze zur Unterstützung nicht-funktionaler Eigenschaften. Die Ansätze unterscheiden sich darin, bis zu welchem Grad jeweils Anwendungen und Verteilungsinfrastruktur an der Unterstützung nichtfunktionaler Eigenschaften beteiligt sind. Dies hat natürlich Auswirkungen darauf, welche Typen nichtfunktionaler Eigenschaften unterstützt werden können. Im Folgenden werden die verschiedenen Ansätze näher beleuchtet und die Einschränkungen in Bezug auf die Unterstützung der klassifizierten nichtfunktionalen Eigenschaften geprüft.

3.3.1 Applikations-basierter Ansatz

Im applikations-basierten Ansatz nimmt die Anwendung die zentrale Rolle bezüglich der Unterstützung nichtfunktionaler Eigenschaften ein. Die Realisierung nichtfunktionaler Eigenschaften geschieht einzig und allein in Eigenverantwortung innerhalb der Anwendung. Hierzu wird die Unterstützung bestimmter nichtfunktionaler Eigenschaften direkt auf Anwendungsebene implementiert. Die Verteilungsinfrastruktur bleibt davon unberührt.

Dass keine Veränderungen der zugrundeliegenden Verteilungsinfrastruktur erforderlich sind, ist zwar von Vorteil. Auf diesem Wege können zumindest in Hinsicht auf die Unterstützung nichtfunktionaler Anforderungen beliebige Verteilungsinfrastrukturen eingesetzt werden, da letztere nicht zur Unterstützung der Eigenschaften beitragen müssen. Dieser Ansatz weist jedoch einen erheblichen Nachteil auf. Indirekte vertikale Eigenschaften können in diesem Ansatz nicht realisiert werden. Diese Eigenschaften erfordern, wie zuvor beschrieben, eine Unterstützung durch jedes verarbeitende Objekt des Kommunikationspfades. Beispielsweise ist die maximale Verzögerung einer Nachricht maßgeblich davon abhängig, wieviel Zeit jedes Objekt für die Verarbeitung einer Nachricht bzw. eines Aufrufes benötigt. Da die Verteilungsinfrastruktur einen Großteil des Kommunikationspfades darstellt und Anwendungen in diesem Ansatz auf Grund der fehlenden Unterstützung innerhalb der Verteilungsinfrastruktur explizit keine nichtfunktionalen Anforderungen spezifizieren können, können indirekte nichtfunktionale Anforderungen auch auf Umwegen nicht auf Anwendungsebene unterstützt werden.

Das Rover Toolkit [15] realisiert direkt horizontale Eigenschaften wie Konsistenz und Zuverlässigkeit beispielsweise mit Hilfe von *Relocatable Dynamic Objects* und *Queued RPCs*. Dabei handelt es sich jeweils um proxy-ähnliche Objekte, die zwischen Anwendung und Verteilungsinfrastruktur die jeweiligen Eigenschaften realisieren. Ebenso folgen die *Power Aware Reconfigurable Middleware* [22] und Implementierungen der *Fault-tolerant Corba* Spezifikation, die bereits in Abschnitt 3.1.1 kurz erwähnt wurden, dem applikations-basierten Ansatz.

PCOM bietet ebenfalls eine rudimentäre Unterstützung horizontaler Eigenschaften auf Komponentenebene. In Kapitel 1.4 wurde bereits erläutert, dass Komponenten grundlegend durch Komponentenbeschreibungen definiert werden. Eine solche Komponentenbeschreibung umfasst sowohl Informationen über die unterstützten Eigenschaften einer Komponente als auch über die zu erfüllenden Eigenschaften durch andere Komponenten. Neben den funktionalen Eigenschaften, der Schnittstelle einer Komponente, können auch nicht-

```

<contract>
  <offer></offer>
  <requirement>
    <component name=''input'' proxy=''pcom.example.InputProxy''>
      <interface type=''pcom.example.InputService''/>
      <language compare=''equals''>english</language>
    </component>
  </requirement>
  ...
</contract>

```

Abbildung 3.2: Anforderungsbeschreibung einer Komponente

```

<contract>
  <offer>
    <interface type=''pcom.example.InputService''/>
    <language>english</language>
  </offer>
  <implementation>
    ...
  </implementation>
</contract>

```

Abbildung 3.3: Angebotsbeschreibung einer Komponente

funktionale Eigenschaften spezifiziert werden. Diese werden zusammen mit den funktionalen Eigenschaften in Form eines Vertrages zusammengefasst. An einem Beispiel soll dieser Sachverhalt verdeutlicht werden. Abbildung 3.2 zeigt den Anforderungsteil eines exemplarischen Vertrages. Dieser Anforderungsteil ermöglicht die Suche nach Komponenten, die neben den geforderten Schnittstellen die nichtfunktionale Eigenschaft *Englisch als Sprache* unterstützen. Das Gegenstück zum Anforderungsteil stellt der Angebotsteil des Vertrages dar, den Abbildung 3.3 zeigt.

PCOM ermöglicht den automatischen Vergleich von Verträgen beziehungsweise der in diesen enthaltenen Angebots- und Anforderungsteilen. Benötigt eine Komponente eine weitere Komponente, die einen bestimmten Vertrag erfüllt, kann PCOM entsprechende Komponenten finden und Referenzen auf diese zurückliefern. Allerdings werden bisher nur direkte Eigenschaften unterstützt, da die Verteilungsinfrastruktur BASE derzeit keine Unterstützung für nichtfunktionale Eigenschaften anbietet.

3.3.2 Verteilungsinfrastruktur-basierter Ansatz

Im Vergleich zum applikations-basierten Ansatz werden im verteilungsinfrastruktur-basierten Ansatz nichtfunktionale Eigenschaften ausschließlich auf Ebene der Verteilungsinfrastruktur betrachtet. Der Vorteil dieses Ansatzes ist, dass Anwendungsentwickler keine formalen Beschreibungen ihrer nichtfunktionalen Anforderungen spezifizieren und der Verteilungsinfrastruktur zur Verfügung stellen müssen. Allerdings stellt sich die Frage, ob ohne explizite Formulierungen die Verteilungsinfrastruktur in der Lage ist, die nichtfunktionalen Anforderungen einer Anwendung zu kennen. Offensichtlich müssen diese anderweitig festgestellt werden. Dies ist nur dann möglich, wenn die nichtfunktionalen Anforderungen mehrerer Anwendungen identisch und a priori bekannt sind. Sofern diese Voraussetzungen erfüllt sind, können diese informalen Anforderungen innerhalb der Verteilungsinfrastruktur realisiert werden, indem die Verteilungsinfrastruktur die Aktivitäten der laufenden Anwendungen analysiert und anhand der gewonnen Informationen erkennt, ob diese eingehalten werden oder ob eventuelle Veränderungen im Laufzeitverhalten der Verteilungsinfrastruktur erforderlich sind. Der Nachteil dieses Ansatzes ist offensichtlich. Spezielle nichtfunktionale Anforderungen einzelner Anwendungen werden in diesem Ansatz nicht unterstützt.

Beispiele, die zwar keine Verteilungsinfrastrukturen im engeren Sinne, aber eine Abstraktionsschicht zwischen Anwendungen und Betriebssystem darstellen und anwendungs-übergreifend gewisse Nichtfunktionalitäten realisieren, sind das *Mobile Application Framework* [14] und *NFS*. *NFS* steht für *Network File System* und ist ein verteiltes Dateisystem, das gegenüber Anwendern eine gewisse Verteilungstransparenz realisiert. In diesem Zusammenhang sind die nichtfunktionalen Anforderungen der Anwender in Bezug auf die Transparenz stets dieselben: die eingebundenen Dateisysteme müssen langfristig konsistent sein und eine hohe Verfügbarkeit garantieren. Deshalb werden keine ausdrücklich formulierten Anforderungen benötigt.

Das *Mobile Application Framework* funktioniert auf ähnliche Weise. Hier werden sogenannte *Call Handler* eingesetzt, die anwendungs-übergreifend konfiguriert werden und im Einzelnen eine bestimmte Nichtfunktionalität wie beispielsweise *File Logging* zur Konsistenzerhaltung (ähnlich *NFS*) oder *Scheduling* von ein- und ausgehenden Nachrichten realisieren, um die Dauer und Häufigkeit geöffneter Verbindungen zu minimieren.

3.3.3 Kooperativer Ansatz

Der kooperative Ansatz vermeidet die Nachteile der beiden zuvor beschriebenen Ansätze. Anwendungen spezifizieren explizit nichtfunktionale Anforderungen und die Verteilungsinfrastruktur ist für die Realisierung dieser verantwortlich. Beispiele hierfür sind in [17, 4, 23, 10, 11] zu finden. Der kooperative Ansatz ermöglicht sowohl direkte und indirekte, als auch horizontale und vertikale nichtfunktionale Eigenschaften, da sowohl auf Applikations-Ebene als auch auf Ebene der Verteilungsinfrastruktur nichtfunktionale Eigenschaften unterstützt werden. Darüber hinaus sind die nichtfunktionalen Anforderungen jeder einzelnen Anwendung formal verfügbar, weshalb der Verteilungsinfrastruktur im Vergleich zum rein verteilungsinfrastruktur-basierten Ansatz die zu realisierenden Anforderungen der Anwendungen bekannt sind.

Ein Nachteil muss bei diesem Ansatz in Kauf genommen werden. Durch die Kooperation zwischen Verteilungsinfrastruktur und Anwendungen entsteht aus software-technischer Sicht eine höhere Kopplung zwischen diesen. Deshalb muss auf eine möglichst schlanke Schnittstelle zwischen beiden Schichten geachtet werden, um eine möglichst geringe Kopplung zu erreichen.

3.4 Technische Ansätze

Verteilungsinfrastrukturen benötigen zwangsläufig für die Verarbeitung von Aufrufen und Nachrichten in gewissem Umfang Ressourcen. Da Ressourcen wie beispielsweise Prozessoren, Speicher und Netzwerkkarten unter Umständen nur einen exklusiven Zugriff erlauben, knapp bemessen und nicht in beliebigem Umfang zu jedem Zeitpunkt zur Verfügung stehen, kann die Funktionalität der Verteilungsinfrastruktur zur Laufzeit mehr oder weniger eingeschränkt und die Qualität gegenüber Komponenten herabgesetzt werden. Sofern Komponenten nichtfunktionale Anforderungen besitzen, kann ohne weiteres nicht garantiert werden, dass diese von der Verteilungsinfrastruktur erfüllt werden können. Es existieren zwei grundsätzlich verschiedene Ansätze, mit nur eingeschränkt verfügbaren Ressourcen umzugehen: *reservierungsbasierte* Systeme und *adaptive* Systeme. Diese werden im Folgenden kurz beschrieben.

3.4.1 Reservierungen

Reservierungssysteme verwalten exklusiv die verfügbaren Ressourcen einer oder mehrerer Rechereinheiten unter Zuhilfenahme eines zentralen Ressourcenverwalters. Der Zugriff auf Ressourcen durch Verteilungsinfrastruktur und Komponenten ist nur möglich, indem Reservierungsanfragen an den Ressourcen-

verwalter gestellt werden und dieser explizit eine Ressource für den jeweiligen Interessenten freigibt. In konventionellen Netzwerkstrukturen kann so durch angemessene Verteilungsverfahren (Scheduling) eine effiziente Auslastung der verfügbaren Ressourcen erreicht werden. Nutzer von Ressourcen können, sofern sie tatsächlich den Zugriff auf die erforderlichen Ressourcen erhalten haben, ihre Funktionalität zumindest hinsichtlich der zur Verfügung stehenden Ressourcen garantiert realisieren.

In Adhoc-Netzen ist dieser Ansatz allerdings nur bedingt brauchbar. Der hier relevante Unterschied zwischen Adhoc-Netzen und klassischen Netzen ist der, dass in letzteren die erfolgreiche Kommunikation unter Verwendung lokaler Netzwerkressourcen in hohem Maße garantiert werden kann. In Adhoc-Netzen ist dies jedoch nicht möglich, da gerade drahtlose Verbindungen eine vergleichsweise hohe Unzuverlässigkeit aufweisen und eine erfolgreiche Kommunikation trotz Reservierung von Netzwerkressourcen nicht immer möglich ist. Darüber hinaus ist die Verwaltung entfernter Ressourcen in Adhoc-Netzen sehr fehleranfällig, da sich mit einer hohen Wahrscheinlichkeit manche Rechneinheiten auf Grund der Mobilität aus dem Verwaltungsbereich entfernen und somit vorhandene Reservierungen hinfällig werden. Die automatische Erkennung und Entfernung solcher Reservierungen ist praktisch nur durch periodisches Überwachen aller Beteiligten möglich und deshalb sehr aufwendig. Der Einsatz reservierungsbasierter Systeme in Adhoc-Netzen ist kein Garant für die erfolgreiche Ausführung von Anwendungen, sofern von diesen nicht-lokale Ressourcen entfernter Rechneinheiten benötigt werden.

Die Verwaltung von Ressourcen über mehrere Rechner hinweg impliziert, dass die zu verwaltenden Rechner kooperativ sind und sich nicht gegenseitig absichtlich oder unabsichtlich behindern. Kooperative Ansätze in Adhoc-Netzwerken sind jedoch zu hinterfragen. In klassischen Netzen können die beteiligten Rechneinheiten meist relativ einfach identifiziert und als vertrauenswürdig betrachtet werden. Adhoc-Netze bereiten in dieser Hinsicht jedoch Schwierigkeiten, da die Menge der beteiligten Rechneinheiten potenziell unbekannt ist. Das kann dazu führen, dass bestimmte Ressourcen Teilnehmern nicht zur Verfügung stehen, weil sie von anderen Teilnehmern über kurz oder lang blockiert werden. Infolgedessen können unerwünschte Ressourcenengpässe im Reservierungssystem auftreten. Ein Ansatz zur Vermeidung dieses Problems ist, dass der Zugriff auf Ressourcen nur dann erlaubt wird, wenn der Anfragende eine Gegenleistung erbringt, beispielsweise in Form von monetären Mitteln. Das kostenpflichtige Anbieten von Ressourcen ist allerdings im Ubiquitous Computing nicht praktikabel, da, wie bereits erwähnt, die Identifikation aller

Teilnehmer eines Adhoc-Netztes kaum möglich ist.

3.4.2 Adaption

Adaptive Ansätze behandeln diese Probleme grundlegend anders. Im Gegensatz zu reservierungsbasierten Ansätzen bieten sie von vornherein keine Garantien, dass für die Verarbeitung von Aufrufen und Nachrichten ausreichend Ressourcen zur Verfügung stehen. Vielmehr wird die Verarbeitung von Aufrufen einfach durchgeführt, in der Hoffnung, dass ausreichend Ressourcen zur Verfügung stehen. Im Vergleich zu reservierungsbasierten Systemen führt dies je nach Auslastung einzelner Rechneinheiten und Erreichbarkeit anderer Teilnehmer zu erheblichen Fehlschlägen bei der Durchführung von Aufrufen. Infolgedessen ist die Idee adaptiver Systeme die, dass nach einem Fehlschlag explizit Anwendungen über diesen Sachverhalt informiert werden, damit diese ihre Anforderungen herabsetzen und unter Umständen in einem degradierten Modus fortfahren. Auf diese Weise können Anwendungen weniger Ressourcen in Anspruch nehmen und somit die Wahrscheinlichkeit erfolgreicher Aufrufe erhöht werden. Allerdings führt Adaption in diesem Fall oftmals zu einer reduzierten Qualität der erbrachten Funktionalität einer Anwendung. Häufig wird jedoch der degradierte Betrieb einer Anwendung der Terminierung dieser vorgezogen.

Da adaptive Systeme auf Grund ihrer Instabilität und Unzuverlässigkeit keine Garantien bieten können, sind diese zumindest für sicherheitskritische Anwendungen keine brauchbare Lösung. Für den Betrieb anderer Anwendungen stellt Adaption jedoch unter Umständen eine brauchbare Alternative zu reservierungsbasierten Systemen dar, vorallem auf Grund der Tatsache, dass letztere im Vergleich zu klassischen Netzen in Adhoc-Netzen nicht für entfernte Ressourcen Garantien bieten können. Sofern nur die Verfügbarkeit lokaler Ressourcen relevant ist, können beide Ansätze durchaus miteinander kombiniert werden.

Kapitel 4

Analyse

Die Betrachtung verwandter Arbeiten hat gezeigt, dass zum einen verschiedene Domänen und zum anderen unterschiedliche Ansätze zur Unterstützung nichtfunktionaler Eigenschaften existieren. Zur Unterstützung nichtfunktionaler Eigenschaften werden einerseits domänen-spezifische Lösungen angeboten, deren Einsatz sich jeweils auf eine einzelne Domäne beschränkt und deshalb Anwendungsentwicklern maßgeschneiderte Mechanismen und Schnittstellen zur Verfügung stellt. Andererseits existieren auch domänen-übergreifende Lösungen, die diesen Vorteil nicht aufweisen, stattdessen aber mehr oder weniger für beliebige Domänen aufbereitet werden können.

Aus den Anforderungen geht klar hervor, dass das Ziel der Erweiterung eine domänen-übergreifende Unterstützung nichtfunktionaler Eigenschaften ist. Deshalb dienen die bisher betrachteten domänen-spezifischen Systeme nunmehr als Überblick über die derzeit aktuellen Domänen nichtfunktionaler Eigenschaften. PCOM stellt bereits geeignete Mittel zur Unterstützung direkter nichtfunktionaler Eigenschaften auf Anwendungsebene bereit. Aus diesem Grund steht die Unterstützung indirekter nichtfunktionaler Eigenschaften innerhalb der Verteilungsinfrastruktur BASE im Vordergrund dieser Arbeit. Es hat sich gezeigt, dass indirekte nichtfunktionale Eigenschaften die Unterstützung auf allen Ebenen, von der Anwendung bis hin zum Betriebssystem, erfordern. Die Unterstützung dieser Eigenschaften benötigt zwangsläufig einen kooperativen Ansatz. Darüber hinaus hat sich in Kapitel 3.4 gezeigt, dass eine Unterstützung indirekter nichtfunktionaler Eigenschaften allein auf der Basis von Reservierungen zumindest in Adhoc-Rechnernetzen nicht ausreichend ist. Vielmehr bedarf es eines adaptiven Ansatzes, der es Komponenten erlaubt, bei Unerfüllbarkeit einer nichtfunktionalen Anforderung diese zu reduzieren und in einem degradierten Modus fortzufahren. Der Einsatz von Reservierungen wird dadurch nicht ausgeschlossen. Beide Ansätze können durchaus miteinander kombiniert wer-

den. Im Rahmen der Erweiterung liegt der Fokus zunächst auf einem rein adaptiven Ansatz. Die Integration von Ressourcenverwaltern und Reservierungen ist für zukünftige Arbeiten vorgesehen. Zusammenfassend stellt also ein kooperativer Ansatz mit Adaption auf Komponentenebene die konzeptionelle Grundlage für diese Arbeit dar.

Die erforderlichen Punkte zur Realisierung eines solchen Ansatzes werden in den betrachteten Arbeiten bereits aufgezeigt. Damit Komponenten nichtfunktionale Anforderungen beschreiben können, ist eine formale und maschinell verarbeitbare Beschreibungssprache notwendig. Diese sollte möglichst allgemein anwendbar und ausreichend mächtig sein, um nicht bestimmte Anwendungsszenarien oder Domänen von vornherein auszuschließen. Darüber hinaus muss auf Grund der vorgesehenen Adaption die Beschreibungssprache Grenzfälle berücksichtigen, in denen nichtfunktionale Anforderungen nicht erfüllt werden können. Schließlich muss ein geeigneter Mechanismus zur Verfügung gestellt werden, der die Übergabe nichtfunktionaler Anforderungen an die Verteilungsinfrastruktur ermöglicht.

Auf Ebene der Verteilungsinfrastruktur muss betrachtet werden, auf welche Weise nichtfunktionale Anforderungen effektiv durch einen adaptiven Ansatz realisiert werden können. In dieser Hinsicht fällt vor allem die geeignete Selektion von Plugins ins Auge, da diese letztlich für die Übertragung von Aufrufen herangezogen werden und somit ausschlaggebend nichtfunktionale Eigenschaften während der Kommunikation beeinflussen. Natürlich sind zur Umsetzung vertikaler nichtfunktionaler Eigenschaften maßgeblich alle Instanzen der Kommunikationskette entscheidend. Insofern sollten sowohl Proxies als auch Broker samt dienstbasierter Registrierungen nicht vernachlässigt werden. Gerade für zeitkritische nichtfunktionale Eigenschaften sind diese von enormer Wichtigkeit. Allerdings wird in der vorliegenden Arbeit zunächst der Fokus auf die Selektion angemessener Plugins gerichtet, da diese in jedem Fall zur Unterstützung nichtfunktionaler Eigenschaften beitragen müssen. Die Anpassung von Proxies und Broker wird als zukünftige Erweiterung dieser Diplomarbeit gesehen.

In den folgenden Abschnitten werden zunächst die Möglichkeiten zur Realisierung einer Unterstützung nichtfunktionaler Eigenschaften betrachtet. Dazu wird vor allem näher auf die existierende Plugin-Architektur und die Selektion von Plugins eingegangen. Anschließend wird beschrieben, auf welchen Ebenen die Selektion von Plugins konzeptionell durchgeführt werden kann. In Kapitel 5 wird anschließend die Formalisierung nichtfunktionaler Anforderungen diskutiert und hergeleitet. Kapitel 6 behandelt die Umsetzung nichtfunktionaler Anforderungen durch die geeignete Auswahl von Plugins.

4.1 Flexibilisierung der Plugin–Architektur

In diesem Abschnitt wird die Plugin–Architektur von BASE näher beschrieben, da diese als Grundlage für die Auswahl geeigneter Plugins dient. Dabei werden Nachteile der existierenden Architektur aufgedeckt und in Hinsicht auf zukünftige Selektionsstrategien konzeptionell angepasst.

4.1.1 Originäre Plugin–Architektur

BASE verfolgt das Ziel, die Interoperabilität zu anderen Verteilungsinfrastrukturen auf flexible Art und Weise zu gewährleisten. In diesem Zusammenhang ist die primäre Aufgabe der Verteilungsinfrastruktur einen Kommunikationskanal zwischen getrennten Adressräumen bereitzustellen und Aufrufe lokaler Objekte an entfernte Objekte und umgekehrt auszuliefern. Um diese Aufgabe zu bewerkstelligen, ist es auf Grund der getrennten Adressräume zwischen Objekten¹ erforderlich, dass Aufrufe zum Versand über ein Netzwerk in Form serialisierter und adressierter Objekte, im Folgenden als *Nachrichten* bezeichnet, zur Verfügung stehen. Die notwendigen Transformationen zwischen Aufrufen und Nachrichten und die Übertragung von Nachrichten werden unter Zuhilfenahme des bereits erwähnten Konzepts der Plugins realisiert. Abbildung 4.1 zeigt die Plugin–Architektur von BASE. In BASE stellt jedes Plugin Funktionalitäten bereit, die zur Transformation und Übertragung von Nachrichten erforderlich sind. Ersteres wird mit Hilfe eines beliebigen Interoperabilitätsprotokolls realisiert. Die Übertragung von Nachrichten wird durch Zugriff auf entsprechende Mechanismen der Laufzeitumgebung durchgeführt. Normalerweise wäre hier auch der Einsatz eines Transportprotokolls notwendig, allerdings wird dieses derzeit implizit durch das Interoperabilitätsprotokoll realisiert. Im Folgenden wird dennoch davon ausgegangen, dass ein Plugin explizit ein Transportprotokoll realisieren muss.

Da auf Grund des Konzeptes der Plugins beliebige Protokolle und Übertragungswege realisiert werden können, muss entfernten Rechneinheiten mitgeteilt werden, welche Protokolle und Übertragungswege zur Kommunikation mit einer bestimmten Rechneinheit eingesetzt werden können. Diese Notwendigkeit wird in BASE behandelt, indem jedes Plugin eine eindeutige Kennung, auch *Ability* genannt, besitzt und spezielle Discovery–Plugins asynchron zur eigentlichen Kommunikation diese Kennungen beispielsweise mit Hilfe von Multi– oder Broadcasts über das Netzwerk senden.

¹Dies ist nicht immer der Fall. Mehrere lokale Objekte können zwar im gleichen Adressraum liegen, aber selbst dann ist die Kommunikation über die Verteilungsinfrastruktur möglich.

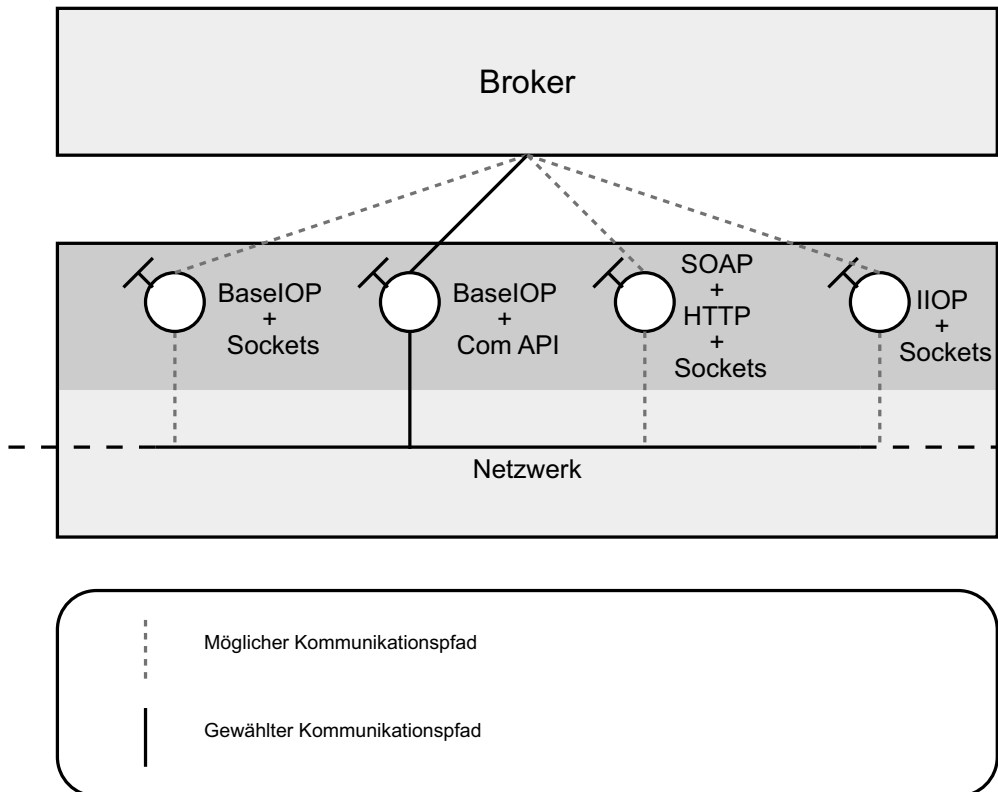


Abbildung 4.1: Originäre Plugin-Architektur

Auch während der Kommunikation spielen Abilities eine wichtige Rolle. Auf Basis der veröffentlichten Abilities kann BASE die auf beiden Seiten verfügbaren Abilities herausfinden und zur Verarbeitung eines Aufrufes eine Plugin-Implementierung verwenden, die auch auf der entfernten Seite unterstützt wird. Damit die entfernte Instanz von BASE eine entsprechende Plugin-Implementierung zur Verarbeitung der Nachricht auswählen kann, muss auch innerhalb der Nachricht die Ability ausgelesen werden können.

Ein wesentlicher Vorteil, den die Plugin-Architektur bietet, ist, dass Anfrage und Antwort eines synchronen Aufrufes jeweils mit Hilfe unterschiedlicher Protokolle und Übertragungswege versandt und empfangen werden können. Dies ist gerade dann von Vorteil, wenn zwischen Erhalt einer Anfrage und Versand der zugehörigen Antwort die zuvor verwendete Plugin-Implementierung beispielsweise auf Grund eines Verbindungsabbruchs nicht mehr eingesetzt werden kann. Unter normalen Umständen wäre eine erneute Übermittlung der Anfrage erforderlich, was zu redundantem Nachrichtenverkehr und zusätzlichen Verzögerungen führt.

4.1.2 Nachteile der originären Plugin–Architektur

Die Plugin–Architektur weist einen weiteren erheblichen Vorteil auf, der gerade für ressourcenarme Rechnersysteme ausschlaggebend ist. Auf Grund der gezielten Unterstützung einzelner Interoperabilitätsprotokolle und Netzwerkschnittstellen ist es je nach Anwendungsszenario möglich, die Menge der zur Laufzeit geladenen Plugin–Implementierungen auf ein Minimum zu reduzieren und somit eine unnötig hohe Speicherauslastung für ressourcenarme Rechereinheiten zu vermeiden. Dadurch ist die Lauffähigkeit der Verteilungsinfrastruktur auf kleineren Rechnersystemen besser gewährleistet.

Allerdings ist unter Umständen eine noch bessere Speichereffizienz und Flexibilität zu erreichen, denn derzeit impliziert die aktuelle Plugin–Architektur, dass ein einzelnes Plugin sowohl die Transformation von Aufrufen und Nachrichten als auch den Versand letzterer durchführt. Bei der Verarbeitung von Aufrufen und Nachrichten führt die Auswahl eines Plugins somit zwangsläufig zur Festlegung des Interoperabilitätsprotokolls, des Transportprotokolls und des Übertragungsmediums. Infolgedessen ist die Flexibilität der Plugin–Architektur nicht optimal. Sofern ein anderes Protokoll oder ein anderer Übertragungsweg realisiert werden soll, muss unter Umständen der Wechsel der jeweils anderen Funktionalitäten hingenommen werden oder eine neue Implementierung eines Plugins bereitgestellt werden, die die gewünschten Bedürfnisse erfüllt. Letzteres führt dann allerdings dazu, dass redundanter Code eingesetzt wird. Die Folge ist ein zusätzlicher Aufwand bei der Entwicklung von Plugins und eine erhöhte Speicherauslastung der betroffenen Rechereinheiten.

Diese Redundanzen können jedoch konzeptionell durch eine feinere Granularität bei der Organisation der Plugins vermieden werden. Unter der Annahme, dass Plugins jeweils nur eine einzige Funktionalität wie beispielsweise die Übertragung einer Nachricht oder die Kapselung einer Nachricht in ein Transportprotokoll realisieren und Plugins unterschiedlicher Funktionalitäten miteinander kombiniert werden können, hat dies im Vergleich zur bestehenden Plugin–Architektur den wesentlichen Vorteil, dass zukünftige Plugin–Implementierungen durch die klare Trennung der Funktionalitäten keine unnötigen Redundanzen zu existierenden Implementierungen aufweisen und beliebige Kombinationen an Protokollen und Netzwerkschnittstellen möglich sind. Die Überarbeitung der Plugin–Architektur führt somit zu einer effizienteren Realisierung der Plugin–Architektur bezüglich Speicherverbrauch und Wiederverwendung bei gleichzeitig erhöhter Interoperabilität zu anderen Verteilungsinfrastrukturen.

Allerdings darf nicht vergessen werden, dass eine solche restrukturierte Plugin–Architektur einen zusätzlichen Verwaltungsaufwand zur Laufzeit bedeutet.

Die Auswahl von Plugins muss anhand der bereitgestellten Funktionalität und unter Beachtung zusätzlicher Bedingungen wie beispielsweise der Vollständigkeit in Bezug auf die erforderlichen Funktionalitäten durchgeführt werden. Das bedeutet, dass eine schlechtere Performanz in Bezug auf Geschwindigkeit zu erwarten ist als im Vergleich zur originären Plugin-Architektur. Offensichtlich geht somit die erhöhte Interoperabilität und geringere Speicherauslastung zu Lasten der Verarbeitungsgeschwindigkeit. Zum ansatzweisen Ausgleich dieses Nachteils ist es denkbar, dass der zusätzliche Verwaltungsaufwand in dringenden Fällen dadurch reduziert wird, indem einzelne Plugins verwendet werden, die wie bisher die gesamte Funktionalität aller Plugin-Schichten realisieren.

Neben der erhöhten Speichereffizienz und Interoperabilität ist vor allem im Rahmen dieser Diplomarbeit die Motivation einer restrukturierten Plugin-Architektur, die Auswahl einzelner Plugins auf der Basis nichtfunktionaler Eigenschaften durchzuführen, um nichtfunktionalen Anforderungen von Komponenten gerecht zu werden. Im Vergleich zur originären Plugin-Architektur kann eine Auswahl auf einer wesentlich breiteren Basis an Plugins stattfinden und mit einer gewissen Wahrscheinlichkeit zu einem besseren Erfüllungsgrad führen.

4.1.3 Restrukturierung

Im Folgenden wird davon ausgegangen, dass Plugins nur noch eine einzige Funktionalität realisieren. Infolgedessen lassen sich Plugins, die dieselbe Funktionalität realisieren, in einer gemeinsamen konzeptionellen Verarbeitungsschicht gruppieren. Solche Schichten werden auf Grund der Plugins auch als *Plugin-Schichten* bezeichnet. Jedes Plugin einer Plugin-Schicht verfügt auf Grund derselben realisierten Funktionalität über dieselbe Schnittstelle. Die resultierenden Plugin-Schichten werden wiederum anhand der existierenden Abhängigkeiten bezüglich der jeweiligen Funktionalitäten angeordnet. Auf diese Weise entsteht ein Schichtenmodell, indem jede Plugin-Schicht auf einer anderen Plugin-Schicht aufsetzt und deren wohldefinierte Schnittstelle nutzen kann. In Abbildung 4.2 werden die Plugin-Schichten in Bezug zur Plugin-Architektur gezeigt. Unter Beachtung der Anordnung der einzelnen Plugin-Schichten kann nun aus jeder Plugin-Schicht ein beliebiges Plugin selektiert und mit Plugins aus anderen Schichten kombiniert werden. Somit entsteht gewissermaßen eine Kette von Plugins, die zur Verarbeitung von Aufrufen und Nachrichten in jeweils eine Richtung durchlaufen wird. Damit eine Plugin-Kette tatsächlich verwendet werden kann, muss allerdings gewährleistet sein, dass die Plugin-Kette genau ein Plugin pro Plugin-Schicht enthält. Dieses Kriterium wird im Folgenden als *Vollständigkeit* der Plugin-Kette bezeichnet.

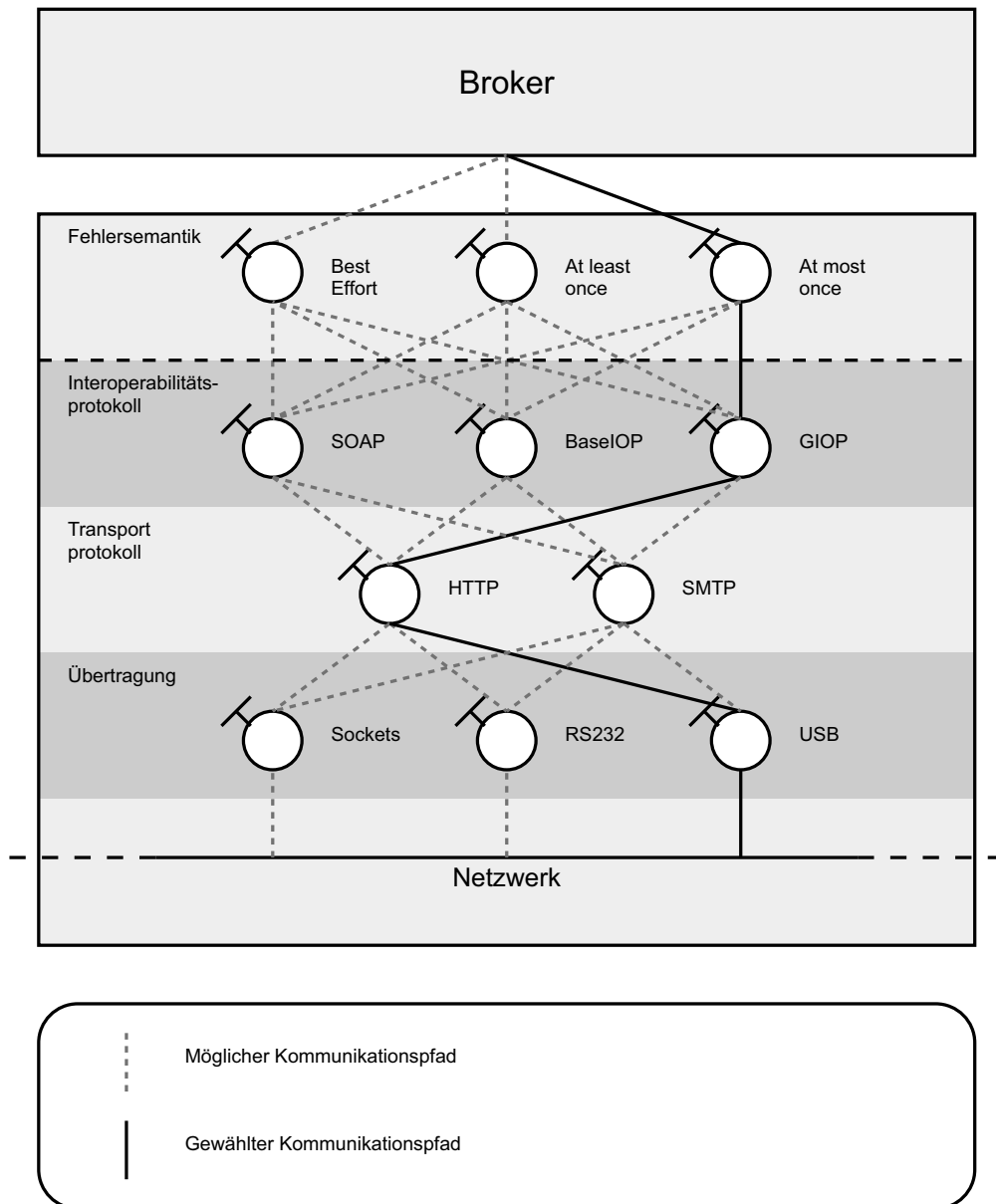


Abbildung 4.2: Restrukturierte Plugin-Architektur

Manche Plugin-Implementierungen der originären Plugin-Architektur können nicht anhand der realisierten Funktionalitäten der neuen Plugin-Architektur eindeutig zugeordnet werden. Ein Beispiel hierfür ist Java RMI, das bereits in der originären Plugin-Architektur in Form eines Plugins zur Verfügung steht. Java RMI realisiert sowohl das Interoperabilitäts- und Transportprotokoll als auch die Übertragung von Nachrichten über Sockets. Insofern handelt es sich bei diesem Plugin um ein schichten-übergreifendes Plugin, dessen Integration in die neue Plugin-Architektur zumindest in Hinsicht auf die Trennung von Funktionalitäten nicht ohne weiteres möglich ist. Um solche Plugins nicht von vornherein auszuschließen, wird die Vereinbarung getroffen, dass solche Plugins gezwungenermaßen alle Pluginschichten realisieren, die unterhalb der ersten realisierten Pluginschicht liegen. Auf diese Weise wird die Komplexität bei der Bildung von Plugin-Ketten zwar erhöht, dafür ist allerdings die Wiederverwendung existierender Plugins im Rahmen der neuen Plugin-Architektur möglich.

Bisher wurden ausschließlich die Funktionalitäten Interoperabilität, Transport und Übertragung betrachtet, da dies diejenigen Funktionalitäten sind, die in der existierenden Verteilungsinfrastruktur bereits in Form von Plugins realisiert wurden und die in jedem Fall zur Kommunikation mit entfernten Instanzen der Verteilungsinfrastruktur erforderlich sind. Eine Funktionalität, die in der Vergangenheit von den Stellvertreterobjekten übernommen wurde, konzeptionell jedoch auch als eigene Plugin-Schicht realisiert werden kann, wurde in den bisherigen Betrachtungen vernachlässigt. Dabei handelt es sich um die Unterstützung einzelner Fehlersemantiken, die während der Kommunikation zwischen Objekten zum Einsatz kommen kann. Da einerseits von Objekt zu Objekt unterschiedliche Fehlersemantiken relevant sein können und andererseits dieselbe Fehlersemantik für mehrere Objekte eventuell benötigt wird, ist die Realisierung von Fehlersemantiken in Form einzelner Plugins aus Gründen der Wiederverwendung sinnvoll. Auf diese Weise wird zum einen die automatische Erzeugung der Stellvertreterobjekte einfacher und kleiner und zum anderen die Plugin-Architektur um eine logisch zugehörige Funktionalität erweitert.

Die Einführung einer Plugin-Schicht zur Behandlung von Fehlersemantiken zeigt jedoch, dass es einen fundamentalen Unterschied im Vergleich zu den vorherigen Plugin-Schichten gibt. Die bisher betrachteten Plugin-Schichten kommen jeweils zur einmaligen Verarbeitung eines Aufrufes zwischen Instanzen der Verteilungsinfrastrukturen zum Einsatz. Das bedeutet, dass sofern ein Aufruf erfolgreich transformiert und versandt oder empfangen und zurück transformiert wurde, dieser als erfolgreich verarbeitet betrachtet wird. In Bezug auf Plugins, die im Einzelnen eine Fehlersemantik realisieren, ist die Verarbeitung

eines Plugins jedoch erst genau dann erfolgreich, wenn ein Aufruf tatsächlich vom Kommunikationspartner an das entfernte Objekt ausgeliefert wurde. Auf Grund von Übertragungsstörungen oder fehlerhaftem Verhalten des Kommunikationspartners kann es passieren, dass die Auslieferung jedoch nicht stattgefunden hat. Um diese Tatsache zur Laufzeit feststellen zu können, muss der empfangende Kommunikationspartner die Auslieferung eines Aufrufes dem Absender bestätigen. Erhält der Absender keine positive Bestätigung, muss je nach Fehlersemantik eine Nachricht wiederholt transformiert und übertragen werden. Da eine Fehlersemantik durch ein Plugin realisiert wird, ist dieses für die Behandlung fehlgeschlagener Übertragungen von Nachrichten zuständig. Je nach Fehlersemantik muss unter Umständen die Wiederholung einer Nachricht veranlasst werden. Die Wiederholung eines Aufrufes führt in diesem Fall zu einer neuen Nachricht, weshalb nochmals der Teil der Plugin-Kette durchlaufen wird, der unterhalb der Plugin-Schicht für Fehlersemantiken liegt. In diesem Zusammenhang kann man deshalb erkennen, dass es eine Grenze innerhalb der Plugin-Architektur gibt, anhand derer sich Plugin-Schichten in Bezug auf Aufrufe unterscheiden. Sowohl in der Interoperabilitäts- und Transportschicht als auch in der Übertragungsschicht wird die Verarbeitung eines Plugins dann erfolgreich betrachtet, wenn allein die Transformation oder Übertragung erfolgreich durchgeführt wurde. Auf Ebene der Fehlersemantik-Schicht ist die Verarbeitung eines Aufrufes erst dann erfolgreich, wenn sichergestellt wurde, dass der Aufruf beim entfernten Objekt tatsächlich ausgeliefert wurde. Dieser konzeptionelle Unterschied wird in Abbildung 4.2 durch die gestrichelte Markierung zwischen der Fehlersemantik-Schicht und der Interoperabilitätsschicht angedeutet.

Neben den bisher betrachteten Pluginschichten ist es zukünftig eventuell notwendig, dass weitere Schichten integriert werden. Da diese Schichten offensichtlich nicht zur grundlegenden Kommunikation mit anderen Instanzen der Verteilungsinfrastruktur benötigt werden, sind diese Schichten zumindest zu diesem Zeitpunkt optional. Die Position einer optionalen Plugin-Schicht im Schichtenmodell ist im Einzelnen von der Funktionalität und der logischen Verarbeitungsreihenfolge abhängig und muss von Fall zu Fall geprüft werden. Beispiele für solche Plugin-Schichten sind beispielsweise eine Kompressionsschicht oder eine Schicht zur Verschlüsselung von Aufrufen. Auf Grund der wohldefinierten Schnittstellen zwischen Plugin-Schichten ist die Integration optionaler Plugin-Schichten relativ einfach.

4.1.4 Konzeptionelle Integration der Plugin-Auswahl

Die restrukturierte Plugin-Architektur stellt eine ideale Grundlage zur Unterstützung nichtfunktionaler Eigenschaften dar. Während der Durchführung von Aufrufen kann innerhalb der Plugin-Architektur eine optimale Auswahl an Plugins vorgenommen werden, deren Eigenschaften insgesamt eine gegebene nichtfunktionale Anforderung möglichst erfüllen. Je nach betrachteten Eigenschaften wird die Auswahl an Plugins zukünftig sogenannten *Selektionsstrategien* zugeschrieben. Die genaue Funktionsweise der Selektionsstrategien sei zunächst dahingestellt, da an dieser Stelle vielmehr interessant ist, auf welcher Ebene diese am besten realisiert und in die existierende Verteilungsinfrastruktur integriert werden können. In Kapitel 6 wird dies jedoch nachgeholt und genauer diskutiert.

Selektionsstrategien in Komponenten

Einerseits wäre es möglich, dass Komponenten selbst dafür verantwortlich sind, ihre nichtfunktionalen Anforderungen zu erfüllen, indem sie die Selektion von Plugins eigenständig durchführen und der Verteilungsinfrastruktur quasi vorschreiben, welche Plugins letztendlich zur Kommunikation mit anderen Komponenten verwendet werden sollen.

Das hat den Vorteil, dass die nichtfunktionalen Anforderungen einer Komponente nicht formuliert und der Verteilungsinfrastruktur mitgeteilt werden müssen. Letztere wäre nach wie vor einzig und allein dafür zuständig, die Kommunikation zwischen Komponenten zu ermöglichen, unabhängig davon, ob und wie eine Unterstützung nichtfunktionaler Eigenschaften realisiert wird.

Allerdings bringt der Vorteil, dass nichtfunktionale Anforderungen nicht explizit formuliert werden müssen, einen erheblichen Nachteil mit sich. Für eine effektive Selektion von Plugins müssen die nichtfunktionalen Eigenschaften dieser Komponenten bekannt sein und von der Verteilungsinfrastruktur zur Verfügung gestellt werden, damit Selektionsstrategien innerhalb von Komponenten überhaupt realisiert werden können. Zum einen bedeutet dies, dass die Abstraktion, die eine Verteilungsinfrastruktur definitionsgemäß bietet, aufgeweicht wird, was zu einer höheren Kopplung zwischen Verteilungsinfrastruktur und Komponenten führt. Zum anderen ist langfristig mit einem gewissen Wartungsaufwand für Komponenten zu rechnen, da sich nichtfunktionale Eigenschaften unter Umständen in Zukunft verändern, obsolet werden oder neue Eigenschaften integriert werden.

Die Integration von Selektionsstrategien in Komponenten führt zu einigen weiteren Nachteilen. Erstens bedeutet dieser Ansatz zusätzlichen Aufwand für

den Komponentenentwickler und lenkt diesen mehr oder weniger von seiner eigentlichen Aufgabe ab. Zweitens steht die Selektionsstrategie lediglich der Komponente zur Verfügung, für die sie entwickelt wurde, obwohl sie eventuell in anderen Komponenten, die dasselbe Anwendungsszenario voraussetzen, ebenfalls sinnvoll eingesetzt werden könnte. Als letztes besteht die Gefahr einer starken Vermischung zwischen dem Code der Komponente und dem Code zur Realisierung einer Selektionsstrategie. Eine saubere Trennung ist praktisch nur schwer zu erreichen und verhindert die Wiederverwendung von Selektionsstrategien in anderen Komponenten. Langfristig führt dieser Ansatz zu einem verhältnismäßig hohen Aufwand bei der Entwicklung von Komponenten und angemessenen Selektionsstrategien.

Selektionsstrategien in der Verteilungsinfrastruktur

Da die Realisierung von Selektionsstrategien innerhalb von Komponenten mit Nachteilen verbunden ist, wird eine weitere Möglichkeit betrachtet, die die Realisierung einer Selektionsstrategie direkt in der Verteilungsinfrastruktur vorsieht. Die Verteilungsinfrastruktur übernimmt die Funktion einer Selektionsstrategie und führt anhand gegebener nichtfunktionaler Anforderungen von Komponenten die Selektion von Plugins durch. Damit die Verteilungsinfrastruktur als Selektionsstrategie fungieren kann, müssen die nichtfunktionalen Anforderungen von Komponenten im Vergleich zum vorherigen Ansatz natürlich explizit spezifiziert und zur Verfügung gestellt werden. Folglich ist es in diesem Fall unvermeidbar, dass Komponentenentwickler die jeweiligen nichtfunktionalen Anforderungen in einer formalen Beschreibung ausdrücken.

Ein offensichtlicher Vorteil dieser Alternative ist die Abstraktion mittels formaler nichtfunktionaler Beschreibungen. Komponentenentwickler werden nicht mit der Aufgabe belastet, für jede Komponente eine neue Selektionsstrategie zu implementieren, sondern drücken ihre Anforderungen in einer Beschreibung aus, die von der Verteilungsinfrastruktur berücksichtigt werden kann. Mittels formaler Beschreibung werden Komponenten und Selektionsstrategien entkoppelt. Somit können prinzipiell beliebige Selektionsstrategien implementiert und eingesetzt werden, ohne dass Änderungen an Komponenten deshalb erforderlich werden.

Allerdings sind Komponenten in diesem Ansatz nicht ohne weiteres in der Lage, zu erkennen, ob die geäußerten nichtfunktionalen Anforderungen durch die Verteilungsinfrastruktur erfüllbar sind oder nicht. Die Unerfüllbarkeit einer nichtfunktionalen Anforderungen muss deshalb zusätzlich behandelt werden. Dies kann nur geschehen, indem die Verteilungsinfrastruktur explizit die

Unerfüllbarkeit einer Anforderung der beteiligten Komponente meldet. Die benachrichtigte Komponente kann daraufhin zur Laufzeit in irgendeiner Form adaptieren.

Eine generische Selektionsstrategie, die beliebige nichtfunktionale Domänen zur selben Zeit ausreichend unterstützt, ist, wie bereits in der Betrachtung der Literatur erwähnt wurde, kaum zu realisieren. Insofern ist eine einzige und fest integrierte Selektionsstrategie innerhalb der Verteilungsinfrastruktur nur unzureichend. Es wäre besser, wenn Selektionsstrategien je nach Anwendungsszenario eine eingeschränkte Menge nichtfunktionaler Eigenschaften behandeln und zum Ausgleich dieser Beschränkung beliebig ausgetauscht werden könnten. Um dies zu ermöglichen, ist natürlich eine starke Entkopplung zwischen Funktionalität der Verteilungsinfrastruktur und ihrer Selektionsstrategien von Vorteil.

Da Selektionsstrategien innerhalb der Verteilungsinfrastruktur Vorteile gegenüber Komponenten bieten, die der Komponenten-basierte Ansatz nicht aufweist, dient das zuletzt beschriebene Konzept als weitere Grundlage für diese Arbeit. Auf Grund dessen, dass die Unterstützung nichtfunktionaler Anforderungen innerhalb der Verteilungsinfrastruktur realisiert wird, entstehen zwei neue Zuständigkeiten:

Entwicklung anwendungsspezifischer Selektionsstrategien Die Aufgabe, eine spezifische Selektionsstrategie für ein bestimmtes Anwendungsszenario zu entwickeln. Hierzu gehört sowohl die Feststellung geeigneter nichtfunktionaler Eigenschaften als auch die Bereitstellung einer Implementierung, die auf der Basis der relevanten nichtfunktionalen Eigenschaften prinzipiell eine geeignete Menge an Plugins bestimmen kann, die erstens zur Kommunikation mit entfernten Instanzen der Verteilungsinfrastruktur genutzt werden können und zweitens nichtfunktionale Anforderungen prinzipiell erfüllen können.

Administration von Selektionsstrategien Die Aufgabe der Integration und Aktivierung spezifischer Selektionsstrategien, die den Bedürfnissen der installierten Komponenten gerecht werden.

Um den gewählten Ansatz realisieren zu können, muss in den nächsten Kapiteln zum einen die Formalisierung nichtfunktionaler Eigenschaften und Anforderungen und zum anderen die Erfüllung nichtfunktionaler Anforderungen durch die Verteilungsinfrastruktur diskutiert werden.

Kapitel 5

Formalisierung

In Abschnitt 4.1.4 wurde festgestellt, dass für die Verwirklichung eines kooperativen Ansatzes eine formale Beschreibung nichtfunktionaler Anforderungen notwendig ist. In diesem Kapitel wird deshalb zunächst untersucht, in welcher Form nichtfunktionale Anforderungen spezifiziert und von Komponentenentwicklern zur Verfügung gestellt werden können. Dazu wird in Bezug auf die Anforderungen aus Abschnitt 2.2 auf eine einfache Handhabung der Erweiterung geachtet, indem die Aktivität der Bereitstellung einer nichtfunktionalen Anforderung möglichst einfach gehalten wird. Es wurde bereits darauf hingewiesen, dass der Aspekt der Unerfüllbarkeit unter Umständen Einfluss auf die Form nichtfunktionaler Anforderungen hat. Deshalb wird darüber hinaus betrachtet, auf welchem Weg die Unerfüllbarkeit einer nichtfunktionalen Anforderung effektiv behandelt werden kann.

5.1 Nichtfunktionale Informationen

Komponenten weisen im Einzelnen spezielle Bedürfnisse an das Kommunikationsverhalten der Verteilungsinfrastruktur auf, die zur Bereitstellung der Funktionalität einer Komponente erfüllt werden müssen. Diese Bedürfnisse sollen zukünftig in *nichtfunktionalen Anforderungen* erfasst werden. Analog zu den nichtfunktionalen Anforderungen beschreiben sogenannte *nichtfunktionale Angebote*, welche Bedürfnisse durch die Verteilungsinfrastruktur gedeckt werden können. Sind sowohl Angebot als auch Anforderung bekannt, ist prinzipiell die Feststellung der Erfüllbarkeit durch einen Vergleich beider möglich. Hierzu ist eine genauere Betrachtung der jeweiligen Inhalte erforderlich.

Nichtfunktionale Eigenschaften

Eine *nichtfunktionale Eigenschaft* dient der Beschreibung eines einzelnen Bedürfnisses und stellt sowohl für nichtfunktionale Anforderungen als auch für nichtfunktionale Angebote ein Grundelement dar. Beispielsweise ist eine Komponente zur Videoübertragung darauf angewiesen, dass zur Laufzeit ausreichend *Bandbreite* zur Verfügung steht, um anfallende Bildinformationen in angemessener Qualität übertragen zu können. Eine Komponente für Internettelefonie hat unter Umständen das Bedürfnis, dass die *Latenz* zwischen Senden und Empfangen von Datenpaketen möglichst gering ist, um bei der Wiedergabe der Audiodaten keine unerwünschten Aussetzer zu verursachen. Mit Hilfe nichtfunktionaler Eigenschaften können solche Bedürfnisse beschrieben werden.

Nichtfunktionale Domänen und Aspekte

Nichtfunktionale Eigenschaften können je nach Bezug zu bestimmten Anwendungsszenarien *nichtfunktionalen Domänen* zugeordnet werden. In Kapitel 3.1.1 wurden bereits einige Domänen vorgestellt. Zum Beispiel werden die zuvor genannten nichtfunktionalen Eigenschaften Bandbreite und Latenz typischerweise der Domäne *Echtzeit* zugeschrieben. Zur Vereinfachung erfassen *nichtfunktionale Aspekte* eine Untermenge nichtfunktionaler Eigenschaften einer Domäne und ermöglichen die einfache Referenzierung häufig gemeinsam wiederkehrender nichtfunktionaler Eigenschaften. Dadurch lassen sich beispielsweise die Eigenschaften *Bandbreite*, *Latenz* und *Jitter* als Aspekt *Multimediaqualität* zusammenfassen. Aus Gründen der Vereinfachung wird angenommen, dass nichtfunktionale Aspekte immer disjunkte Mengen an nichtfunktionalen Eigenschaften spezifizieren. Andernfalls könnten Konflikte auftreten, da ohne weiteres nicht bekannt ist, auf welchem Wege redundante nichtfunktionale Eigenschaften aus mehreren Aspekten vereinigt werden.

Beschreibung nichtfunktionaler Informationen

Mit Hilfe von nichtfunktionalen Eigenschaften ist es grundlegend möglich, nichtfunktionale Anforderungen und Angebote zu beschreiben, indem diese eine Menge nichtfunktionaler Eigenschaften beinhalten. Allerdings stellt sich die Frage, wie eine konkrete Beschreibungssprache in diesem Fall aussieht und welche Mächtigkeit hierzu erforderlich ist. PCOM bietet bereits eine Beschreibungssprache für nichtfunktionale Anforderungen und Angebote, wie man an dem aufgeführten Beispiel aus Kapitel 3.3.1 erkennen konnte. Allerdings ist die Beschreibung nichtfunktionaler Informationen mit Hilfe der PCOM-Verträge nur

eingeschränkt möglich. Dies liegt daran, dass die Verträge zwar die Benennung nichtfunktionaler Eigenschaften und die Zuweisung angemessener Werte erlauben. Der Vergleich nichtfunktionaler Eigenschaften wird jedoch nur durch einfache String-Vergleiche umgesetzt. Ein offensichtlicher Nachteil dieser Beschreibungssprache ist somit, dass lediglich untypisierte Vergleiche möglich sind, die im Einzelnen zu Inkonsistenzen zwischen Zuweisung und Vergleich führen können. Da insofern bereits eine einfache Beschreibung nichtfunktionaler Eigenschaften zur Verfügung steht, wird im Rahmen dieser Arbeit aus Gründen der Wiederverwendung eine Erweiterung dieser angestrebt.

Auch in der Literatur wird die formale Erfassung generischer, nichtfunktionaler Eigenschaften behandelt [8, 9, 5, 20]. Eine Betrachtung dieser Arbeiten hat gezeigt, dass die formale und generische Beschreibung nichtfunktionaler Eigenschaften in diesen Arbeiten relativ ähnlich realisiert wird. Aus diesem Grund wurde die übersichtlichste der Beschreibungssprachen, QML [8, 9], ausgewählt und näher betrachtet. Der Grund, weshalb QML ausgewählt wurde ist, dass QML nicht für eine bestimmte Verteilungsinfrastruktur entworfen wurde und somit erstens keine Anforderungen an die zugrunde liegende Verteilungsinfrastruktur stellt und zweitens zur Beschreibung nichtfunktionaler Informationen keine Rücksicht auf proprietäre Mechanismen oder Schnittstellen nimmt. Der Vorteil einer implementierungs-unabhängigen Modellierung nichtfunktionaler Informationen besteht darin, dass die Verwendung eines entsprechend unabhängigen Modells im Vergleich zu bereits individualisierten Modellen einfacher ist.

QML ist eine Erweiterung der *Unified Modelling Language* [28] und wird primär in den Spezifikations- und Entwurfsphasen der Softwareentwicklung eingesetzt, da nach Meinung der Autoren von QML bereits in diesen Phasen eine frühzeitige und genaue Betrachtung nichtfunktionaler Eigenschaften von immenser Bedeutung ist. Zu diesem Zweck existieren sogenannte *Dimensionen*, die in Bezug zu dieser Arbeit konzeptionell nichtfunktionalen Eigenschaften entsprechen und die Beschreibung einzelner Bedürfnisse ermöglichen. Darüber hinaus können nichtfunktionale Anforderungen und Angebote mit Hilfe von *Kontrakten* formuliert werden, die jeweils eine Menge quantifizierter Dimensionen vereinigen, um die insgesamt anfallenden Bedürfnisse einer Anwendung oder einer Komponente vollständig zu beschreiben. Dieser Zusammenhang entspricht konzeptionell der Beziehung zwischen nichtfunktionalen Anforderungen, Angeboten und Eigenschaften, die zuvor aufgezeigt wurde. Darüber hinaus unterstützt QML noch weitere Anwendungsfälle, die in dieser Arbeit bisher noch nicht betrachtet wurden. Beispielsweise können neu erzeugte Kontrakte von

existierenden Kontrakten zur einfachen Weiterverwendung erben. Ein weiteres Beispiel ist die Angabe von Perzentilen zur Quantifizierung von Dimensionen, die genau dann sinnvoll ist, wenn Eigenschaften wie beispielsweise *Zuverlässigkeit* über einen bestimmten Zeitraum anteilig festgelegt werden sollen. So würde die Zuverlässigkeit eines Dienstanbieters durch 99%ige Zuverlässigkeit beschrieben werden.

Das grundlegende Modell aus QML entspricht dem Modell, das bisher hergeleitet wurde. Die Möglichkeiten, die darüber hinaus von QML angeboten werden, sind allerdings für diese Arbeit erstens nicht erforderlich und zweitens nicht geeignet, da eine einfache Erweiterung zur Unterstützung nichtfunktionaler Eigenschaften realisiert werden soll, die auch auf ressourcenarmen Rechnersystemen eingesetzt werden kann.

5.1.1 Formulierung von Bedürfnisgraden

Um sowohl nichtfunktionalen Anforderungen als auch nichtfunktionalen Angeboten eine gewisse Aussagekraft zu verleihen und diese auch später vergleichen zu können, ist es entscheidend, dass die jeweils darin enthaltenen nichtfunktionalen Eigenschaften den Grad der jeweiligen Bedürfnisse widerspiegeln.

Explizite Formulierung

In QML wird dies durch explizit formulierte Wertzuweisungen auf einer Skala realisiert, die einen logischen Bezug zur Eigenschaft aufweist. Beispielsweise kann für eine nichtfunktionale Eigenschaft *Energieverbrauch* eine absolute Skala mit der Einheit *milliAmpereSekunden* verwendet werden. Der Vorteil expliziter Wertzuweisungen unter Zuhilfenahme einer Skala ist, dass nichtfunktionale Anforderungen und Angebote exakt formuliert werden können. Allerdings hat das den Nachteil, dass für jede nichtfunktionale Eigenschaft zunächst eine angemessene Skala gefunden werden muss. Unter Umständen kann das Auffinden einer solchen Skala je nach Eigenschaft für Komponentenentwickler mehr oder weniger schwierig sein, da eine Skala samt Einheit einen logischen Bezug zur Eigenschaft aufweisen sollte. Hierfür ist in vielen Fällen wiederum fachspezifisches Wissen erforderlich. Darüber hinaus ist im Rahmen der Plugin-Architektur die Erhebung exakter Werte für einzelne Plugins unter Umständen schwierig, da hierzu geeignete Messmethoden zur Verfügung stehen müssen. Spätestens bei der praktischen Erhebung solcher Werte fällt auch auf, dass nicht alle nichtfunktionalen Eigenschaften konstante Werte aufweisen und auf Grund verschiedener Einflussfaktoren gewissen Schwankungen unterliegen. Zumindest für die Feststellung geeigneter Werte und die Verwendung dieser in nichtfunktionalen

Anforderungen stellt letzteres jedoch kein sonderliches Problem dar, da eine Veränderung einer Anforderung durch erneute Mitteilung der Anforderung erkannt werden kann.

Implizite Formulierung

Neben der expliziten Zuweisung von Werten für einzelne nichtfunktionale Eigenschaften ist jedoch gerade im Kontext der Plugin-Architektur von BASE eine weitere Art der Festlegung von Bedürfnisgraden denkbar. Diese wird im Folgenden kurz vorgestellt.

Die Idee ist, dass der Grad eines Bedürfnisses nicht explizit festgelegt wird, sondern mit Hilfe einer explizit vorgegebenen Ordnung über die verfügbaren Plugins hinweg und der Referenzierung eines einzelnen Plugins aus dieser Ordnung, welches implizit das Bedürfnis ausreichend erfüllt, festgelegt wird. Als Grundlage für Selektionsstrategien ist dies ausreichend und die erforderliche Angabe einer solchen Ordnung ist relativ einfach. Eine Selektionsstrategie könnte folglich diejenigen Plugins in Betracht ziehen, die das betrachtete Bedürfnis laut Ordnung mindestens so gut wie das referenzierte Plugin erfüllen. Somit ließe sich der Nachteil einer expliziten und eventuell schwierigen Wertzuweisung vermeiden, da Komponentenentwickler in diesem Fall zum einen keine geeignete Skala finden und zum anderen auch keine angemessenen Werte erheben müssten. Die indirekte Festlegung der Grade durch Referenzierung von Plugins scheint auf den ersten Blick zumindest aus Sicht eines Komponentenentwicklers relativ praktisch und einfach zu sein.

Bei einer genaueren Betrachtung zeigt sich jedoch, dass die Referenzierung geordneter Plugins nicht sinnvoll ist. Erstens setzt diese Alternative voraus, dass die Menge der lokal verfügbaren Plugins stabil ist, damit spezifizierte Referenzen auf Plugins im Laufe der Zeit nicht auf Grund fehlerhafter Referenzen ungültig werden. Die Plugin-Architektur von BASE zielt jedoch gerade auf den flexiblen Einsatz beliebiger Plugins ab, weshalb einerseits durch dynamisches Laden oder Entladen und andererseits durch eine Rekonfiguration der Verteilungsinfrastruktur Veränderungen in der Verfügbarkeit lokaler Plugins auftreten können. Folglich ist die Menge der Plugins nicht immer stabil und jede Veränderung der Plugin-Menge führt gezwungenermaßen zu einer Überarbeitung aller lokal verwendeten nichtfunktionalen Anforderungen, damit diese wieder eingesetzt werden können. Bei entsprechender Vielzahl an Anwendungen kostet dies einen erheblichen Aufwand, der durch den Strategie-Administrator bewältigt werden muss. Im dynamischen Fall ist die Anpassung nichtfunktionaler Anforderungen praktisch sogar unmöglich. Diese Schwierig-

keit wäre durch eine zusätzliche Abstraktion von einzelnen Plugins zu umgehen, indem allgemein bekannte Plugin-Klassen definiert und nur diese referenziert werden. Allerdings setzt dies wiederum zusätzliches Wissen über diese Klassen und vorallem Konsistenz zwischen beliebigen Instanzen und Versionen der Verteilungsinfrastruktur voraus. Folglich macht eine indirekte Festlegung von Ausprägungen nur bedingt Sinn und erfordert zumindest im Vergleich zur expliziten Wertzuweisung einen zusätzlichen Aufwand bei der Umsetzung und der Ausführung.

Ein zweites Problem der indirekten Festlegung stellt die fehlende Quantifizierung nichtfunktionaler Eigenschaften von Plugins dar. Da keine konkreten Werte bekannt sind, können bei späteren Analysen durch Selektionsstrategien auch keine Betrachtungen quantitativer Abhängigkeiten zwischen Plugins angestellt werden. Eine Betrachtung dieser Abhängigkeiten ist gerade dann sinnvoll, wenn die Auswahl eines Plugins die Ausprägungen einer oder mehrerer nichtfunktionaler Eigenschaften eines anderen Plugins beeinflussen. Unter Umständen bedeutet das, dass Selektionsstrategien nur ineffizient realisiert werden können.

Drittens treten spätestens bei der Verwendung vertikaler Eigenschaften Probleme auf. Der Grund hierfür ist, dass durch die Referenz eines einzelnen Plugins nur der Bedürfnisgrad dieses einzelnen Plugins ausgedrückt werden. Benötigt wird jedoch der insgesamt realisierbare Bedürfnisgrad aller Plugins. Folglich wäre es erforderlich, mehrere Referenzen spezifizieren zu können.

Im weiteren Verlauf der Arbeit werden auf Grund der Nachteile der impliziten Wertzuweisung nichtfunktionale Anforderungen auf der Basis explizit formulierter Ausprägungen in Anlehnung an QML hergeleitet, da die indirekte Festlegung von Ausprägungen erhebliche Nachteile im Gegensatz zu explizit formulierten Ausprägungen aufweist.

5.1.2 Wertzuweisungen

Wertzuweisungen können nur dann durchgeführt werden, wenn eine nichtfunktionale Eigenschaft einen Werttyp besitzt, der die möglichen Werte einer Eigenschaft und eventuelle Vergleichsoperatoren definiert. Da ein Werttyp in manchen Fällen eine Ordnung impliziert, ist ein sinnvoller Vergleich offensichtlich nur dann möglich, wenn bekannt ist, welche Ausrichtung der Ordnung für die jeweilige nichtfunktionale Eigenschaft angemessen ist. Beispielsweise besitzen numerische Werttypen eine implizite Ordnung, die den Vergleich mit Hilfe der $>$, $<$ und $=$ Operatoren ermöglicht. Die Ausrichtung einer nichtfunktio-

nalen Eigenschaft sagt etwas darüber aus, ob bei einem Vergleich größere oder kleinere Werte besser als der eigentlich geforderte Wert sind. Die nichtfunktionale Eigenschaft *Latenz* impliziert beispielsweise, dass kleinere Werte besser sind, wohingegen die nichtfunktionale Eigenschaft *Bandbreite* größere Werte bevorzugt. Im Wesentlichen hängt die Ausrichtung der Ordnung vom Verwendungszweck der Eigenschaft ab und kann deshalb zum einen von Eigenschaft zu Eigenschaft unterschiedlich ausfallen und zum anderen lediglich vom Komponentenentwickler vorgegeben werden. In Rahmen dieser Arbeit wird die Ausrichtung einer Eigenschaft als *absteigend* bezeichnet, falls kleinere Werte besser sind und *aufsteigend*, falls größere Werte besser sind. Diese Konvention wurde aus [3] übernommen. Auch QML behandelt die Wertzuweisung für nichtfunktionale Eigenschaften auf diese Weise. Nichtfunktionale Eigenschaften, deren Wertetypen keine Ordnung aufweisen, ermöglichen lediglich die Prüfung auf Gleichheit.

5.1.3 Lokalität

Man muss davon ausgehen, dass unter Umständen manche nichtfunktionalen Eigenschaften einer Anforderung explizit nur auf der lokalen Rechneinheit erfüllt werden sollen. Dies ist zum Beispiel dann der Fall, wenn die Eigenschaft *Energieverbrauch* nur für die lokale Rechneinheit interessant ist. Andererseits ist häufig die Umsetzung einer nichtfunktionalen Eigenschaft nur dann sinnvoll, wenn diese auf beiden miteinander kommunizierenden Rechneinheiten erfüllt wird. Gerade zeitkritische Eigenschaften wie beispielsweise die *Latenz* von Aufrufen stellt eine solche Eigenschaft dar. Folglich ist es erforderlich, dass die *Lokalität*, der Bezug einer Eigenschaft, ausdrücklich festgelegt werden kann. Zur weiteren Unterscheidung werden Eigenschaften, die geräte-übergreifend umgesetzt werden müssen, als *globale* Eigenschaften bezeichnet. Alle andere Eigenschaften, deren Realisierung sich auf das lokale Gerät beschränkt, werden zukünftig *lokale* Eigenschaften genannt.

5.2 Nichtfunktionale Anforderungen

Die Grundlage, die in Abschnitt 5.1 beschrieben wurde, ermöglicht bereits, einfache nichtfunktionale Anforderungen zu formulieren. Hierzu muss lediglich eine Menge nichtfunktionaler Eigenschaften erfasst werden, die bei der Erfüllung der Anforderungen betrachtet werden sollen. Die jeweils gewünschten Bedürfnisgrade müssen durch den Komponentenentwickler mit Hilfe von Wertzuweisungen spezifiziert werden, da nur diesem die Bedürfnisse einer Komponente

bekannt sind. Wie bereits in 5.1.2 erläutert wurde, sind in dieser Hinsicht auch die Ausrichtung und die Lokalität einer nichtfunktionalen Eigenschaft relevant. Auch diese müssen für jede Eigenschaft angegeben werden.

Veränderlichkeit

Wird durch einen Komponentenentwickler eine nichtfunktionale Anforderung bereitgestellt, stellt sich die Frage, welchen Bezug diese zur nachfolgenden Kommunikation hat und wie lange diese Anforderung gelten soll. Es sind mehrere Möglichkeiten denkbar, wie eine solche Beziehung aussehen könnte:

- Gültigkeit für alle Methodenaufrufe
- Gültigkeit für mehrere Methodenaufrufe
- Gültigkeit für einzelne Methodenaufrufe

Die Gültigkeit nichtfunktionaler Anforderungen hängt stark mit der Veränderlichkeit dieser zusammen. Je nach Funktionalität einer Komponente können nichtfunktionalen Anforderungen während einer Bindung konstant sein oder sich auf Grund irgendwelcher Faktoren zur Laufzeit verändern. Sind die Anforderungen einer Komponente stabil, ist es vorteilhaft, wenn die Anforderungen der Verteilungsinfrastruktur zu Beginn der Ausführung einer Komponente ein einziges Mal der Verteilungsinfrastruktur mitgeteilt werden und die Gültigkeit einer Anforderung sich auf die gesamte Dauer der Ausführung einer Komponente erstreckt. Es kann allerdings auch erforderlich sein, dass zur Laufzeit unterschiedliche Anforderungen zum Einsatz kommen müssen. Dies ist beispielsweise der Fall, wenn Komponenten mehrere Funktionalitäten gleichzeitig anbieten und je nach Funktionalität unterschiedliche Anforderungen gestellt werden. Auch Benutzer, die zur Laufzeit die Qualität der Ausführung einer Komponente beeinflussen können, sind eine Ursache für veränderte Anforderungen. In diesen Fällen ist es durchaus vorteilhaft, wenn nichtfunktionale Anforderungen für Einzelne oder eine Folge von Methodenaufrufen festgelegt werden können. Letzteres führt zwar zu einem vergleichsweise hohem Nachrichtenverkehr zwischen Verteilungsinfrastruktur und Komponenten. Dafür ist jedoch eine flexiblere Unterstützung nichtfunktionaler Anforderungen möglich. Eine längere Gültigkeit kann je nach Bedarf durch mehrfache Angabe derselben Anforderungen simuliert werden.

5.2.1 Adaptive Erweiterungen

Wie sich später herausstellen wird, sind nichtfunktionale Angebote, die zur Prüfung der Erfüllbarkeit nichtfunktionaler Anforderungen herangezogen werden, ebenfalls Veränderungen unterworfen. Veränderungen der nichtfunktionalen Angebote können im Laufe der Zeit in der Unerfüllbarkeit einer Anforderung resultieren. Folglich sollte auch betrachtet werden, inwieweit nichtfunktionale Anforderungen in Hinsicht auf Unerfüllbarkeit angepasst und somit der Aspekt der Adaption bereits bei der Beschreibung nichtfunktionaler Anforderungen berücksichtigt werden muss. Zu diesem Zweck wird angenommen, dass die von einer Komponente spezifizierten nichtfunktionalen Anforderungen nicht erfüllbar sind.

Im einfachsten Fall benachrichtigt die Verteilungsinfrastruktur eine Komponente darüber, dass eine gegebene nichtfunktionale Anforderung unerfüllbar ist. Als Konsequenz können einerseits weitere Methodenaufrufe mit den vorgesehenen nichtfunktionalen Anforderungen unterlassen werden und führen eventuell sogar zur Terminierung der Komponente und der mit ihrer Hilfe realisierten Anwendung. Andererseits besteht die Möglichkeit, dass die Ausführung einer Komponente trotz der derzeit widrigen Umstände unter Verwendung derselben unveränderten nichtfunktionalen Anforderung solange fortgeführt wird, bis Aufrufe unter den gegebenen Anforderungen wieder durchgeführt werden können. Dabei kann es jedoch dazu kommen, dass die Anforderungen selbst nach mehrfacher Wiederholung erst nach längerer Zeit oder auch gar nicht erfüllt werden können. Die daraus resultierenden Verzögerungen werden oftmals von den Benutzern einer Komponente nicht geduldet. Darüber hinaus führt dieses Verhalten je nach Häufigkeit der getätigten Methodenaufrufe zusätzlich zu einem enormen Nachrichtenverkehr zwischen Verteilungsplattform und der Komponente. Bei Mangel an lokalen Ressourcen kann dies sogar eine vollständige Auslastung einer Rechneinheit bedeuten. Um diesen Umstand zu vermeiden, sind geeignetere Alternativen erforderlich, die im Einzelnen die Wahrscheinlichkeit einer erfolgreichen Durchführung eines Methodenaufrufes erhöhen können. Prinzipiell können zwei Verhaltensweisen in Betracht gezogen werden: die künstliche Verzögerung des Aufrufes und die Reduktion der nichtfunktionalen Anforderung.

Verzögerte Durchführung (*Queued RPC*)

Gerade in Adhoc-Rechnernetzen ist häufig mit Veränderungen der aktuellen Umgebungsbedingungen zu rechnen. Vorallem die Bewegung einzelner Rechnerknoten führt zu solchen Veränderungen der Umgebungsbedingungen. Je nach

Veränderungen dieser kann sich dies unter Umständen positiv auf die Ausführung eines Aufrufes auswirken. Beispielsweise ist eine Rechneinheit nach einem kurzen Aufenthalt in einem Gebäude eventuell wieder erreichbar. In solchen Fällen ist eine künstliche Verzögerung mit Sicherheit sinnvoll. Allerdings hat die verzögerte Durchführung eines Aufrufes den Nachteil, dass sich die Umgebungsbedingungen nicht immer in absehbarer Zeit zum Positiven verändern und somit auch eine künstliche Verzögerung keine erfolgreiche Durchführung eines Aufrufes garantieren kann. Letztlich ist dies ausschließlich von den Umgebungsbedingungen abhängig und diese können nur schwer vorhergesagt werden.

Die Idee der verzögerten Ausführung stammt aus dem *Rover Toolkit* [15], dass dieselbe Funktionalität in Form von *Queued RPCs* zur Verfügung stellt.

Reduktion nichtfunktionaler Anforderungen

Eine Alternative zur Verzögerung von Aufrufen ist eine Reduktion der gestellten Anforderungen. Das bedeutet, dass die Bedürfnisgrade, die in einer Anforderung spezifiziert wurden, ausdrücklich herabgesetzt werden. Auf diese Weise steigt offensichtlich die Wahrscheinlichkeit, dass der Aufruf unter Beachtung der Anforderung erfolgreich durchgeführt werden kann. Komponenten müssen in diesem Fall jedoch einen qualitativen Verlust bei der Durchführung des Methodenaufrufes in Kauf nehmen. Sofern Komponenten mit einer solchen Situation umgehen können und in den jeweiligen Anwendungsszenarien eventuelle Qualitätsverluste für den Benutzer akzeptabel sind, ist dieses Verhalten vorteilhaft. Unter suboptimalen Bedingungen würde auf diese Weise der Betrieb einer Komponente nicht vollständig zum Erliegen kommen, sondern vielmehr im degradierten Modus zumindest die notwendigen Funktionen ausführen und dem Benutzer eine eingeschränkte Version der Anwendung präsentieren. Allerdings besteht abermals die Gefahr, dass es zu einem regen Nachrichtenverkehr zwischen Verteilungsplattform und Komponenten kommt, falls die Reduktionen nichtfunktionaler Anforderungen nur marginal ausfallen oder ausschließlich nichtfunktionale Eigenschaften reduziert werden, die derzeit in Bezug auf die Erfüllbarkeit keine ausschlaggebende Rolle spielen.

Gerade in Adhoc-Rechnernetzen, in denen sich die Umgebungsbedingungen stark ändern können, müssen adaptive Maßnahmen ergriffen werden. Sowohl die Reduktion als auch die verzögerte Durchführung sind durchaus praktisch brauchbare Möglichkeiten, um auf unerfüllbare Anforderungen zu reagieren. Folglich ist die Unterstützung dieser Mechanismen für die Erweiterung dieser Arbeit sinnvoll. Insofern stellt sich nun die Frage, auf welcher Ebene die

notwendigen Mechanismen am besten umgesetzt werden, da prinzipiell sowohl Komponenten als auch die Verteilungsinfrastruktur diese bereitstellen können.

Manuelle Verzögerung und Reduktion

Einerseits können sowohl die künstliche Verzögerung von Aufrufen als auch die Reduktion nichtfunktionaler Anforderungen vollständig auf Ebene der Komponenten realisiert werden. Das würde bedeuten, dass ausschließlich Komponenten diese Mechanismen umsetzen, indem diese erstens selbständig Aufrufe verzögern und zweitens nach eigenem Ermessen reduzierte Anforderungen bereitstellen, sofern die Unerfüllbarkeit einer vorangegangenen Anforderung durch die Verteilungsinfrastruktur festgestellt wurde.

Auf den ersten Blick scheint dies durchaus sinnvoll. Allerdings ist dieser Ansatz praktisch nicht realisierbar. Grund hierfür ist, dass bei Unerfüllbarkeit der Anforderung einer Komponente nicht eindeutig bekannt ist, welche nichtfunktionalen Eigenschaften tatsächlich zur Unerfüllbarkeit geführt haben und wie diese angepasst werden müssten, um eine erfüllbare Anforderung zu erhalten. Eine Aussage der Verteilungsinfrastruktur darüber ist aus praktischer Sicht nicht möglich. Das liegt daran, dass bezüglich der Erfüllbarkeit einer Anforderung prinzipiell die Kombination aller nichtfunktionalen Eigenschaften und ihrer Ausprägungen ausschlaggebend ist. Um die Erfüllbarkeit einer Anforderung herzustellen, kann prinzipiell jede einzelne Eigenschaft oder auch eine Untermenge an Eigenschaften relevant sein. Die Reduktion einer einzigen Eigenschaft führt eventuell zu größeren Freiheitsgraden einer oder mehrerer anderer Eigenschaften. Eine Betrachtung aller möglichen Veränderungen einzelner oder mehrerer Eigenschaften bedeutet einen kombinatorischen Anstieg der Anzahl an möglichen Reduktionen. Allein die Feststellung dieser vielzähligen Kombinationen kann sehr aufwendig sein. Die Mitteilung dieser an die entsprechende Komponente würde den Aufwand zusätzlich erhöhen. Eine Beschränkung auf einige wenige Kombinationen würde wiederum mit hoher Wahrscheinlichkeit dazu führen, dass gerade die getroffene Auswahl nicht den Wünschen einer Komponente entspricht. Selbst wenn der resultierende Aufwand in Kauf genommen wird, wäre mit einem weiteren Problem zu rechnen. Zwischen der Bereitstellung einer reduzierten Anforderung durch die Verteilungsinfrastruktur und deren Verwendung durch eine Komponente besteht ein gewisses Zeitfenster, indem sich die Umgebungsbedingungen derart verändern können, dass unter Umständen die reduzierte Anforderung zum Zeitpunkt des Methodenaufrufes nicht mehr akkurat ist.

Automatische Verzögerung und Reduktion

Als Alternative zur manuellen Verzögerung von Aufrufen und der Reduktion nichtfunktionaler Anforderungen kann auch die Verteilungsinfrastruktur diese Aufgabe übernehmen. Hierzu ist es allerdings erforderlich, dass zum einen Komponenten, sofern diese nur befristet auf die Ausführung eines Methodenaufrufes warten können, die Dauer der Verzögerung vorgeben und zum anderen die Reduktion der Anforderungen gewissermaßen der Verteilungsinfrastruktur überlassen. Sofern die Verteilungsinfrastruktur in der Lage ist, automatisch geeignete Zeitpunkte zu erkennen, kann dies wesentlich effektiver sein als eine manuelle Verzögerung von Aufrufen.

Ein Ansatz zur automatischen Reduktion nichtfunktionaler Anforderungen wird bereits in der Literatur beschrieben. Es handelt sich dabei um die sogenannten *Windows of Expectation* [30], auch *Windows of Tolerance* [23] genannt. Dieser Ansatz sieht vor, dass eine nichtfunktionale Anforderung zu jeder nichtfunktionalen Eigenschaft zwei Ausprägungen festlegt: einen minimalen Wert, der bei der Erfüllung der Anforderung in keinem Fall unterschritten werden darf und ein optionaler Wert, dessen Erfüllung von der Komponente erwünscht wird, aber für die Ausführung einer Komponente nicht kritisch ist. Auf diese Weise erhält die Verteilungsinfrastruktur zusätzliche Freiheitsgrade, da diese bei Unerfüllbarkeit einer Anforderung innerhalb der festgelegten Wertebereiche der Eigenschaften die jeweiligen Ausprägungen abschwächen kann. Auch hier ist es natürlich eine Voraussetzung, dass eine Komponente unter Umständen im degradierten Modus fortfahren kann. Da die Verteilungsinfrastruktur prinzipiell nicht ohne weiteres entscheiden kann, in welcher Reihenfolge die nichtfunktionalen Eigenschaften reduziert werden müssten, ist zu diesem Zweck über die *Windows of Expectations* hinaus eine zusätzliche Prioritätenliste denkbar, die von Komponenten vorgegeben und der Verteilungsinfrastruktur mitgeteilt wird.

Die Diskussion der beschriebenen Möglichkeiten zeigt deutlich, dass letztlich nur die automatische Umsetzung künstlicher Verzögerungen und reduzierter Anforderungen praktisch geeignet ist. Die Auswirkungen für die vorliegende Arbeit sind, dass die beschriebenen *Windows of Expectations* einschließlich einer Prioritätenliste in die Beschreibung nichtfunktionaler Anforderungen zusätzlich integriert werden.

Kapitel 6

Erfüllung von Anforderungen

Da bereits diskutiert wurde, auf welcher Ebene die Auswahl geeigneter Plugins idealerweise realisiert wird und auf welche Weise hierfür die Anforderungen von Komponenten formal erfasst werden, können nun Betrachtungen angestellt werden, die im Einzelnen auf die Umsetzung der nichtfunktionalen Anforderungen eingehen. Dazu werden zunächst die Aufgaben von Selektionsstrategien und weitere notwendige Informationen genauer bestimmt. Anschließend sollte in einem weiteren Schritt betrachtet werden, inwiefern Selektionsstrategien über mehrere Instanzen der Verteilungsinfrastruktur hinweg zur Umsetzung nicht-funktionaler Anforderungen kooperieren müssen.

6.1 Selektionsstrategien

Wie bereits in Abschnitt 4.1.4 angedeutet wurde, dienen Selektionsstrategien primär der Selektion von Plugins. Die Auswahlkriterien für eine Selektion von Plugins sind folgende:

- Interoperabilität zum Kommunikationspartner
- Erfüllung einer gegebenen nichtfunktionalen Anforderung durch die Eigenschaften der ausgewählten Plugins

Auf diese Punkte wird in den folgenden Abschnitten näher eingegangen.

6.1.1 Interoperabilität

Sowohl auf der lokalen als auch auf der entfernten Rechneinheit sind jeweils eine gewisse Menge an Plugins verfügbar, die zur Installationszeit eingebunden und während der Laufzeit zur Kommunikation verwendet werden können. Damit die Kommunikation zwischen beiden Rechnern tatsächlich stattfinden kann,

ist es erforderlich, dass entsprechend der Plugin-Architektur beiden Instanzen der Verteilungsinfrastruktur interoperable Plugins zur Verfügung stehen, die jeweils eine vollständige Plugin-Kette bilden. Die Verwendung interoperabler Plugins auf beiden Seiten ist offensichtlich Voraussetzung für die erfolgreiche Durchführung von Methodenaufrufen. Da eine Selektionsstrategie innerhalb der Verteilungsinfrastruktur realisiert wird, ist der direkte Zugriff auf ein Verzeichnis möglich, das in der aktuellen Implementierung von BASE bereits existiert und Informationen darüber zur Verfügung stellt, welche Plugins sowohl auf lokaler als auch auf entfernter Seite vorhanden sind. Dieses Verzeichnis wird in regelmäßigen Abständen periodisch aktualisiert und ermöglicht die Bestimmung der notwendigen Interoperabilitätsmenge.

6.1.2 Nichtfunktionale Teilangebote

Um feststellen zu können, ob eine nichtfunktionale Anforderung durch einen bestimmten Satz an Plugins erfüllt wird, ist es notwendig, dass diese im Einzelnen explizit eine bestimmte Menge an Eigenschaften unterstützen, um die jeweils erfüllbaren Bedürfnisgrade ihrer Eigenschaften in Erfahrung bringen zu können. Die Menge der Eigenschaften, die von einem einzelnen Plugin unterstützt wird, und die zugehörigen Bedürfnisgrade werden als *nichtfunktionales Teilangebot* bezeichnet. Mit nichtfunktionalen Teilangeboten als Grundlage können sukzessiv über mehrere Plugins hinweg vollständige nichtfunktionale Angebote erzeugt und in Bezug auf eine nichtfunktionale Anforderung auf die Erfüllbarkeit dieser geprüft werden.

Die Erhebung nichtfunktionaler Teilangebote ist, wie im folgenden Abschnitt aufgezeigt wird, nicht trivial. Aus diesem Grund müssen Möglichkeiten, wie nichtfunktionale Teilangebote praktischerweise erhoben und zu vollständigen nichtfunktionalen Angeboten zusammengeführt werden können, gefunden und untersucht werden.

6.1.3 Veränderlichkeit nichtfunktionaler Teilangebote

In Abschnitt 5.2 wurde aufgezeigt, dass nichtfunktionale Anforderungen oftmals einer gewissen Veränderlichkeit unterworfen sind. Dasselbe trifft in wesentlich größerem Ausmaß auch auf nichtfunktionale Teilangebote und die (aus diesen zusammengeführten) nichtfunktionale Angebote zu. Die Ursache für die hohe Veränderlichkeit nichtfunktionaler Teilangebote ist, dass die erfüllbaren Bedürfnisgrade einzelner Eigenschaften von unzähligen Faktoren abhängen können, die zur beschriebenen Veränderlichkeit während der Laufzeit maßgeblich beitragen.

Die Verfügbarkeit lokaler Ressourcen ist einer der ausschlaggebenden Faktoren. Entscheidend sind diesbezüglich der Prozessor, die Stromversorgung, die Netzwerkschnittstellen und der verfügbare Hauptspeicher einer Rechneinheit. Beispielsweise wird die nichtfunktionale Eigenschaft *Latenz* eines Kompressionsplugins durch eine ungleichmäßige Prozessorbelastung beeinflusst. Diese Eigenschaft unterliegt auch Veränderungen, wenn ein Plugin zur Übertragung von Nachrichten eine Netzwerkschnittstelle verwendet, die in unregelmäßigem Umfang bereits andere Daten überträgt oder empfängt. Weitere Abhängigkeiten bestehen zwischen den einzelnen Ressourcen. Ist der Prozessor permanent ausgelastet, führt dies unter Umständen zu einer schwachen Stromversorgung, falls das Gerät durch Batterien betrieben wird. Andererseits kann der Prozessor nur mit maximaler Geschwindigkeit arbeiten, falls ausreichend Energie zur Verfügung steht. Diese ressourcenbasierten Abhängigkeiten führen automatisch zu indirekten Abhängigkeiten zwischen nichtfunktionalen Eigenschaften, die dieselben Ressourcen verwenden. Diese Tatsache ist beispielsweise offensichtlich, wenn man die Eigenschaften *Verarbeitungsgeschwindigkeit* und *Energieverbrauch* eines Kompressionsplugins betrachtet.

Neben der Verfügbarkeit lokaler Ressourcen sind Art und Umfang der getätigten Aufrufe relevant. Interaktionsmuster wie *Request/Response* implizieren, dass beispielsweise der Energieverbrauch von Plugins auf Grund des Empfangs der Antwort höher ausfällt, als wenn Nachrichten lediglich in eine Richtung versendet werden. Die Anzahl der Parameter und die Komplexität der Aufruf- und Ergebnisparameter sind ebenfalls Einflussfaktoren.

Sogar die Plugin-Selektion selbst, die einerseits durch die getroffene Auswahl und Verwendung von bestimmten Plugins Einfluss auf die Struktur der zu versendenden Nachrichteninhalte nimmt und andererseits selbst gewisse Ressourcen für sich in Anspruch nehmen muss, kann zur Veränderlichkeit nichtfunktionaler Eigenschaften beitragen. Basierend auf den nichtfunktionalen Eigenschaften einzelner Plugins entstehen übergreifende Abhängigkeiten zwischen mehreren Konzeptionsschichten. Da die Nachrichteninhalte bei jedem Verarbeitungsschritt innerhalb der Plugin-Architektur je nach verwendetem Plugin eine inkrementelle Veränderung ihrer Struktur und Inhaltes erfahren, ist die Kombination der verwendeten Plugins ebenfalls ein ausschlaggebender Faktor für die erfüllbaren Bedürfnisgrade nichtfunktionaler Eigenschaften.

Die Erhebung nichtfunktionaler Teilangebote ist auf Grund der beschriebenen Abhängigkeiten äußerst schwierig und unter Beachtung aller Abhängigkeiten praktisch nicht möglich. Folglich müssen die jeweils erfüllbaren Bedürfnisgrade erstens auf Grund der Veränderlichkeit zur Laufzeit und zweitens sowohl effek-

tiv als auch in Hinsicht auf ressourcenarme Rechnersysteme effizient bestimmt werden. Insgesamt lassen sich nichtfunktionale Eigenschaften in die folgenden Kategorien untergliedern:

1. invariant (Festlegung zur Entwicklungszeit)
2. dynamisch invariant (Festlegung während der Administration)
3. dynamisch variant (Festlegung zur Laufzeit)

Die Art der Veränderlichkeit wird grundlegend durch den Zeitpunkt der Zuweisung eines Bedürfnisgrades festgelegt. Wird der Bedürfnisgrad einer Eigenschaft zur Entwicklungszeit zugewiesen, handelt es sich um eine *invariante* Eigenschaft. In diesem Fall ist eine nachträgliche Wertzuweisung nach Ablauf dieser Phase nicht mehr möglich. Die zweite Phase beschreibt den Zeitraum, in der die Implementierung in das existierende System eingebunden und bei Bedarf konfiguriert wird. Nichtfunktionale Eigenschaften, deren Bedürfnisgrade in dieser Phase festgelegt werden, sind *dynamisch invariant*, da Veränderungen dieser zwar nicht zur Laufzeit, aber noch spätestens zum Zeitpunkt der Installation durchgeführt werden können. Die verbleibende Menge der nichtfunktionalen Eigenschaften ist *variant*. Das bedeutet, dass die erfüllbaren Bedürfnisgrade dieser zur Laufzeit angepasst werden können und somit zu jeder Zeit veränderlich sind.

6.1.4 Erhebung nichtfunktionaler Teilangebote

Gerade auf Grund der starken Veränderlichkeit nichtfunktionaler Teilangebote ist es notwendig, einen effektiven und effizienten Mechanismus zur Erhebung dieser zu entwickeln. Sowohl für statische als auch dynamisch invariante nichtfunktionale Eigenschaften sind angemessene Bedürfnisgrade zumindest in Hinsicht auf die Veränderlichkeit verhältnismäßig einfach festzulegen. Da die Werte dieser Eigenschaften nicht zur Laufzeit variieren, müssen angemessene Werte nur ein einziges Mal erhoben oder berechnet werden. Die festgestellten Werte können prinzipiell entweder bereits in der Implementierung einer Selektionsstrategie oder verteilt in allen Plugins fixiert werden, falls es sich im Einzelnen um eine statische Eigenschaft handelt und der jeweilige Wert bereits zur Entwicklungszeit bekannt ist, oder falls es sich jeweils um eine dynamisch invariante Eigenschaft handelt, während der Konfiguration der Verteilungsinfrastruktur durch einen Administrator zugewiesen werden. Schwieriger und wesentlich aufwendiger ist die Erfassung dynamisch varianter Eigenschaften. Da diese stark veränderlich sind, ist es erforderlich, dass diese erst zur Laufzeit bestimmt werden. Die Feststellung der nichtfunktionalen Teilangebote kann auf zwei, im Folgenden beschriebenen, unterschiedlichen Wegen geschehen.

Monitoring

Eine Möglichkeit zur Erfassung dynamischer Eigenschaften ist die Erhebung der jeweils erfüllbaren Bedürfnisgrade durch unterstützte Messverfahren (engl. *Monitoring*). Beispielsweise können Prozessor, Speicher- und Netzwerklast in jedem modernen Betriebssystem dynamisch über eine API erfragt werden. Mit den dadurch gewonnenen Informationen kann eine Art „Lernendes System“ entwickelt werden, das mehr oder weniger genaue Werte feststellt und somit brauchbare nichtfunktionale Angebote ermöglicht.

Abschätzfunktionen

Eine weitere Möglichkeit ist die automatisierte Wertzuweisung mit Hilfe von Abschätzfunktionen, die für jedes Plugin die aktuellen Werte einer vorgegebenen Menge nichtfunktionaler Eigenschaften schätzt. Die Abschätzfunktion eines Plugins erhält als Eingabe sowohl einen Hinweis auf die abzuschätzende nichtfunktionale Eigenschaft als auch eine Menge von Faktoren, die jeweils eine bestimmte Abhängigkeit repräsentieren und bei der Abschätzung je nach Bedarf berücksichtigt werden können oder auch nicht. Als Resultat wird ein Schätzwert zurückgeliefert, der dem tatsächlich erfüllbaren Bedürfnisgrad möglichst nahe kommt und von Selektionsstrategien für weitere Analysen verwendet werden kann.

Einige Nachteile machen den Monitoring-Ansatz nur bedingt brauchbar. Abgesehen davon, dass dieser Ansatz nur für tatsächlich meßbare Eigenschaften eingesetzt werden kann, muss erstens zum Zeitpunkt der Erhebung bereits ein gewisser Nachrichtenverkehr stattgefunden haben, um brauchbare Werte überhaupt feststellen zu können. Dies ist allerdings gerade zu Beginn der Kommunikation zwischen Komponenten nicht der Fall und führt zwangsläufig dazu, dass auf künstlichem Wege ein gewisser Nachrichtenverkehr zwischen Komponenten erzeugt werden muss. Auf diese Weise werden jedoch Ressourcen belastet, die unter Umständen irreversibel und kostbar sind ¹. Zweitens müssen zur Laufzeit die gesammelten Informationen der vergangenen Zeit kurzfristig zwischengespeichert werden, was bedeutet, dass zusätzliche Speicherkapazitäten auf einer Rechneinheit erforderlich sind. Darüber hinaus ist nicht bekannt, wie lange die erhobenen Informationen akkurat sind. Je länger eine nichtfunktionale Eigenschaft nicht aktualisiert wird, desto größere Ungenauigkeiten können auftreten. Um inakzeptable Ungenauigkeiten zu vermeiden, müsste periodisch und

¹z.B. die Energieversorgung mobiler Rechner

in relativ kurzen Abständen gemessen werden. Infolgedessen werden abermals lokale Ressourcen in Anspruch genommen.

Abschätzfunktionen sind unter Umständen aufwendig zu entwickeln. Da die Implementierungen der Plugins und deren realisiertes Verhalten einem Strategieentwickler nicht ohne weiteres bekannt sind, müssen von diesem zunächst nähere Untersuchungen der Implementierung und der Einflussfaktoren angestellt werden. Anschließend muss dementsprechend für jedes Plugin und jede nichtfunktionale Eigenschaft eine Abschätzfunktion erstellt werden, die später von einer Selektionsstrategie genutzt werden kann.

Sowohl der Monitoring-Ansatz als auch die Verwendung von Abschätzfunktionen erfordern einen Aufwand, der vom Strategieentwickler erbracht werden muss. Monitoring kann in Verbindung mit beliebigen Eigenschaften nicht immer eingesetzt werden, ist in hoch dynamischen Umgebungen eventuell ungenau oder für einige Eigenschaften wie beispielsweise Zuverlässigkeit nicht einsetzbar. In manchen Fällen jedoch können bereits einfache Routinen brauchbare Ergebnisse liefern. Letzteres trifft auch auf Abschätzfunktionen zu. Allerdings sind ausgefeilte Abschätzfunktionen, die möglichst viele Abhängigkeiten einer Eigenschaft berücksichtigen, schwieriger zu realisieren. Da sich sowohl Monitoring als auch Abschätzfunktionen prinzipiell nicht widersprechen, ist es im Rahmen dieser Arbeit sinnvoll, beide Techniken zu unterstützen und die Auswahl dem Strategieentwickler zu überlassen. Prinzipiell ist sogar der gleichzeitige Einsatz beider Techniken denkbar. Da letztlich die Implementierung der notwendigen Funktionen durch den Strategieentwickler bereitgestellt wird und die Tatsache, ob es sich dabei im Einzelnen um Abschätz- oder Monitoringfunktion handelt, kaum Auswirkungen auf das Rahmenwerk hat, müssen keine weiteren Vorkehrungen zur Unterstützung beider Techniken getroffen werden. Lediglich die Bereitstellung ausreichender Informationen über Abhängigkeitsfaktoren muss zur Realisierung der Abschätzfunktionen sichergestellt werden.

Um die eigentliche Funktionalität des Plugins nicht mit der Implementierung von Abschätzfunktionen oder Monitoring-Routinen zu vermischen und den flexiblen Austausch von Selektionsstrategien zu ermöglichen, ist eine saubere Trennung der jeweiligen Funktionalitäten natürlich vorteilhaft.

6.1.5 Feststellung nichtfunktionaler Angebote

Auf der Basis der inzwischen verfügbaren nichtfunktionalen Teilangebote können nun weiterführende Betrachtungen angestellt werden. Wie bereits erwähnt, können aus nichtfunktionalen Teilangeboten vollständige nichtfunktionale Angebote hergeleitet werden. Auf Grund dessen, dass abgesehen von anderen Fak-

toren dynamische Eigenschaften auch von der konkreten Auswahl von Plugins abhängen, können prinzipiell mehrere nichtfunktionale Angebote erstellt werden, die jeweils in Abhängigkeit der Kombination an Plugins unterschiedliche Werte ihrer Eigenschaften aufweisen. Es sei an dieser Stelle auch darauf hingewiesen, dass gerade wegen der Verfügbarkeit mehrerer Teilangebote und der Kombination dieser erheblich mehr nichtfunktionale Angebote erstellt werden können als im Vergleich zur originären Plugin-Architektur. Dadurch steigt die Wahrscheinlichkeit, dass ein nichtfunktionales Angebot gefunden werden kann, welches eine nichtfunktionale Anforderung später erfüllt.

Im Folgenden wird nun betrachtet, wie nichtfunktionale Angebote aus nicht-funktionalen Teilangeboten erhoben werden können. Dazu wird genauer untersucht, wie im Einzelnen die jeweils enthaltenen Eigenschaften der Teilangebote und ihre Bedürfnisgrade vereinigt werden. Zu diesem Zweck sind maßgeblich zwei Informationen notwendig, die durch den Strategieentwickler implizit in der Implementierung einer Strategie festgelegt werden müssen.

Aggregationsfunktionen

Da sich vertikale Eigenschaften über den gesamten Kommunikationspfad erstrecken, müssen zur Bestimmung aussagekräftiger Bedürfnisgrade eines Angebots die einzelnen Bedürfnisgrade der erhobenen Teilangebote aggregiert werden. Um dies zu bewerkstelligen ist somit eine spezifische *Aggregationsfunktion* erforderlich, die festlegt, auf welche Weise die von mehreren Plugins erhobenen Bedürfnisgrade zu einem endgültigen und aussagekräftigen Gesamtwert vereinigt werden. Eine solche Aggregationsfunktion ist natürlich vom Verwendungszweck der betrachteten Eigenschaft abhängig und muss deshalb vom Strategieentwickler in der Implementierung einer Selektionsstrategie festgelegt werden.

Ein anschauliches Beispiel für die Aggregation einzelner Bedürfnisgrade ist die Erhebung der nichtfunktionalen Eigenschaft *Latenz*. Eine sinnvolle Aussage kann nur dann gemacht werden, wenn die einzelnen Latenzen der betrachteten Plugins durch eine Addition als Aggregationsfunktion vereinigt werden.

Referenz von Plugin-Schichten

Eine Aggregationsfunktion ist zwar für vertikale Eigenschaften erforderlich, auf horizontale Eigenschaften trifft dies jedoch nicht zu. Das liegt daran, dass horizontale Eigenschaften lediglich in einer einzigen Plugin-Schicht unterstützt werden müssen und somit der erfüllbare Bedürfnisgrad eines nichtfunktionalen Angebotes exakt dem erfüllbaren Bedürfnisgrad des nichtfunktionalen Teilangebotes eines der Plugins dieser Schicht entspricht. Folglich ist anstatt einer

Aggregationsfunktion vielmehr ein Hinweis auf die relevante Plugin-Schicht erforderlich. Auch diese Information muss durch den Strategieentwickler in der Implementierung vorgegeben werden. Um eine möglichst große Auswahl an Plugins in dieser Schicht zu erreichen, ist es natürlich sinnvoll, in die nichtfunktionalen Teilangebote aller Plugins dieser Schicht die entsprechende horizontale Eigenschaft zu integrieren.

6.1.6 Erfüllbarkeit nichtfunktionaler Anforderungen

Da nun die Feststellung einzelner nichtfunktionaler Angebote möglich ist, können diese zur Prüfung auf Erfüllbarkeit einer gegebenen nichtfunktionalen Anforderung herangezogen werden. Die Erfüllbarkeit einer nichtfunktionalen Anforderung durch ein nichtfunktionales Angebot ist wie folgt definiert:

Erfüllbarkeit Eine nichtfunktionale Eigenschaft einer Anforderung ist genau dann erfüllt, wenn die nichtfunktionale Eigenschaft auch im Angebot enthalten ist und der Bedürfnisgrad der nichtfunktionalen Eigenschaft aus dem Angebot, unter Beachtung der Ausrichtung, eventuell erforderlicher Aggregation und festgelegter Vergleichsoperatoren, mindestens gleich oder besser ausgeprägt ist, als der Bedürfnisgrad derselben Eigenschaft aus der Anforderung.

Eine nichtfunktionale Anforderung wird genau dann durch ein nichtfunktionales Angebot erfüllt, wenn alle Eigenschaften der Anforderung durch das nichtfunktionale Angebot erfüllt werden.

Folgendes Beispiel verdeutlicht den Begriff der Erfüllbarkeit: In einem nichtfunktionalen Angebot der Verteilungsinfrastruktur wird die Eigenschaft *Energieverbrauch* einer Plugin-Kette explizit mit 200mWsek spezifiziert. Eine nichtfunktionale Anforderung einer Komponente enthält dieselbe Eigenschaft mit der Ausprägung 300mWsek und absteigender Ausrichtung. In diesem Fall ist die Anforderung durch das Angebot erfüllbar. Umfangreichere Angebote und Anforderungen mit mehreren Eigenschaften sind entsprechend konstruierbar.

6.1.7 Interoperable Erfüllbarkeit

Da die Erfüllbarkeit einer Anforderung durch ein oder mehrere nichtfunktionale Angebote allein nicht ausreicht, sondern auch die Interoperabilität zur entfernten Instanz der Verteilungsinfrastruktur sichergestellt werden muss, muss eine Selektionsstrategie sowohl die interoperable Menge an Plugins als auch diejenigen Mengen an Plugins bestimmen, die eine nichtfunktionale Anforderung erfüllen. Vorteilhafterweise wird zuerst die Menge der interoperablen Plugins

bestimmt, da dadurch die Menge der relevanten, erfüllbaren nichtfunktionalen Angebote von vornherein eingeschränkt wird.

Alle nichtfunktionalen Angebote, die die Anforderung jeweils erfüllen, könnten nun für die Durchführung eines Aufrufes verwendet werden. Da allerdings nur ein Angebot tatsächlich für die Durchführung eines Aufrufes eingesetzt werden kann, muss die Selektionsstrategie eine abschließende Entscheidung darüber treffen, welches Angebot und welche Plugins letztlich zum Einsatz kommen sollen. Diese Menge an Plugins wird als *finale Menge* bezeichnet.

Wird in einem ersten Schritt kein erfüllendes Angebot gefunden, kann mit Hilfe der gegebenen Prioritätenliste aus der Anforderung letztere Schritt für Schritt reduziert werden, bis entweder ein erfüllendes Angebot gefunden wird oder aber die unteren Grenzen der Window of Expectations erreicht werden und keine Freiheitsgrade mehr existieren. Letzteres führt dazu, dass die Durchführung des Aufrufes unter Einhaltung der gegebenen Anforderung nicht möglich ist. Dies wird der aufrufenden Komponente mitgeteilt, damit diese entsprechend auf die Unerfüllbarkeit der Anforderungen reagieren kann.

Je nach Selektionsstrategie können unterschiedliche Vorgehensweisen realisiert werden. Beispielsweise ist nicht von vornherein festgelegt, ob eine vollständige Betrachtung aller Kombinationsmöglichkeiten durchgeführt wird, um eine optimale Menge an Plugins zu finden, oder ob die Analyse vorzeitig beendet werden kann, sofern eine Plugin-Kette ermittelt wurde, die die minimale Anforderung mindestens erfüllt. Da diese Fragen von den jeweiligen Anwendungsszenarien abhängen und somit nur durch den Strategieentwickler entschieden werden können, legt letztlich die Implementierung einer Selektionsstrategie diese Punkte fest.

6.2 Kooperierende Selektionsstrategien

Je nach Art der Aufrufe und Lokalität der Eigenschaften einer nichtfunktionaler Anforderung muss unter Umständen auch die entfernte Instanz der Verteilungsinfrastruktur miteinbezogen werden, da diese ebenfalls Aufrufe verarbeitet und deshalb auf der Basis ihrer nichtfunktionalen Angebote auch für die Erfüllung einer nichtfunktionalen Anforderung mitverantwortlich ist. Zunächst werden die Voraussetzungen für eine Kooperation zwischen Selektionsstrategien diskutiert. Anschließend werden die möglichen Aufrufszszenarien betrachtet, in denen die Kooperation zwischen Selektionsstrategien unabdingbar ist und untersucht, wie eine Kooperation sowohl effektiv als auch effizient realisiert werden kann.

6.2.1 Äquivalenz zwischen Selektionsstrategien

Offensichtlich können Selektionsstrategien, die nicht dieselben nichtfunktionalen Eigenschaften unterstützen, nicht miteinander kooperieren. Das schließt zwar nicht die Kommunikation zwischen Komponenten der beteiligten Instanzen der Verteilungsinfrastruktur aus, allerdings kann sofern eine Kooperation zur Erfüllbarkeit einer nichtfunktionalen Anforderung notwendig ist, letztere praktisch nicht erfüllt werden. Damit eine Kooperation zwischen Selektionsstrategien auf der Basis gemeinsamer, nichtfunktionaler Angebote möglich ist, wird eine entsprechende und angemessene Äquivalenz zwischen Selektionsstrategien definiert.

Definition Domänen-Äquivalenz Zwei Selektionsstrategien sind genau dann *domänen-äquivalent*, wenn sie dieselbe Menge nichtfunktionaler Eigenschaften unterstützen.

Da nichtfunktionale Eigenschaften eindeutig über ihre Kennung beziehungsweise ihren Namen unterschieden werden können, zeigt ein Vergleich der Kennungen, ob zwei Selektionsstrategien dieselben Eigenschaften unterstützen oder nicht. Die Eigenschaft der Domänen-Äquivalenz garantiert lediglich, dass die beteiligten Selektionsstrategien zumindest die verwendeten nichtfunktionalen Eigenschaften behandeln können und somit zur Unterstützung nichtfunktionaler Eigenschaften derselben Domäne konzipiert wurden.

Eine weitere Äquivalenz muss definiert werden, da zwar verschiedene Selektionsstrategien domänen-äquivalent sein können, es aber dennoch möglich ist, auf Grund menschlicher Entscheidungen einzelnen Eigenschaften unterschiedliche Bedürfnisgrade innerhalb der nichtfunktionalen Angebote zuzuordnen. Zum Beispiel ist nicht allgemein bekannt, ob ein bestimmtes Plugin zur Verschlüsselung von Nachrichten als *sicher* oder sehr *sicher eingestuft* wird. Da solche Inkonsistenzen zu falschen Entscheidungen der Selektionsstrategien führen, muss die sogenannte *Zuweisungs-Äquivalenz* für eine sinnvolle Kooperation erfüllt werden.

Definition Zuweisungs-Äquivalenz Zwei Selektionsstrategien sind genau dann *zuweisungs-äquivalent*, wenn die Zuweisung von Bedürfnisgraden einzelner Eigenschaften gleich behandelt wird.

Allerdings reicht auch die Zuweisungs-Äquivalenz noch nicht aus, dass nichtfunktionale Anforderungen durch beide Selektionsstrategien auf dieselbe Weise unterstützt werden. Die *Erfüllbarkeits-Äquivalenz* schließt die Domänen-Äquivalenz und Zuweisungs-Äquivalenz ein und garantiert, dass trotz eventuell

unterschiedlicher Implementierungen zwei Selektionsstrategien unter den selben Voraussetzungen (Angebote und Anforderung) stets zum selben Resultat kommen.

Definition *Erfüllbarkeits-Äquivalenz* Zwei Selektionsstrategien sind genau dann *erfüllbarkeits-äquivalent*, wenn diese domänen-äquivalent sind und für alle beliebigen nichtfunktionalen Anforderungen und Angebote auf der Basis derselben Plugin-Menge dieselbe Plugin-Kette bestimmen.

Zur weiteren Betrachtung wird davon ausgegangen, dass Selektionsstrategien erfüllbarkeits-äquivalent sind.

6.2.2 Kooperation

Die Notwendigkeit einer Kooperation zwischen Selektionsstrategien hängt maßgeblich davon ab, ob eine nichtfunktionale Anforderung globale Eigenschaften enthält und welches Interaktionsmuster während der Kommunikation angewendet werden soll. Zwei Selektionsstrategien müssen genau dann kooperieren, wenn:

1. eine globale Eigenschaft in einer Anforderung enthalten ist oder
2. eine Ergebnisantwort für einen Aufruf erwartet wird.

Zur Erläuterung beider Fälle wird im Folgenden aus Sicht einer lokalen Selektionsstrategie argumentiert, die aus ihrem Blickwinkel mit einer entfernten Selektionsstrategie kommuniziert. Abbildung 6.1 zeigt den Sachverhalt.

Der erste Fall rührt daher, dass eine Anforderung mit globalen Eigenschaften per Definition sowohl durch die lokale als auch die entfernte Selektionsstrategie erfüllt werden muss. Da die lokale Selektionsstrategie durch die konkrete Auswahl interoperabler Plugins die entfernte Selektionsstrategie bei der Erfüllung der gegebenen Anforderung einschränkt und dadurch die Erfüllbarkeit auf entfernter Seite nicht verhindert werden darf, muss die lokale Selektionsstrategie neben ihren eigenen Angeboten auch die Angebote der entfernten Selektionsstrategie kennen. Nur so lässt sich entscheiden, welche Plugins auf beiden Seiten in Hinsicht auf Interoperabilität und Erfüllbarkeit der Anforderung eingesetzt werden können.

Der zweite Fall lässt sich anhand der Tatsache herleiten, dass die Verarbeitung eines Aufrufes und die Erzeugung einer entsprechenden Antwort durch die entfernte Selektionsstrategie einen ausschlaggebenden Einfluss auf die Erfüllbarkeit einer nichtfunktionalen Anforderung durch die lokale Selektionsstrategie

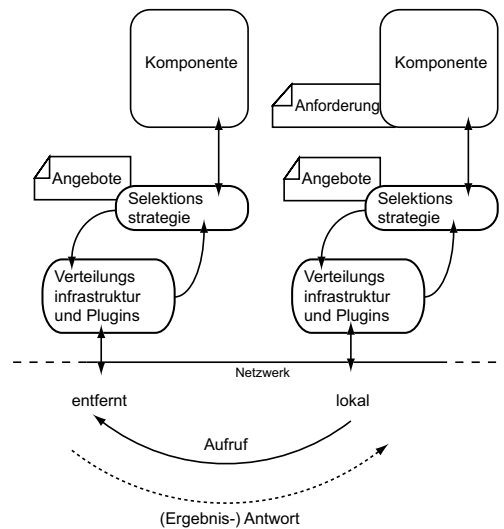


Abbildung 6.1: Kooperation zwischen Selektionsstrategien

hat. Die Verwendung bestimmter Plugins für den Versand der erwarteten Antwort auf Seite der entfernten Selektionsstrategie impliziert die Verwendung entsprechend interoperabler Plugins auf lokaler Seite beim Empfang und schränkt somit die Möglichkeiten bei der Plugin-Selektion auf lokaler Seite ein. Unter Umständen kann diese Einschränkung bereits dazu führen, dass die beidseitige Erfüllbarkeit der Anforderung nicht mehr möglich ist. Insofern müssen auf Seite der entfernten Selektionsstrategie die aktuellen Angebote und die Anforderung der lokalen Selektionsstrategie bekannt sein, damit die entfernte Selektionsstrategie eine angemessene und vorausschauende Plugin-Selektion durchführen kann.

Offensichtlich können lediglich Anforderungen, die sich auf einen einfachen Aufruf ohne Ergebnisantwort beziehen und keine globalen Eigenschaften enthalten, vollständig und ohne weiteres durch die lokale Selektionsstrategie erfüllt werden. Die beschriebenen Fälle, in denen die Erfüllung einer Anforderung unter Mitverantwortlichkeit der entfernten Selektionsstrategie erfolgt, erfordert zwangsläufig die Kooperation dieser.

Eine Notwendigkeit, die sich aus der Kooperation von Selektionsstrategien ergibt, ist, dass beide Selektionsstrategien dasselbe Verhalten bei der Selektion von Plugins an den Tag legen. Dies ist deshalb erforderlich, weil eine Selektionsstrategie an Hand eines entfernten Angebotes auf die spätere Selektion an Plugins durch die entfernte Selektionsstrategie schließt. Diese Schlussfolgerung kann nur genau dann korrekt sein, wenn beide Selektionsstrategien erfüllungs-äquivalent sind.

6.2.3 Aushandlung nichtfunktionaler Angebote

Da zur Erfüllung nichtfunktionaler Anforderungen unter Umständen die möglichen nichtfunktionalen Angebote der jeweils anderen Selektionsstrategie bekannt sein müssen, stellt sich die Frage, wie und wann diese Informationen am besten übermittelt werden können und ob es nicht eine Alternative gibt, die zusätzliche Kommunikationskosten vermeidet.

Explizite Aushandlung

Gerade in reservierungsbasierten Systemen wird häufig vor der eigentlichen Kommunikation eine einmalige Aushandlung durchgeführt, in der auf der Basis nichtfunktionaler Anforderungen und Angebote eine Übereinkunft über die tatsächlich zu realisierenden nichtfunktionalen Eigenschaften getroffen wird. Diese Übereinkunft wird möglichst von den Beteiligten während der gesamten Zeit einer Bindung aufrecht erhalten und gilt somit für sämtliche Nachrichten, die während dieser Zeit versendet werden. Allerdings kann es selbst in klassischen Rechnernetzen vorkommen, dass wegen verschiedener Einflussfaktoren wie beispielsweise Ressourcenverlusten das ursprüngliche Angebot nicht mehr eingehalten werden kann und die Übereinkunft ungültig wird. Auch eine Änderung der Anforderungen kann dies hervorrufen, gerade dann, wenn die Ansprüche zunehmen, weil beispielsweise eine erweiterte Funktionalität oder die bisherige Funktionalität in einer besseren Qualität angeboten werden soll. Infolge dessen muss von Zeit zu Zeit erneut eine Aushandlung angestoßen werden, um abermals eine gültige Übereinkunft zwischen Anforderung und Angebot zu erreichen.

Diese Art der Aushandlung wird im weiteren als *explizite Aushandlung* bezeichnet, da die Beteiligten sich durch Absprache einig werden. Das Prinzip der expliziten Aushandlung läßt sich konzeptionell auch zur Kooperation zwischen Selektionsstrategien einsetzen. In diesem Fall würde vor der Bindung einer Komponente durch eine andere zunächst eine Aushandlung der jeweiligen Anforderungen und Angebote durchgeführt und die getroffene Vereinbarung möglichst während der gesamten Bindungsdauer aufrecht erhalten werden.

Aus praktischer Sicht ist die explizite Aushandlung allerdings verhältnismäßig teuer. Die Aushandlung von Anforderungen und Angeboten nimmt auf Grund der notwendigen Kommunikation eine gewisse Zeit in Anspruch. Da in Adhoc-Netzen wegen der hohen Dynamik häufiger mit Neuverhandlungen zu rechnen ist als im Vergleich zu stabilen Rechnernetzen, führt dies zu erheblich mehr Nachrichtenverkehr als erwünscht. Je nach Grad der Dynamik kann der Aufwand für Neuverhandlungen dementsprechend deutlich zunehmen und

die Dauer der Gültigkeit einer Vereinbarung drastisch sinken. Im schlechtesten Fall kommt die eigentliche Kommunikation nicht zustande, da ständige Veränderungen der Umgebung permanent zu Neuverhandlungen führen.

Implizite Aushandlung

Auf Grund dessen, dass die explizite Aushandlung stets einen gewissen Aufwand mit sich bringt, ist diese in ubiquitären Rechnernetzen und speziell beim Einsatz ressourcenarmer Rechneinheiten eher ungeeignet. Als Ausweg wird im Folgenden ein Ansatz betrachtet, der zwar zu suboptimalen Resultaten führen kann, dafür allerdings keinen zusätzlichen Aufwand zur Laufzeit erzeugt. Ein solcher Ansatz ist vor allem deshalb eine Alternative zur expliziten Aushandlung, da selbst letztere kein Garant dafür ist, dass eine Anforderung nach einer erfolgreichen Aushandlung in jedem Fall erfüllt werden kann. Folglich können die anfallenden Kosten zusätzlicher Kommunikation in diesem Fall eingespart werden. Die Tatsache, dass die Erfüllung nichtfunktionaler Anforderungen suboptimal ausfallen kann, wird darüber hinaus, analog zur Unerfüllbarkeit von Anforderungen, als akzeptabel betrachtet. Zur Erläuterung dieses Ansatzes werden nochmals die auftretenden Fälle betrachtet, in denen ein gemeinsames nichtfunktionales Angebot vorliegen muss.

Für die Verarbeitung eines Aufrufes, für den eine Antwort vorgesehen ist, müsste prinzipiell die lokale Selektionsstrategie die nichtfunktionalen Angebote der entfernten Selektionsstrategie kennen, um eine geeignete Auswahl an Plugins durchführen zu können. Ohne explizite Aushandlung ist ihr jedoch nur bekannt, welche Plugins auf entfernter Seite zur Verfügung stehen. Da die nichtfunktionalen Teilangebote dieser Plugins ihr nicht bekannt sind, kann sie bestenfalls davon ausgehen, dass die Plugins auf entfernter Seite ähnliche Teilangebote realisieren wie die entsprechenden Plugins auf lokaler Seite. Das bedeutet, dass die lokale Selektionsstrategie aus allen interoperablen Plugins diejenigen Plugins auswählt, die rein lokal betrachtet optimal sind oder zumindest die gegebene nichtfunktionale Anforderung erfüllen. Auf diese Weise kann zwar nicht garantiert werden, dass die nichtfunktionale Anforderung auf entfernter Seite erfüllt werden kann. Da jedoch Adhoc-Rechnernetze an sich sehr dynamisch sind und selbst nach einer expliziten Aushandlung die Erfüllbarkeit einer Anforderung nicht garantiert werden kann, ist diese Variante ähnlich effektiv und gleichzeitig kostengünstiger, da keine Kommunikationskosten anfallen.

Sofern eine nichtfunktionale Anforderung globale Eigenschaften beinhaltet, müsste das entfernte nichtfunktionale Angebot der lokalen Selektionsstrategie bekannt sein. Um auch hier die Kommunikationskosten einsparen zu können, ist

wie indem zuvor beschriebenen Fall eine rein lokale Betrachtung durch die Selektionsstrategie möglich. In einem weiteren Schritt wäre eine etwas aufwendigere Variante denkbar. Die entfernte Selektionsstrategie könnte dieselben Plugins für die Antwort verwenden, die auch zuvor für den Versand des eigentlichen Aufrufes eingesetzt wurden. Da diese früher schon zumindest auf lokaler Seite die nichtfunktionale Anforderung erfüllt haben, kann die Wiederverwendung dieser Plugins auch für die Ergebnisantwort geeignet sein und auf entfernter Seite die Anforderungen abermals erfüllen. Allerdings wäre hierfür in jedem Fall ein zusätzlicher Zwischenspeicher erforderlich, der Informationen darüber bereithält, welche Plugins jeweils beim Versand des ursprünglichen Aufrufes verwendet wurden. Auf den ersten Blick schränkt die Wiederverwendung zuvor genutzter Plugins die Flexibilität von BASE ein. Allerdings könnte sowohl bei Unerfüllbarkeit einer Anforderung als auch bei gestörter Kommunikation immer noch auf die allgemeine Kommunikation ohne Berücksichtigung nicht-funktionaler Anforderungen zurückgegriffen werden, um das Ergebnis trotz Unerfüllbarkeit an den Aufrufer zu übertragen. Nichtsdestotrotz wird der flexible Einsatz beliebiger Plugins zwischen Aufruf und Antwort verhindert.

Ein weiteres Problem ist, dass zwischen Erhebung des lokalen Angebotes der lokalen Selektionsstrategie und Wiederverwendung der selektierten Plugins auf entfernter Seite ein gewisses Zeitfenster besteht. Auf Grund der Dynamik der Umgebung kann sich natürlich das nichtfunktionale Angebot der lokalen Selektionsstrategie unter Umständen negativ verändern. In diesem Fall kann die Wiederverwendung der ursprünglichen Pluginauswahl zur Unerfüllbarkeit der Anforderung führen.

Letztlich ist bezüglich der Realisierung dieser Optimierung eine zusätzliche Programmlogik erforderlich, die die Implementierung der Erweiterung zusätzlich kompliziert.

Da zum einen die Zielgruppe der Rechnerknoten zum größtenteils mobil ist und nichtfunktionale Angebote sich kurzfristig verändern können, ist die explizite Aushandlung nichtfunktionaler Angebote in vielen Fällen ungeeignet. Vielmehr scheint ein Ansatz geeignet, der ohne zusätzliche Kommunikationskosten möglichst auf der Basis ähnlich realisierbarer Angebote von Selektionsstrategien die Erfüllbarkeit nichtfunktionaler Anforderung versucht. Da prinzipiell in Adhoc-Rechnernetzen keine Garantien möglich sind, sollten zumindest zusätzliche Kosten für die Aushandlung nichtfunktionaler Anforderungen und Angebote vermieden werden. Aus Gründen der Vereinfachung wird deshalb der Ansatz verfolgt, Selektionsstrategien ausschließlich auf der Grundlage lokal

verfügbarer Angebote zu realisieren. Für diesen Ansatz müssen insofern lediglich nichtfunktionale Anforderungen zwischen Selektionsstrategien übertragen werden.

Kapitel 7

Entwurf

In diesem Kapitel wird der Entwurf der Erweiterung vorgestellt. Zunächst wird die neue Plugin-Architektur von BASE behandelt. Anschließend wird der Entwurf zur Unterstützung nichtfunktionaler Anforderungen beschrieben.

7.1 Restrukturierte Plugin-Architektur

Die neue Version von BASE zeichnet sich vor allem durch die klare Strukturierung mittels Plugin-Schichten aus. Aus diesem Grund beginnt die Betrachtung des Entwurfs an dieser Stelle. Anschließend wird näher beschrieben, auf welche Weise die Plugin-Schichten verwaltet werden. Zuletzt wird näher auf die Verarbeitung von Nachrichten und Aufrufen unter Zuhilfenahme der Selektionsstrategien eingegangen. Die Beschreibung des folgenden Entwurfs konzentriert sich aus Gründen der Übersichtlichkeit nur auf die wesentlichen Teile der neuen Plugin-Architektur.

Schichtenmodell, Plugins und Pluginbeschreibungen

Abbildung 7.1 zeigt die Klassen an Plugins, die von der neuen Architektur unterstützt werden. Je nach Funktionalität erbt jede Implementierung eines Plugins von einer der spezialisierten und abstrakten Klassen *DiscoveryPlugin*, *TransceiverPlugin*, *TransportPlugin*, *IOPPlugin*, oder *ErrorPlugin*. Jede dieser Klassen erbt wiederum von der abstrakten Klasse *Plugin*. Diese bietet übergreifend Funktionen zur Abfrage der Funktionalität, der zur Verwendung eines Plugins eventuell erforderlichen Parameter und der Schicht (*Layer*) eines Plugins. Letztere wird bereits durch die spezialisierte Plugin-Klasse festgelegt, indem eine statische Konstante die entsprechende Schicht referenziert. Die Funktionalität eines Plugins wird durch einen beliebigen, aber einheitlichen Bezeichner

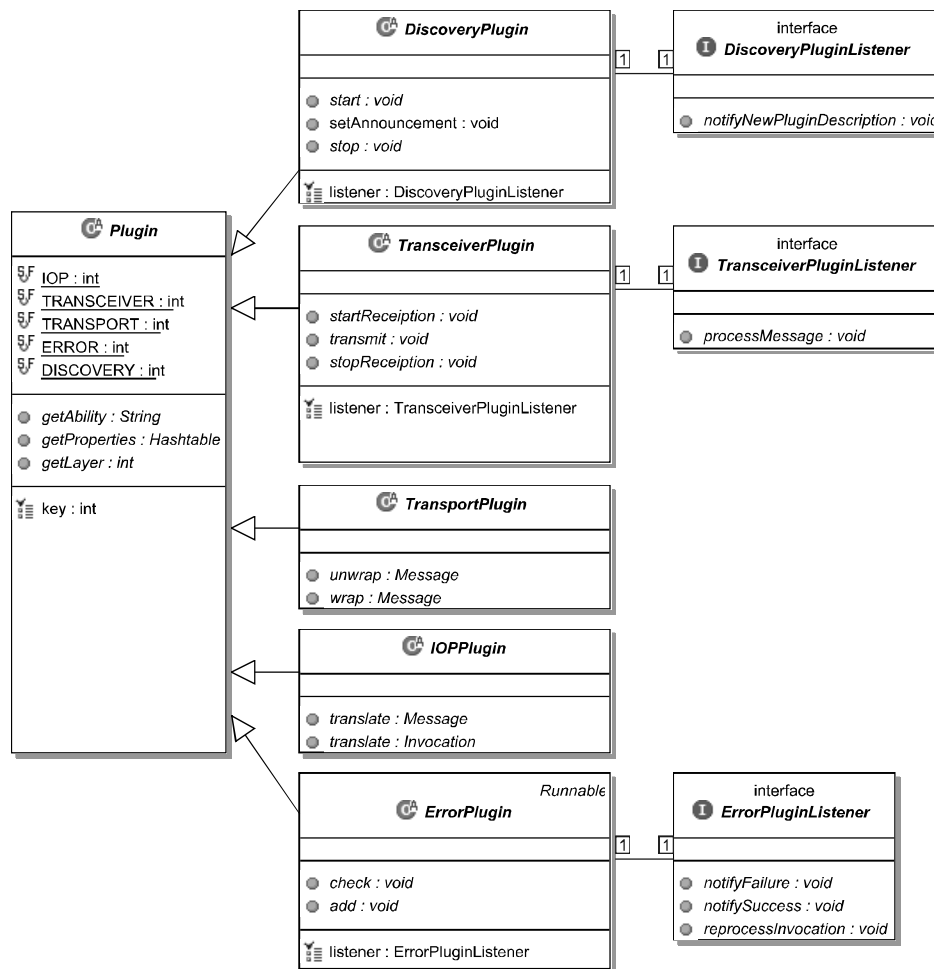


Abbildung 7.1: Schichtenmodell

beschrieben (*Ability*). Beispielsweise würde ein Plugin zur Übertragung von Nachrichten über eine serielle Schnittstelle von *TransceiverPlugin* erben und als Ability *RS232* angeben. Als Parameter würden in diesem Fall unter anderem *Baudrate* und *Parität* verwendet werden.

DiscoveryPlugins werden zum Versand und Empfang von Informationen über die lokal bzw. entfernt installierten Plugin-Instanzen (kurz Plugins) verwendet. An Hand dieser Informationen, die in sogenannten *PluginDescriptions* erfasst werden, können zwischen Instanzen der Verteilungsinfrastruktur später die Interoperabilitätsmengen bestimmt werden. Abbildung 7.2 zeigt die Beziehung zwischen Plugins und *PluginDescriptions*. Um eine eindeutige Referenz zwischen einem Plugin und seiner Beschreibung herstellen zu können, erhalten beide Objekte einen gemeinsamen Schlüssel (*Key*), anhand dessen eine eindeutige Zuordnung möglich ist. Mengen von Plugins und *PluginDescriptions* können

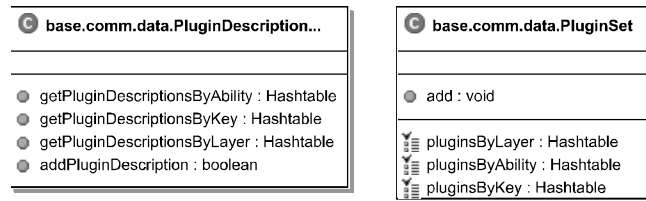


Abbildung 7.2: Plugins und ihre Beschreibungen

zur Vereinfachung in *PluginSets* bzw. *PluginDescriptionSets* gespeichert (siehe Abbildung) und mit dem Key, der Konstante einer Pluginschicht oder einer Ability abgefragt werden. Sofern eine Menge an PluginDescriptions über das Netzwerk angekündigt werden soll, werden diese über die Methode *setAnnouncement()* konfiguriert. *Start()* und *Stop()* aktivieren bzw. deaktivieren jeweils die periodische Ankündigung. Empfangene PluginDescriptions werden über die Schnittstelle *DiscoveryPluginListener()* weitergereicht. Das Objekt, das die PluginDescriptions erhält, wird als *Listener* gesetzt.

TransceiverPlugins ermöglichen den Empfang und Versand von Nachrichten über das Netzwerk. Die Schnittstelle hierfür ist relativ einfach. Es existiert lediglich eine Methode *transmit()* zum Versand einer Nachricht. Um empfangene Nachrichten verarbeiten zu können, muss am TransceiverPlugin ein Objekt registriert werden, das die Schnittstelle *TransceiverPluginListener* implementiert. Dieses erhält bei eingehenden Nachrichten die entsprechenden Daten. *TransportPlugins* kapseln Nachrichten in ein bestimmtes Transportprotokoll, damit Nachrichten, die über eine mehrfach verwendete Verbindung übertragen werden, von einander klar getrennt werden können. Um Nachrichten zu verpacken, wird die Methode *wrap()* verwendet. Um Nachrichten zu entpacken, steht die Methode *unwrap()* zur Verfügung. *IOPPlugins* unterstützen im Einzelnen ein Interoperabilitätsprotokoll, das sowohl aus einer vollständigen Nachricht einen Aufruf erzeugt als auch aus einem Aufruf eine Nachricht in Form eines Bytestroms erzeugt. *ErrorPlugins* ermöglichen den Einsatz von Fehlersemantiken in Bezug zu einzelnen Aufrufen.

PluginManager und Selektionsstrategien

Die Verwaltung der Plugins wird vom sogenannten *PluginManager* übernommen und in Abbildung 7.3 dargestellt. Er ist der zentrale Verwalter der neuen Architektur und das eigentliche Kommunikationsmittel des *InvocationBrokers*. Letzterer stammt noch aus der originären Plugin-Architektur. Der *InvocationBroker* erzeugt zu Beginn der Laufzeit eine neue Instanz des *PluginManagers* und

übergibt Konfigurationsdaten in Form einer *CommConfiguration* an den Plugin-Manager. Dieses Konfigurationsobjekt beschreibt, welche Plugins und welcher Treiber einer Selektionsstrategie geladen und später verwendet werden sollen. Mit Hilfe einer gesonderten Lader-Klasse *PluginLoader* werden die Klassen der Plugins aufgelöst, instanziiert und, sofern notwendig, gestartet. Anschließend werden diese zur späteren Verwendung in einem einfachen Repository *PluginDatabase* abgelegt. Die zuvor beschriebenen *PluginDescriptions* und Plugins werden vom *PluginManager* in einem Repository namens *PluginDatabase* vorgehalten. Dieses ermöglicht die effiziente Suche nach Plugins und *PluginDescriptions*. Ein interner Thread entfernt asynchron *PluginDescriptions*, die seit längerem nicht mehr über das Netzwerk empfangen wurden.

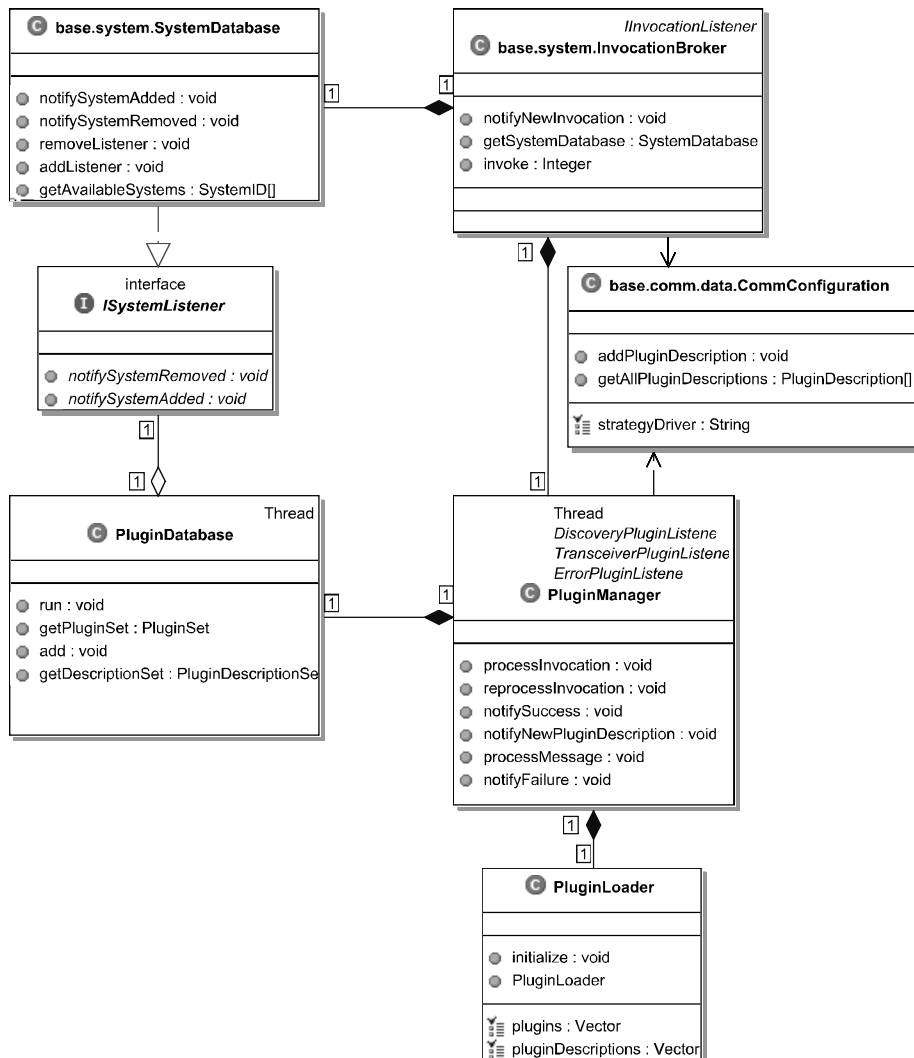


Abbildung 7.3: Plugin-Verwaltung

Sofern das Laden und Registrieren der Selektionsstrategie, der Plugins und der dazugehörigen Beschreibungen abgeschlossen wurde, ist der PluginManager bereit, Aufrufe und Nachrichten entgegenzunehmen. Abbildung 7.4 zeigt das Klassendiagramm zur Beschreibung von Nachrichten und Aufrufen.

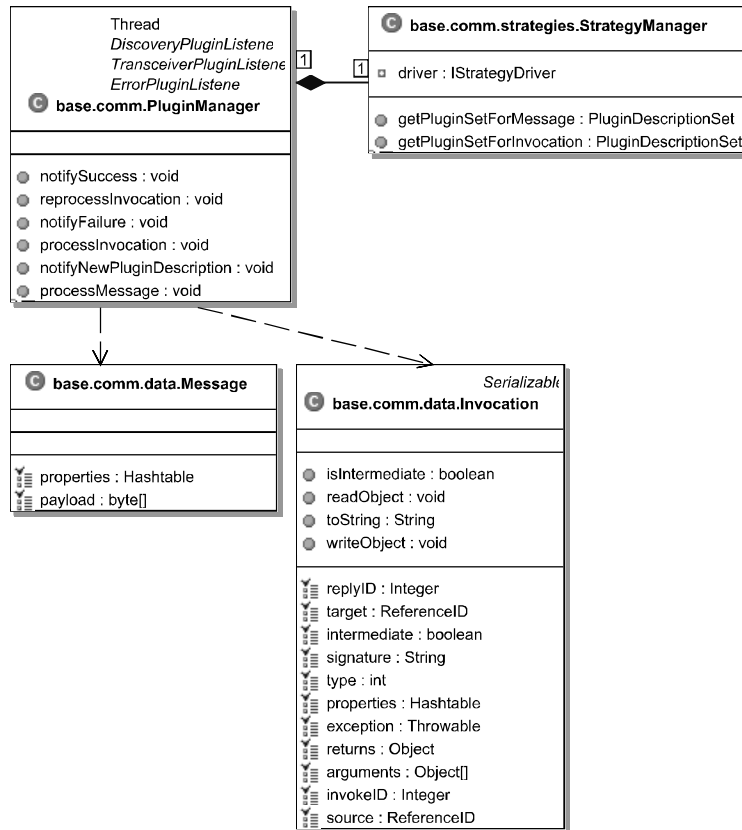


Abbildung 7.4: Verarbeitung von Aufrufen und Nachrichten

Sofern ein Aufruf mit Hilfe der Methode `processInvocation()` vom InvocationBroker übergeben wurde, führt der PluginManager die Transformation des Aufrufes in eine oder mehrere Nachrichten durch. Die Abbildungen 7.5 und 7.6 zeigen hierzu die Sequenz-Diagramme für die Transformation eines Aufrufes in eine Nachricht. Ausgelöst durch eine empfangene Nachricht über die Schnittstelle des `TransceiverPluginLister` wird die Transformation einer Nachricht in einen Aufruf angestoßen (siehe Sequenz-Diagramm in Abbildung 7.7). Der PluginManager verwendet zur Bestimmung derjenigen Plugins, die zur Transformation eingesetzt werden sollen, den sogenannten `StrategyManager`. Zu diesem Zweck versorgt der PluginManager den StrategyManager mit PluginBeschreibungen aus der PluginDatabase. Ist die Menge der zu verwendenden Plugins vom StrategyManager bestimmt worden, werden diese im Einzelnen Schritt für Schritt

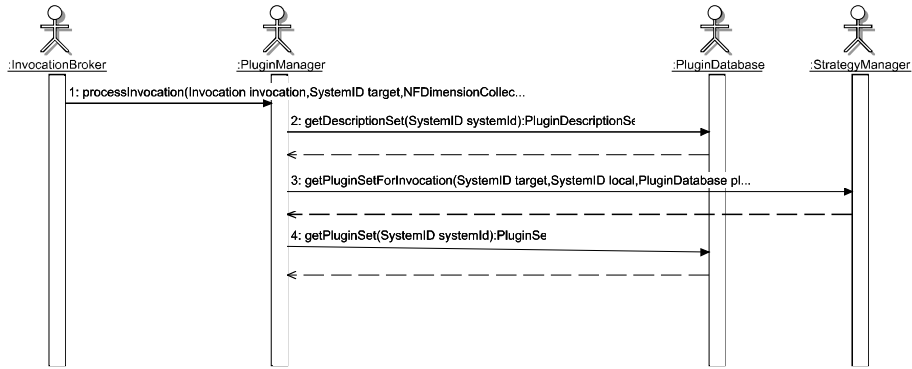


Abbildung 7.5: Transformation eines Aufrufes in eine Nachricht

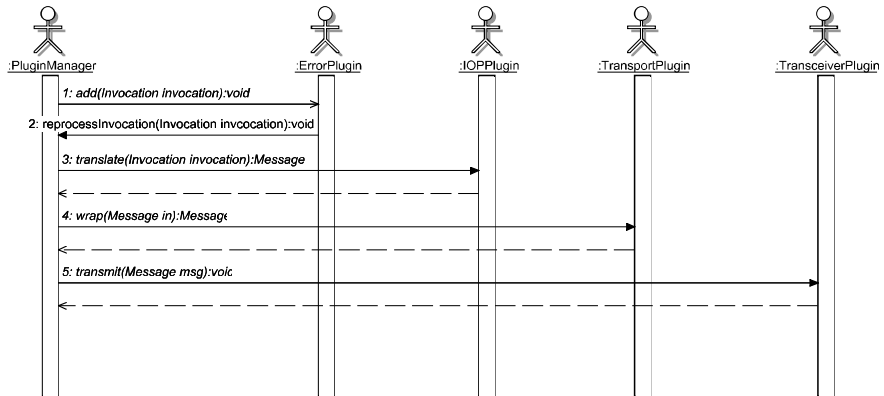


Abbildung 7.6: Transformation eines Aufrufes in eine Nachricht

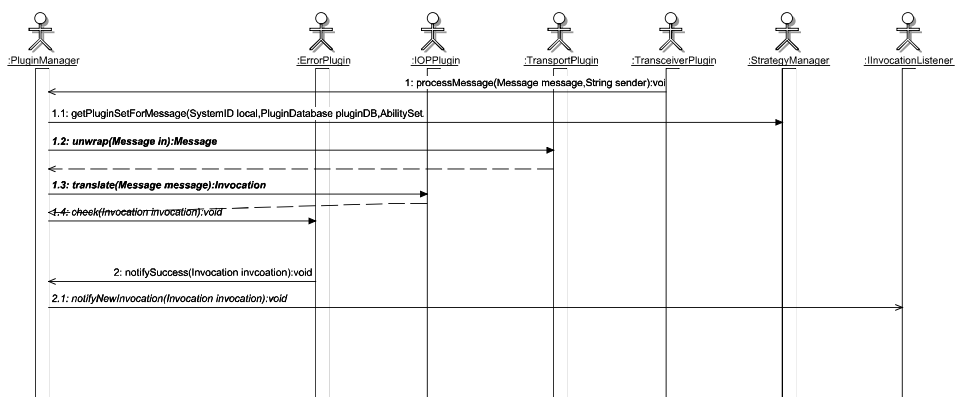


Abbildung 7.7: Transformation einer Nachricht in einen Aufruf

für die Transformation des Aufrufes verwendet. Die Verarbeitungsreihenfolge der Plugins ist dem PluginManager implizit bekannt und kann zukünftig und bei Bedarf entsprechend verändert oder erweitert werden. Letztendlich wird die resultierende Nachricht mit Hilfe eines TransceiverPlugins über das Netzwerk verschickt. Dieselbe Vorgehensweise gilt prinzipiell auch für die Retransformation einer Nachricht in einen Aufruf. Sofern der PluginManager über die Schnittstelle *TransceiverPluginListener* eine Nachricht erhält, führt dieser abermals mit Hilfe des StrategyManagers und der lokal verfügbaren Plugins die Retransformation durch. Allerdings müssen dazu die zur Erzeugung einer Nachricht verwendeten Abilities direkt aus der Nachricht ausgelesen werden.

Wie in Abbildung 7.8 zu erkennen ist, verwendet der StrategyManager

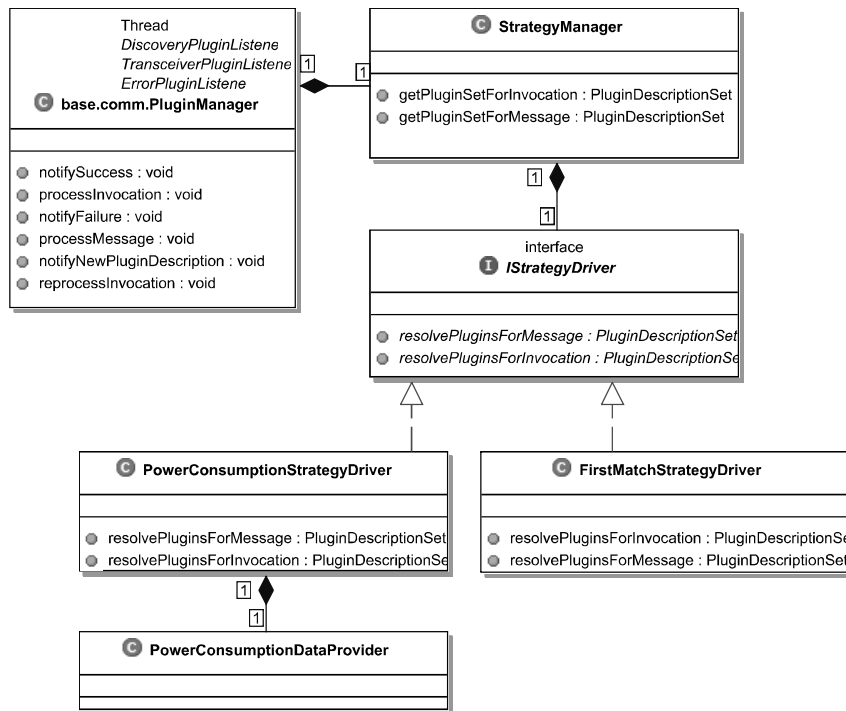


Abbildung 7.8: Entwurf der Selektionsstrategien

intern einen Treiber, der die gewünschte Selektionsstrategie realisiert. Dieser wählt zu einer gegebenen Plugin-Menge einen Satz an Plugins aus, der eine gegebene nichtfunktionale Anforderung erfüllt. Der Treiber entscheidet somit letztendlich, welche Plugins für die Verarbeitung durch den PluginManager verwendet werden müssen. Unabhängig vom Treiber können bereits durch den StrategyManager die kompatiblen Pluginbeschreibungen bestimmt werden, die später vom Treiber benötigt werden. Da dies unter Erfüllung nichtfunktionaler Anforderungen geschehen muss, müssen diese zur Verfügung stehen. Dazu inte-

grieren Komponenten ihre Anforderungen in den Aufruf und können somit bei der Transformation dieses Aufrufes ausgelesen werden. Damit nichtfunktionale Anforderungen auch auf der Gegenseite bekannt sind, werden diese innerhalb einer Nachricht serialisiert. Zur Vereinheitlichung aller Treiber implementieren diese eine generische Schnittstelle *IStrategyDriver*. Diese Schnittstelle sieht Methoden zur Bestimmung der Erfüllbarkeitsmenge sowohl für einen Aufruf als auch eine Nachricht vor. Wurde ein Treiber zur Bestimmung der zu verwendenden Plugins aufgerufen, wird die resultierende Menge bis zum *PluginManager* zurückgereicht und von diesem während der Verarbeitung von Aufrufen und Nachrichten verwendet. Ist die Menge leer, wird die leere Menge zurückgegeben. Der *PluginManager* ist in diesem Fall für weitere Schritte zuständig. Abbildung 7.8 zeigt zwei exemplarische Treiber *PowerConsumptionStrategyDriver* und *FirstMatchStrategyDriver*.

Da ein Treiber auch auf die Verfügbarkeit nichtfunktionaler Teilangebote aller installierten Plugins angewiesen ist, müssen diese ebenfalls bereitgestellt werden. Einerseits wäre es möglich, die jeweiligen Teilangebote direkt in der Plugin-Implementierung zu realisieren. Das hätte allerdings den Nachteil, dass diese für jede Selektionsstrategie bzw. jeden Treiber verändert und neu kompiliert werden müssten. Andererseits ist es unpraktisch, die Teilangebote direkt im Treiber zu realisieren, da dieser, sofern neue Plugins hinzu oder Plugins aus der aktuellen Konfiguration der Verteilungsinfrastruktur entfernt werden, ebenfalls angepasst und neu kompiliert werden müsste. Der einzige Ausweg ist die Kapselung dieser Funktionalität in einem eigenen Objekt, das als *DataProvider* bezeichnet wird. Anhand der Abbildung ist klar ersichtlich, dass dieser nicht verwendet werden muss. Der relativ einfache *FirstMatchStrategyDriver* benötigt diesen beispielsweise nicht.

7.1.1 Nichtfunktionale Beschreibungen

Um die Verwendung nichtfunktionaler Anforderungen möglichst einfach zu halten und nicht auf Grund einer hohen Anzahl an Klassen den Speicherverbrauch in die Höhe zu treiben, wurde, anstatt einer mehrschichtigen Modellierung nichtfunktionaler Angebote, Anforderungen und Eigenschaften, der Entwurf diesbezüglich relativ klein gehalten. Abbildung 7.9 zeigt den Entwurf.

Eine nichtfunktionale Eigenschaft wird mit Hilfe eines Objekts der Klasse *NFDimension* modelliert. Diese Klasse ermöglicht es, den Namen der Eigenschaft und sofern eine Ordnung für die Eigenschaft vorgesehen ist, die Ausrichtung dieser zu formulieren. Der Name einer *NFDimension* dient gleichzeitig als eindeutige Kennung. Für die Ausrichtung existieren die zwei Konstanten

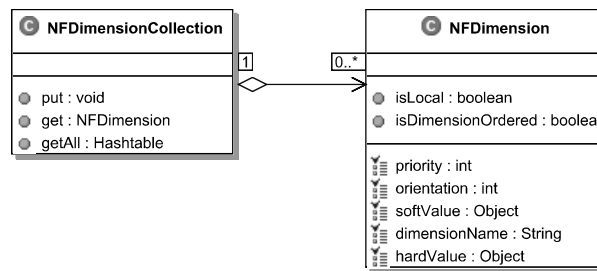


Abbildung 7.9: Nichtfunktionale Beschreibungen

Aufsteigend und *Absteigend*. Darüber hinaus können einer *NFDimension* zwei Objekte zugewiesen werden, die den minimalen und den optimalen Wert der Eigenschaft angeben. Der minimale Wert muss immer gesetzt werden, der optionale Wert nur bei Bedarf. Mit Hilfe der Priorität einer *NFDimension* können später eventuelle adaptive Maßnahmen angewendet werden. Die Objekte, die als Werte übergeben werden, müssen zur Laufzeit später derselben Klasse entspringen, damit ein Vergleich dieser möglich ist. Da die Selektionsstrategie als einzige die Werte einer *NFDimension* analysiert, können dort die erforderlichen Methoden implementiert werden. Auf diese Weise können einer *NFDimension* künftig beliebige Wertetypen zugewiesen werden. Damit eine bestimmte nichtfunktionale Eigenschaft ausschließlich auf der lokalen Instanz der Verteilungsinfrastruktur erfüllt wird, muss bei der Konstruktion einer *NFDimension* diese Information explizit im Konstruktor spezifiziert werden.

Um eine nichtfunktionale Anforderung aus mehreren nichtfunktionalen Eigenschaften erstellen zu können, ist es möglich, eine Sammlung nichtfunktionaler Eigenschaften in einer *NFDimensionCollection* zusammenzufassen. In dieser werden *NFDimensions* anhand der Namen gespeichert und können bei Bedarf über diesen abgefragt werden. Darüber hinaus können auch alle Eigenschaften gleichzeitig abgefragt werden. Konzeptionell kann eine *NFDimensionCollection* sowohl als nichtfunktionale Anforderung als auch als nichtfunktionales Angebot fungieren.

Kapitel 8

Evaluation

In diesem Kapitel wird das erarbeitete Konzept und die daraus resultierende Implementierung der Erweiterung zur Unterstützung nichtfunktionaler Eigenschaften bewertet. Zu diesem Zweck werden einige exemplarische Anwendungen der Erweiterung vorgestellt, die zeigen sollen, dass die Erweiterung tatsächlich angewendet werden kann. Anschließend werden die Kosten für die Erweiterung ermittelt.

8.1 Proof of Concept

Es werden nun einige Selektionsstrategien für die Anwendung der Erweiterung präsentiert. Zunächst wird mit einigen einfachen Selektionsstrategien begonnen. Nach und nach werden diese dann umfangreicher, was zeigen soll, dass die Erweiterung auch für komplexere Anwendungsszenarien geeignet ist.

Beispiel Energieeffizienz

Die Aufgabe der Selektionsstrategien ist es, den Energieverbrauch der lokalen Rechereinheit zu optimieren. Hierzu verfügt die Selektionsstrategie über einen Data Provider, der jedem Plugin einen statischen Wert im Bereich von 0 bis 500 mWsec für die Eigenschaft *Energieverbrauch* zuweist. Die Selektionsstrategie berechnet basierend auf allen möglichen Kombinationen an Plugins, den gesamten Energieverbrauch jeder Kombination. Da die Eigenschaft Energieverbrauch eine vertikale Eigenschaft ist, wird durch Addition der jeweiligen Werte einer Plugin-Kombination der gesamte Energieverbrauch über alle Plugin-Schichten ermittelt. Die Kombination von Plugins, die den minimalen Energieverbrauch aufweist, wird als Resultat zurückgeliefert.

Beispiel Übertragungsgeschwindigkeit

Eine Selektionsstrategie dient zur schnellstmöglichen Übertragung von Aufrufen. Auch hierfür wird ein Data Provider verwendet. Dieser weist jedem Plugin auf Übertragungsebene eine Eigenschaft *Bandwidth* zu. Die Werte liegen zwischen 1 und 11 MBit. Während die Selektion von Plugins auf anderen Ebenen beliebig ist, wird auf Übertragungsebene dasjenige Plugin ausgewählt, das die höchste Bandbreite aufweist.

Beispiel Verschlüsselung

Die Verschlüsselung von Nachrichten erfordert eine zusätzliche Schicht in der Plugin-Architektur. Da der Plugin-Manager diesbezüglich mit minimalen Änderungen angepasst werden kann, stellt die Integration dieser Schicht kein Problem dar. Die Schicht wird direkt oberhalb der Transportschicht eingefügt, da hier bereits vollständig serialisierte Daten verfügbar sind. Die Schicht kann prinzipiell auch auf anderer Ebene der Plugin-Architektur mit entsprechend anderer Semantik realisiert werden. Die Plugins in dieser Schicht implementieren jeweils bestimmte Algorithmen, die den Stufen 1 bis 5 zugeordnet werden können. Stufe 1 impliziert keine Verschlüsselung und Stufe 5 den höchsten Verschlüsselungsgrad. Die Selektionsstrategie liefert eine Menge an Plugins zurück, die das Verschlüsselungs-Plugin mit der höchsten Sicherheitsstufe enthält.

An dieser Stelle muss natürlich beachtet werden, dass die Zuweisung der Sicherheitsgrade zwischen den beteiligten Selektionsstrategien prinzipiell unterschiedlich vorgenommen werden. Die Ursache hierfür liegt in der nur vagen und nicht näher festgelegten Semantik des Begriffs „Verschlüsselungsgrad“. Da allerdings kooperative Selektionsstrategien zuweisungs-äquivalent sind, wird dieser Fall als fehlerhaft betrachtet.

Beispiel Fehlersemantik

Auf Ebene der Fehlerbehandlung sind Plugins vorgesehen, die jeweils die Semantik *Best-Effort*, *At-least-once* oder *At-most-once* realisieren. Jedes dieser Plugins unterstützt dementsprechend eine Eigenschaft *Fehlersemantik*, mit den jeweiligen String-Werten *Besteffort*, *Atleastonce* und *Atmostonce*. Wird beispielsweise in der nichtfunktionalen Anforderung für die Eigenschaft Fehlersemantik der Wert *Atleastonce* spezifiziert, wählt die Selektionsstrategie ein Plugin aus, das genau diesen Wert für die Eigenschaft *Fehlersemantik* aufweist.

Beispiel Verzögerung

In Kapitel 5.2.1 wurde erläutert, weshalb die künstliche Verzögerung von Aufrufen durchaus sinnvoll sein kann. Um diese zu realisieren, können ebenfalls die Plugins, die Fehlersemantiken realisieren, verwendet werden. Allerdings kann auch eine weitere Plugin-Schicht integriert werden, die ausschließlich diese Funktionalität bereitstellt. In diesem Beispiel wird jedoch davon ausgegangen, dass die Plugins der Fehlersemantik diese Aufgabe übernehmen. Dazu unterstützt jedes Plugin die Eigenschaft *MaxDelay*. Die jeweiligen Bedürfnisgrade sind vom Typ Boolean und geben Auskunft darüber, ob eine künstliche Verzögerung unterstützt wird oder nicht. Die Selektionsstrategie würde für eine Anforderung, die diese Eigenschaft voraussetzt, ein Plugin auswählen, das die Eigenschaft mit *True* erfüllt. Da die Fehlersemantik-Plugins bisher nicht für diese Funktionalität vorgesehen waren, müssen nachträglich die Methoden zur Verarbeitung von Aufrufen um einen Parameter erweitert werden, der die jeweils gewünschte maximale Verzögerung angibt.

Beispiel Energieeffizienz 2

Wie im ersten Beispiel beschrieben soll eine Selektionsstrategie die Kombination an Plugins finden, die den minimalen Energieverbrauch aufweist. Anders als im ersten Beispiel soll nun auch der Inhalt eines Aufrufes berücksichtigt werden. Da die Selektionsstrategie unter anderem das Aufruf-Objekt zur Selektion von Plugins erhält, kann diese den Aufruf bezüglich Anzahl der Parameter und Länge der Parameter untersuchen. Vorausgesetzt alle Plugins unterstützen über einen Data Provider die Eigenschaft *PowerConsumption*, die den Energieverbrauch in Abhängigkeit zur Länge in Bytes zurückliefert, kann die Selektionsstrategie unter Beachtung der Gesamtlänge aller Parameter abermals den gesamten Energieverbrauch erheben und die Kombination mit dem geringsten Verbrauch auswählen.

8.2 Performanz

Es werden nun die Kosten für den Einsatz der Erweiterung dieser Arbeit unter Zuhilfenahme quantitativer Messungen betrachtet. Dazu werden zum einen die zusätzlichen Kosten der entwickelten Erweiterung untersucht. Diese setzen sich maßgeblich aus der Flexibilisierung der Plugin-Architektur und dem Einsatz von Selektionsstrategien zusammen. Allerdings können die zusätzlichen Kosten für Selektionsstrategien nicht allgemeingültig erhoben werden, da diese stets für ein bestimmtes Anwendungsszenario entwickelt werden und somit

die jeweilige Komplexität der Selektionsstrategie unterschiedlich ausfällt. Da die neue Plugin-Architektur jedoch nur in Verbindung mit einer Selektionsstrategie funktioniert, ist die einzige sinnvolle Möglichkeit der Kostenerhebung, die Kosten der flexibleren Plugin-Architektur mit einer Selektionsstrategie zu bestimmen, die zur Laufzeit nur minimale Selektionen durchführt und somit minimalen Aufwand verursacht. Da der zusätzliche Aufwand der minimalen Selektionsstrategie auch für andere Selektionsstrategien anfällt, kann eine untere Schranke für die Leistung der Erweiterung ermittelt werden.

Typischerweise wird die Flexibilisierung existierender Systeme durch eine hohe Entkopplung unterschiedlicher Funktionalitäten erreicht, so auch in dieser Arbeit. Im Normalfall führt dies zu einer wesentlich höheren Anzahl an Klassen und Objekten und einem deutlichen Anstieg an Nachrichten zwischen diesen. Zum Vergleich: die ursprüngliche Version von BASE besteht aus 58 Klassen, die neue BASE-Version aus 87 Klassen. Somit ist für die folgenden Untersuchungen zu erwarten, dass die restrukturierte Plugin-Architektur bezüglich Zeit und Speicherverbrauch schlechter abschneidet als die originäre BASE-Version. Im Folgenden werden Messungen der Performanz durchgeführt, die einen Vergleich zwischen der originären BASE-Version und der neuen Version mit der flexibleren Plugin-Architektur ermöglichen.

Testumgebung

Für die Messungen standen folgende Komponenten zur Verfügung:

Rechnertyp	Prozessor	Hauptspeicher	Netzwerk
IBM Thinkpad T40p	Mobile Pentium 1.6GHz	512MB	1000MBit Ethernet
Desktop-PC	Pentium 4 450MHz	256MB	100MBit Ethernet
Netgear FM144P Switch	ADMtek 5106 75Mhz	4MB	4x100Mbit Ethernet

Tabelle 8.1: Testumgebung

Als verteilte Komponentenanzwendung wurde eine existierende Komponente, die bisher zu Demonstrationszwecken diente, angepasst und unter dem *Java Development Kit 1.4.2* von *Sun Microsystems* auf beiden Rechnern installiert. Die Implementierung der Anbieter-Komponente stellt den bekannten Systemaufruf `System.out.println` anderen Komponenten über eine synchrone Schnittstelle zur Verfügung. Eine Nutzer-Komponente verwendet diese Schnittstelle, um die Ausgabe 1234567890 auf der entfernten Konsole zu erzeugen.

Die Ergebnisse sollen zeigen, wie stark die Flexibilisierung von BASE sich auf die Verarbeitungszeit in BASE niederschlägt. Aus diesem Grund wurde inner-

halb der Plugin– Architektur sowohl auf dem Server als auch auf dem Klienten bei jedem ein– und ausgehenden Aufruf die jeweilige Bearbeitungszeit gemessen. Darüber hinaus wurde auf dem Klienten jeweils die Gesamtdauer eines einzelnen Aufrufes gemessen. Auf diese Weise kann die verbleibende Zeit der restlichen Verteilungsinfrastruktur und dem Nachrichtentransfer zugerechnet werden.

Da Ausgaben auf die Konsole erhebliche Auswirkungen auf das Messergebnis besitzen, wurden diese möglichst während der Kommunikation unterdrückt. Insgesamt wurden 3 Durchläufe a 10 mal 1000 Aufrufe getätigt. Die Aufrufe wurden in Zeitintervallen von je 50ms auf Seite der Nutzer–Komponente getriggert. Da seit einiger Zeit aktuellere Java Virtual Machines Just–In–Time–Kompilierung unterstützen und diese in bestimmten, aber nicht in allen Fällen auf Grund der Übersetzung des Java Byte Code in nativen Code zu messbaren Leistungssteigerungen führt, wurden die Testläufe jeweils mit und ohne aktiviertem JIT durchgeführt.

Die Speichermessungen wurden mit den von Java angebotenen Systemfunktionen `Runtime.freeMemory()` und `Runtime.totalMemory()` berechnet. Die Zeitmessungen wurden mit Hilfe der Methode `System.currentTimeMillis()` durchgeführt.

Konfiguration

Für die originäre BASE–Version wurde ausschließlich das existierende Tcpip–Plugin eingesetzt, das sowohl das Interoperabilitäts– und Transportprotokoll als auch die Übertragung über Java Sockets realisiert. Für die neue Version von BASE wurde ein Interoperabilitäts–Plugin verwendet, das im Wesentlichen aus dem Tcpip–Plugin übernommen wurde. Als Transportprotokoll wurde ein einfaches Plugin entwickelt, das lediglich Anfang und Ende einer Nachricht durch feste Kennungen markiert. Zur Übertragung von Nachrichten wurde ein Plugin geschrieben, das analog zum Tcpip–Plugin Java Sockets verwendet.

Bezüglich der neuen BASE–Version wurde, um den Einfluss einer nun erforderlichen Selektionsstrategie minimal zu halten, eine Selektionsstrategie *First–Match* implementiert, die für die Auswahl von Plugins nichtfunktionale Anforderungen nicht berücksichtigt und für jede Schicht der Plugin–Architektur das allererste Plugin, das für diese Schicht registriert wurde, auswählt.

Sowohl für die alte als auch die neue Version von BASE wurden jeweils die grafischen Benutzungsoberflächen deaktiviert, da diese im Vergleich zum eigentlichen System erhebliche Mengen an Speicher benötigen und die Messungen des Speicherbedarfs unnötig verschleiern.

Zeit

Das folgende Diagramm zeigt die über 10 mal 1000 gemessene Durchschnittszeit für einen einzelnen Aufruf und je nach Einsatz des JIT-Compilers sowohl für die alte als auch die neue BASE-Version. Außerdem wurden zum Vergleich in einer weiteren Darstellung die initialen Kosten für das Laden von Klassen und Initialisieren von Objekten durch Bildung der jeweiligen Durchschnitte erst ab dem 2001.ten Aufruf erhoben.

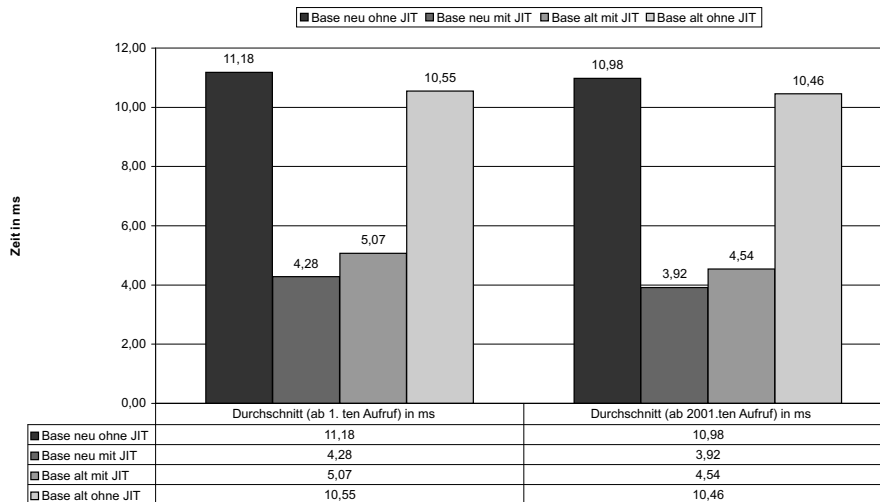


Abbildung 8.1: Vergleich durchschnittliche Aufrufdauern

Es lässt sich erkennen, dass die neue Version im Vergleich zur älteren Version ohne JIT-Compiler um etwa 5–6% langsamer ist, unabhängig davon, ob die ersten 2000 Werte einkalkuliert wurden oder nicht. Da die Flexibilisierung zu einer größeren Anzahl an Nachrichten auf Grund zusätzlich erzeugter Zwischenergebnisse führt, fällt dieses Ergebnis wie erwartet aus. Allerdings ist der Unterschied hinsichtlich der Verarbeitungszeit für einen einzelnen Aufruf zwischen originärer und neuer Plugin-Architektur marginal.

Ein ganz anderes Bild entsteht, wenn der JIT-Compiler eingesetzt wird. In diesem Fall stellt die neue Version von BASE mit ca. 15–18% Geschwindigkeitsvorteil die schnellere Variante dar. Dieses Ergebnis ist überraschend. Der Grund hierfür muss offensichtlich in Zusammenhang mit dem JIT-Compiler stehen. Es ist zu vermuten, dass die wesentlich kleineren Klassen der neuen Plugin-Architektur effizienter in nativen Code übersetzt werden können. Es ist nicht auszuschließen, dass eventuell sogar anstatt der im rein interpretierten Modus eingesetzten *Copy by Value*-Parameter sogar *Copy by Reference*-Parameter eingesetzt werden, die üblicherweise erheblich schneller verarbeitet

werden. Allerdings müsste man hierzu näher die JIT-Technologie untersuchen, was den Rahmen dieser Arbeit sprengen würde. Da die JIT-Technologie in modernen Java Virtual Machines immer häufiger verfügbar wird, ist die bessere Performanz der neuen Version von BASE aus praktischer Sicht durchaus ernst zu nehmen.

Die Zeit zur Verarbeitung von Aufrufen und Nachrichten ausschließlich innerhalb der Plugin-Architektur beträgt zumindest für die neue Version von BASE ohne JIT etwa 73% und mit aktiviertem JIT-Compiler ca. 53% der vollständigen Verarbeitung eines Aufrufes auf einer Seite. Leider ist für die ursprüngliche BASE-Version die Messung der Zeiten zur Übersetzung von Nachrichten in Aufrufe auf Grund der hohen Kopplung unterschiedlicher Funktionalitäten nicht ohne weitere Änderungen dieser möglich gewesen. Da Änderungen der ursprünglichen BASE-Version die Messergebnisse entscheidend verändern können, wurde auf die Erhebung dieser Werte verzichtet.

Die Darstellungen 8.2, 8.3, 8.4 und 8.5 zeigen detailliert den zeitlichen Verlauf der Aufrufzeiten für die unterschiedlichen Versionen mit und ohne JIT-Compiler. Vorallem die initialen Kosten für das Laden und Initialisieren der Klassen und Objekte sind hier klar zu erkennen.

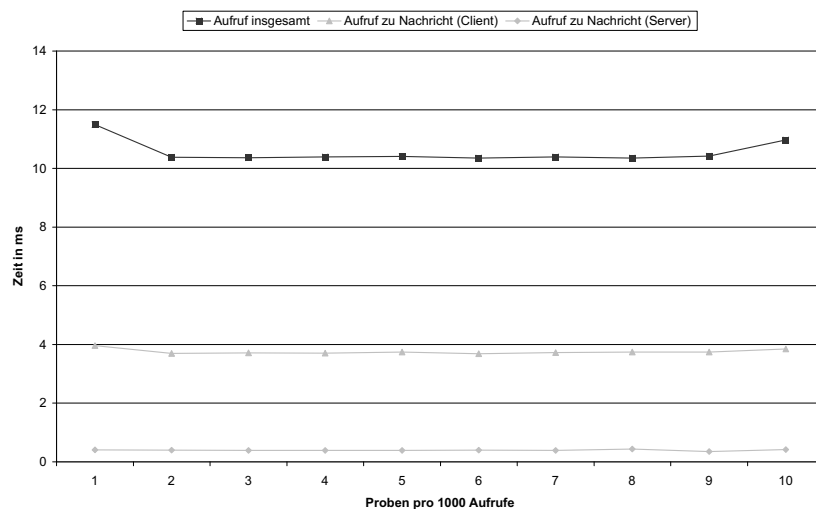


Abbildung 8.2: Alte BASE-Version ohne JIT

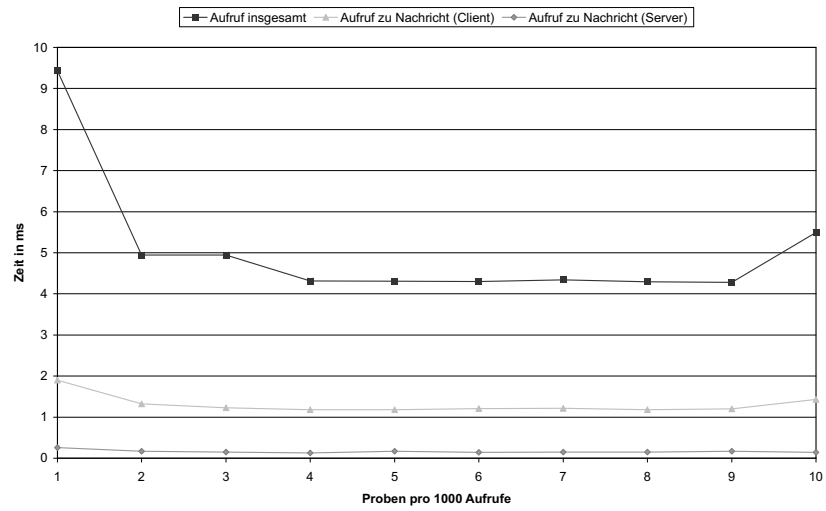


Abbildung 8.3: Alte BASE-Version mit JIT

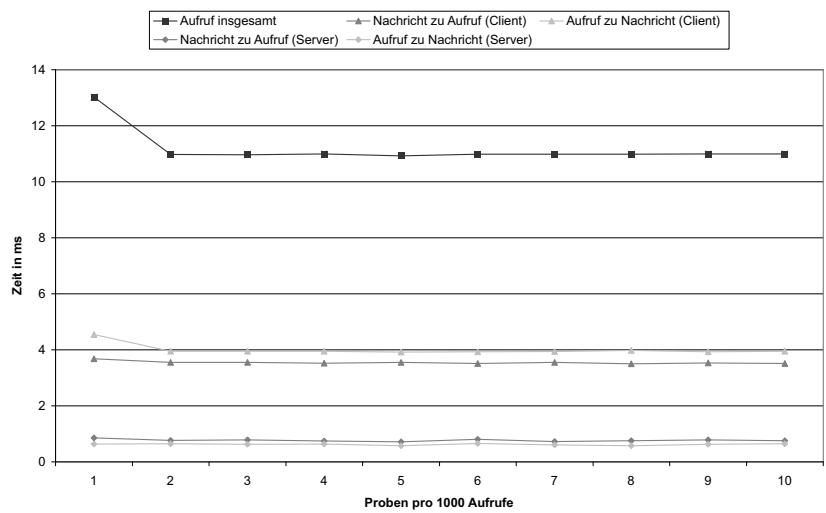


Abbildung 8.4: Neue BASE-Version ohne JIT

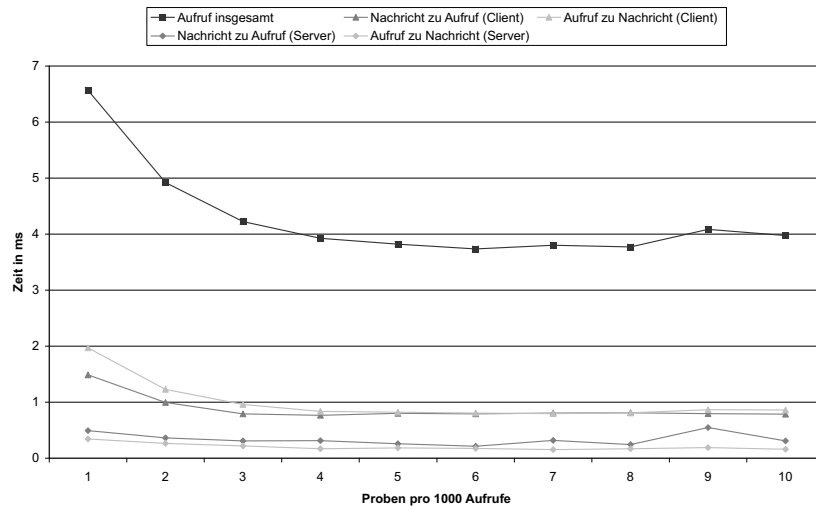


Abbildung 8.5: Neue BASE-Version mit JIT

Speicher

Neben dem zeitlichen Vergleich der alten und neuen Version von BASE wurde auch untersucht, welche Auswirkungen die Flexibilisierung der Plugin-Architektur auf den Speicherverbrauch hat. Dabei wurden vor allem die Messungen in Bezug zu den möglichen Phasen während der Testläufe untersucht. Im Wesentlichen sind dies folgende Phasen:

Ruhephase Zeitraum, in der eine Komponente geladen und gestartet wurde und auf Anfragen anderer Komponenten wartet

Kommunikationsphase Zeitraum, in der eine Komponente mit anderen Komponenten kommuniziert

Erholungsphase Zeitraum, in der die Kommunikation mit anderen Komponenten bereits abgeschlossen wurde und das System wieder auf eingehende Aufrufe wartet

Abbildung 8.6 zeigt die im Durchschnitt gemessenen Speicherauslastungen unter Beachtung dieser Phasen für beide Versionen von BASE und nach Server- und Client-Komponente unterschieden.

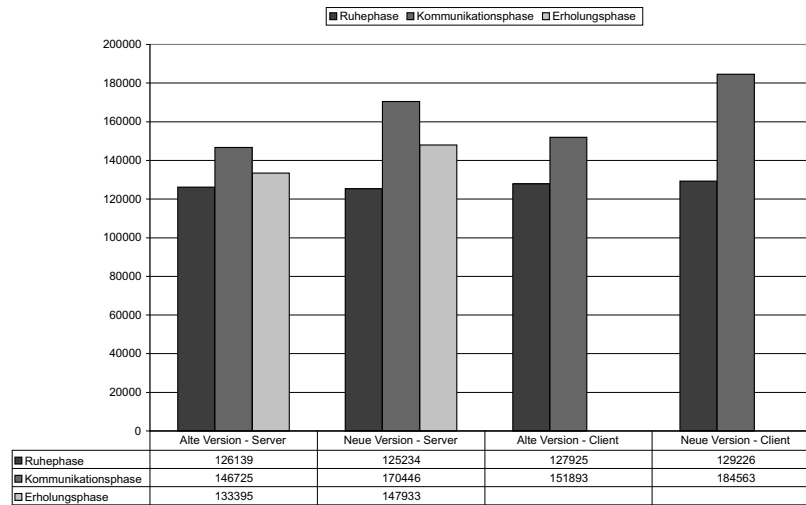


Abbildung 8.6: Vergleich der Speicherauslastungen beider BASE-Versionen

Der Vergleich beider Diagramme zeigt, dass während der Ruhephase von beiden Versionen in etwa die gleiche Speichermenge benötigt wird. Erst in der Kommunikationsphase zeigt sich hier ein Unterschied. Die neue BASE-Version benötigt ca. 24kByte Speicher mehr für die Server-Komponente, für die Client-Komponente sind es ca. 33KByte. Diese Werte lassen sich im Vergleich zur alten Version durch die größere Anzahl an Zwischenpuffern und temporärer Objekte in der neuen Version begründen. Eine Betrachtung der Erholungsphase zeigt, dass in der Ruhephase noch nicht alle verwendeten Objekte geladen und initialisiert worden sind, die zur Kommunikation benötigt werden. Dieses Verhalten wurden bereits bei der zeitlichen Betrachtung aufgezeigt.

Die Abbildungen 8.7 und 8.8 zeigen detailliert den zeitlichen Verlauf der Speicherauslastung. Es ist klar zu erkennen, dass zum Zeitpunkt des Ladens und Initialisieren der Komponenten erhebliche Speichermengen benötigt werden, die aber nach kürzester Zeit wieder freigegeben werden können. Tatsächlich läßt sich dieses Phänomen dadurch erklären, dass die Garbage Collection der Virtual Machine im Normalfall erst dann aktiviert wird, wenn die Speicherauslastung die obere Grenze des aktuell angeforderten Speichers vom Betriebssystem erreicht. Damit sich dies nicht auf die Messung niederschlägt, wurde sowohl für die alte als auch die neue BASE-Version zur Messung des Speicherverbrauchs die Garbage Collection manuell alle 50ms aktiviert. Auf diese Weise lassen sich über längere Zeit angesammelte und überflüssige Speicherbereiche, die die Messergebnisse unnötigerweise verfälschen, größtenteils vermeiden. Allerdings ist die Garbage Collection relativ teuer, weshalb eine zu häufige Aktivierung dieser nicht sinnvoll ist. Somit ist immer noch mit kleinen Ungenauigkeiten zu

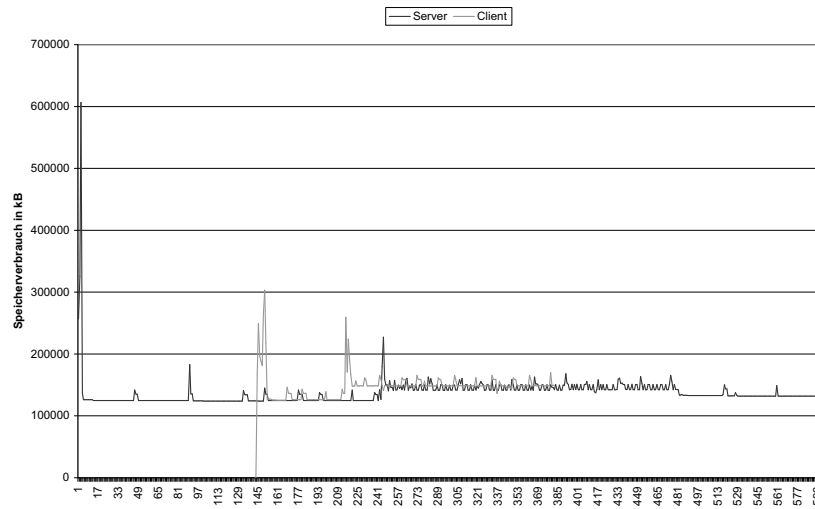


Abbildung 8.7: Verlauf der Speicherauslastung der alten BASE-Version

rechnen. Im Mittel sollten jedoch brauchbare Werte ermittelt werden.

Nichtfunktionale Anforderungen

In Bezug auf den Speicherverbrauch und den Nachrichtenversand ist natürlich die Größe einer nichtfunktionalen Anforderung nicht ohne weiteres zu vernachlässigen. Allerdings können auch hier keine pauschalen Aussagen getroffen werden, da der Umfang einer Anforderung wie bereits erwähnt von Fall zu Fall unterschiedlich ausfallen kann. Folgende Aussagen können dennoch gemacht werden. Insgesamt existieren zur Spezifikation nichtfunktionaler Anforderungen zwei neue Klassen *NFDimension* und *NFDimensionCollection*. Insofern ist der statische Speicherverbrauch im Vergleich zur restlichen Verteilungsinfrastruktur marginal. Eine einfache Anforderung mit einem einzigen Eintrag erfordert zur Laufzeit 96 Bytes im Speicher. Dieser Wert wurde mit einem Profiling-Tool zur Laufzeit erhoben. Inklusive der erforderlichen Markierungen innerhalb der serialisierten Anforderung würde dies in einer Nachrichtenlänge von mindestens 115 Bytes resultieren. Da eine nichtfunktionale Anforderung allein kaum Informationen enthält, ist somit mit einem beinahe linearen Anstieg bei entsprechender Mehrverwendung an Einträgen zu rechnen.

8.3 Nutzwert

Eine quantitative Bewertung des Nutzens, den die in dieser Arbeit entwickelte Erweiterung bringt, ist natürlich nicht möglich. Insofern kann nur eine zu einem gewissen Teil subjektive Bewertung abgegeben werden.

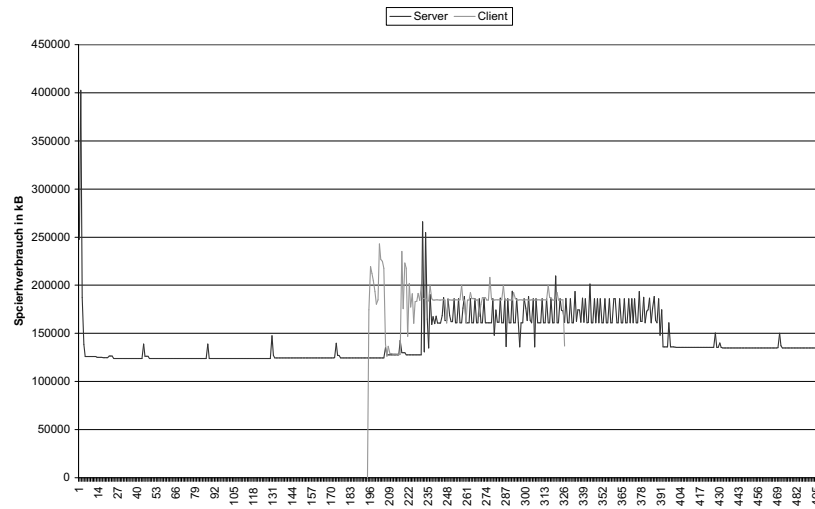


Abbildung 8.8: Verlauf der Speicherauslastung der neuen BASE-Version

Spezifikation nichtfunktionaler Anforderungen

Der Aufwand zur Erstellung nichtfunktionaler Anforderungen ist auf Grund der Anzahl der notwendigen Klassen und der relativ einfachen Schnittstelle aus Sicht des Komponentenentwicklers verhältnismäßig einfach. Tatsächlich hängt der jeweilige Aufwand zur Spezifikation konkreter Anforderungen vom Anwendungsszenario ab und kann somit nicht pauschal behandelt werden. Auch die Abfrage einzelner Eigenschaften aus einer gegebenen nichtfunktionalen Anforderung ist relativ einfach und auf Grund der Implementierung im Wesentlichen mit den üblichen Hash-Tabellen von Java zu vergleichen.

Selektionsstrategien

Im Vergleich zur Spezifikation nichtfunktionaler Anforderungen ist die Entwicklung angemessener Selektionsstrategien zwar bezüglich der Integration in die Verteilungsinfrastruktur relativ einfach, dennoch ist die sinnvolle Entwicklung geeigneter Verfahren zur Erfüllung nichtfunktionaler Anforderungen an sich verhältnismäßig komplex und führt somit zu einem nicht unerheblichen Aufwand für Strategieentwickler. Wie die exemplarischen Selektionsstrategien gezeigt haben, erfordern im Allgemeinen Selektionsstrategien zur Laufzeit einen mehr oder weniger hohen Aufwand. Dieser steht in Zusammenhang mit dem jeweils realisierten Aspekt und kann deshalb nicht allgemein gültig festgestellt werden.

Kapitel 9

Ausblick und zukünftige Erweiterungen

Zum Schluß sollen ein Fazit und ein Ausblick in die Zukunft diese Arbeit abrunden.

9.1 Fazit

Die Notwendigkeit der Unterstützung nichtfunktionaler Eigenschaften und Anforderungen wurde im Einführungskapitel aufgezeigt. Es hat sich auch herausgestellt, dass eine solche Unterstützung nicht einfach umzusetzen ist. Auf unterschiedlichen Ebenen müssen sowohl konzeptionell als auch in der Umsetzung wesentliche Aspekte beachtet und behandelt werden. Auf Ebene der Komponenten muss eine Möglichkeit zur Beschreibung nichtfunktionaler Anforderungen existieren. Auf Ebene der Verteilungsinfrastruktur müssen diese effektiv durch den flexiblen Einsatz von Selektionsstrategien verarbeitet werden.

Zu einem gewissen Grad schlägt sich dies natürlich auch in der Performanz der Verteilungsinfrastruktur nieder. Vorteilhafterweise hält sich der zusätzliche Aufwand zur Laufzeit in Grenzen, so dass die Implementierung der vorliegenden Arbeit den Einsatz und die Weiterentwicklung von BASE und PCOM nicht negativ beeinflusst oder sogar verhindert. Im Gegenteil, die Ergebnisse der Evaluation sind überraschender Weise sogar besser als erwartet.

Der Nutzen der erarbeiteten Erweiterung kann letztlich nur durch deren praktische Anwendung festgestellt werden. Hierzu ist natürlich die Implementierung weiterer Selektionsstrategien erforderlich, die in der Praxis tatsächlich eingesetzt werden. Nur auf diese Weise können eventuelle Grenzen der Erweiterung sowohl in Bezug auf Anwendbarkeit und Mächtigkeit als auch in Bezug auf die Skalierbarkeit ausgelotet werden. Da die Implementierung von Selekt-

tionsstrategien einen gewissen Aufwand für Komponentenentwickler bedeutet, konnten im Rahmen dieser Arbeit leider nur einige einfache Selektionsstrategien umgesetzt werden. Allerdings haben schon diese Selektionsstrategien gezeigt, dass bereits mit einfachen Mitteln eine effektive Selektion von Plugins zum Beispiel in Bezug auf Übertragungsgeschwindigkeit oder Energieverbrauch durchgeführt werden kann. Da die typische Zielgruppe der Verteilungsinfrastruktur hauptsächlich ressourcenarme Rechereinheiten sind, ist künftig allerdings weniger mit umfangreichen Selektionsstrategien zu rechnen.

Aus Sicht der Komponentenentwicklung haben Entwickler keinen zusätzlichen Aufwand durch die Erweiterung zu erwarten, sofern sie auf die Unterstützung nichtfunktionaler Eigenschaften verzichten. In diesem Fall ist der wesentliche Vorteil der Erweiterung die bessere Zeitperformanz der Verteilungsinfrastruktur, sofern zumindest die eingesetzte Java Virtual Machine einen JIT-Compiler verwendet.

9.2 Zukünftige Erweiterungen

Einige Aspekte, deren Behandlung aus Zeitgründen für künftige Arbeiten vorgesehen wurde, stellen sinnvolle Erweiterungen dieser Arbeit und auch der Verteilungsinfrastruktur BASE dar.

Integration in die Komponentenselektion

Die Unterstützung nichtfunktionaler Anforderungen kann nun während der Kommunikation zwischen Komponenten angewendet werden. Allerdings ist eine Verknüpfung sowohl der PCOM-Kontrakte und der darin enthaltenen nichtfunktionalen Eigenschaften als auch der nichtfunktionalen Anforderungen aus dieser Erweiterung sinnvoll. Der Grund hierfür ist, dass, sofern eine Komponente eine andere Komponente benötigt, PCOM die Suche einer passenden Komponente auf Basis von Kontrakten übernimmt. Diesbezüglich spielt die Unterstützung indirekter nichtfunktionaler Eigenschaften unter Umständen eine wichtige Rolle bei der späteren Realisierung direkter nichtfunktionaler Eigenschaften in PCOM. Manche direkten Eigenschaften, die in einem solchen Kontrakt spezifiziert werden, können unter Umständen nur genau dann realisiert werden, wenn bestimmte indirekte, nichtfunktionale Eigenschaften ebenfalls erfüllt werden. Eine getrennte Betrachtung der direkten Eigenschaften in PCOM und der indirekten Eigenschaften dieser Erweiterung führt dazu, dass sofern eine Beziehung zwischen diesen besteht, zuerst eine „PCOM-verträgliche“ Komponente gesucht und gebunden wird, die die direkten Eigenschaften erfüllt.

Anschließend kann erst während der Kommunikation mit dieser Komponente die Erfüllbarkeit indirekter nichtfunktionaler Eigenschaften überprüft werden. Das hat zur Folge, dass bei Unerfüllbarkeit letzterer die Bindung zur genutzten Komponente aufgehoben werden muss und ersatzweise solange andere Komponenten gebunden werden, bis die indirekten Eigenschaften ebenfalls erfüllt werden. Der Aufwand zur Laufzeit ist in der derzeitigen Realisierung natürlich größer als unbedingt notwendig.

Sinnvoller wäre es, wenn PCOM bereits während der Suche nach passenden Komponenten, eine repräsentative Anforderung von einer Komponente erhalten und bei der Suche nach anderen Komponenten berücksichtigen würde. Auf diese Weise würden zuerst die Anforderungen an die Instanzen der Verteilungsinfrastruktur gestellt und sofern diese die Anforderung erfüllen können, die Aushandlung der PCom-Kontrakte durchgeführt. Unnötige Bindungsvorgänge könnten auf diese Weise vermieden werden.

Automatische generierte nichtfunktionale Anforderungen

Derzeit müssen Komponentenentwickler Code in den Komponenten vorsehen, der die jeweiligen nichtfunktionalen Anforderung erzeugt. Da auf diese Weise der eigentliche Code der Komponente mit dem Aspekt der Beschreibung nichtfunktionaler Anforderungen vermischt wird, wäre es vorteilhaft, den Code zur Erzeugung nichtfunktionaler Anforderungen möglichst minimal zu halten. Dies kann beispielsweise erreicht werden, indem der Code gekapselt und lediglich durch eine spezielle Funktion zur Laufzeit geladen wird. Auf diese Weise ist es darüber hinaus möglich, existierende und invariante nichtfunktionale Anforderungen zwischen mehreren Komponenten wiederzuverwenden.

Lokale Ressourcenverwaltung

Es wurde ausführlich dargelegt, dass die Erfüllbarkeit nichtfunktionaler Anforderungen unter anderem stark von den verfügbaren Ressourcen abhängt. Da zumindest im Rahmen dieser Arbeit Selektionsstrategien diese Abhängigkeiten selbst beachten und notwendige Informationen erheben müssen, ist eine zentrale Ressourcenverwaltung, auf die Selektionsstrategien zurückgreifen können, eventuell sinnvoll. Hierzu müsste eine generische Schnittstelle zur Abfrage von verfügbaren Ressourcen zur Verfügung gestellt werden. Der wesentliche Vorteil einer zentralen Ressourcenverwaltung ist, dass die Entwicklung von Selektionsstrategien wesentlich einfacher wird, sofern letztere die verfügbaren Ressourcen als Abhängigkeitsfaktoren betrachten müssen.

Vollständige Unterstützung nichtfunktionaler Eigenschaften innerhalb der Verteilungsinfrastruktur

In dieser Arbeit wurde die Unterstützung nichtfunktionaler Eigenschaften ausschließlich innerhalb der Plugin-Architektur realisiert. Infolgedessen wurde die Unterstützung durch Broker, Proxies und andere Objekte, die zur Verteilungsinfrastruktur gehören, nicht näher berücksichtigt. Wie schon erwähnt wurde, erfordern jedoch gerade vertikale nichtfunktionale Eigenschaften eine Unterstützung innerhalb der gesamten Kommunikationskette. In dieser Arbeit wurde davon ausgegangen, dass der Einfluss dieser Objekte auf vertikale Eigenschaften zur Vereinfachung vernachlässigt werden kann. Ob dies tatsächlich der Fall ist, kann nur durch den mittelfristigen Einsatz der Erweiterung in praktischen Anwendungsszenarien ermittelt werden. Sollte sich herausstellen, dass eine Unterstützung tatsächlich auch in höher angelegten Schichten der Verteilungsinfrastruktur in der Praxis erforderlich ist, muss natürlich eine erweiterte Unterstützung in Erwägung gezogen werden.

Konfiguration von Plugins

In der derzeitigen Implementierung der Plugin-Architektur wird davon ausgegangen, dass eventuelle Konfigurationsparameter direkt in der Plugin-Implementierung festgelegt werden. Dadurch wird die Möglichkeit, ohne erneutes Kompilieren der entsprechenden Klassen Änderungen an diesen Konfigurationsparametern vorzunehmen, verhindert. Um diesen Nachteil zu vermeiden ist ein Konfigurations-Mechanismus denkbar, der zu Beginn der Laufzeit die Plugin-Implementierungen mit den notwendigen Parametern versorgt. Administratoren wäre dadurch die Möglichkeit gegeben, Plugins auf einfache Art und Weise zu konfigurieren.

Tabellenverzeichnis

8.1 Testumgebung 79

Abbildungsverzeichnis

3.1	Die Kommunikationskette zwischen Komponenten	19
3.2	Anforderungsbeschreibung einer Komponente	22
3.3	Angebotsbeschreibung einer Komponente	22
4.1	Originäre Plugin-Architektur	30
4.2	Restrukturierte Plugin-Architektur	33
6.1	Kooperation zwischen Selektionsstrategien	62
7.1	Schichtenmodell	68
7.2	Plugins und ihre Beschreibungen	69
7.3	Plugin-Verwaltung	70
7.4	Verarbeitung von Aufrufen und Nachrichten	71
7.5	Transformation eines Aufrufes in eine Nachricht	72
7.6	Transformation eines Aufrufes in eine Nachricht	72
7.7	Transformation einer Nachricht in einen Aufruf	72
7.8	Entwurf der Selektionsstrategien	73
7.9	Nichtfunktionale Beschreibungen	75
8.1	Vergleich durchschnittliche Aufrufdauern	81
8.2	Alte BASE-Version ohne JIT	82
8.3	Alte BASE-Version mit JIT	83
8.4	Neue BASE-Version ohne JIT	83
8.5	Neue BASE-Version mit JIT	84
8.6	Vergleich der Speicherauslastungen beider BASE-Versionen	85
8.7	Verlauf der Speicherauslastung der alten BASE-Version	86
8.8	Verlauf der Speicherauslastung der neuen BASE-Version	87

Literaturverzeichnis

- [1] Christian Becker, Gregor Schiele, Holger Gubbels, *Base – A Micro–Kernel–based Middleware For Pervasive Computing*, Proceedings of the IEEE International Conference on Pervasive Computing and Communication, Fort Worth, USA
- [2] Christian Becker, Gregor Schiele, *Middleware and Application Adaption Requirements and their Support in Pervasive Computing*, Proceedings "3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems", ICDCS, Providence, USA
- [3] Christian Becker, *Dienstgütemanagement in verteilten Objektsystemen*, Dissertation, Johann-Wolfgang Goethe Universität, Frankfurt am Main
- [4] Christian Becker, Kurt Gheis, *MAQS Management for Adaptive QoS-enabled Services*, Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, 1997
- [5] Christian Becker, Kurt Gheis, Generic QoS Support for Corba, Fifth IEEE Symposium on Computers and Communications (ISCC 2000)
- [6] S. Demurjian, K. Bessette, T. Doan C. Phillips, *Middleware Security*, Computer Science & Engineering Dept. The University of Connecticut Dept. of EE & CS, United States Military Academy
- [7] Foundstone Inc. and CORE Security Technologies, *Security in the Microsoft .NET Framework*
- [8] Sven Frolund, Jari Koistinen, *QML: A language for Quality of Service Specification*, Software Technology Laboratory, Hewlett-Packard
- [9] Sven Frolund, Jari Koistinen, *Quality of Service Specification in Distributed Object Systems Design*, Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS) Santa Fe, New Mexico, 1998

- [10] Kurt Gheis, Torben Weis, Adreas Ulbrich, *QoS Mechanism Composition at Design-Time and Runtime*, 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)
- [11] Xiaohui Gu, Dongyan Xu, Duangdao Wichadakul, Klara Nahrstedt, Baochun Li, *QoS-Aware Middleware for Ubiquitous and Heterogenous Environments*, Department of Computer Science, University of Illinois
- [12] Holger Gubbels, *Basisdienste für Anwendungen in ubiquitären Rechnersystemen*, Universität Stuttgart, 2002
- [13] Marcus Handte, *Entwicklung eines Komponentensystems für Anwendungen in ubiquitären Rechnersystemen*, Universität Stuttgart, 2003
- [14] Stefan Hild, *Managing Mobile Connections*, IEEE The Magazine of Nomadic Communication and Computing Vol. 4 No. 5, 1997
- [15] Anthony D. Joseph, Joshua A. Tauber, M. Franz Kaashoek, *Mobile Computing with The Rover Toolkit*, IEEE Transactions on Computer Systems, 1997
- [16] Tim Kindberg, Armando Fox, *System Software for Ubiquitous Computing*, IEEE Pervasive Computing, 2002
- [17] Fred Kuhns, Carlos O'Ryan, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons, *The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware*, Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks (PfHSN '99), Salem, MA, 1999
- [18] Roland Kurmann, *Spontane Vernetzung - Dienstbeschreibung und Service Discovery*, Fachseminar SS 2000, Abteilung Verteilte Systeme, ETH Zürich
- [19] Arvind Krishna, Douglas C. Schmidt, Raymond Klefstad, and Angelo Cor-saro, *Towards Predictable Real-time Java Object Request Brokers*, Proceedings of the 9th IEEE Real-time/Embedded Technology and Applications Symposium (RTAS), Toronto, Canada, May 2003.
- [20] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, Kenneth R. Anderson, *QoS Asepct Languages and Their Runtime Integration*, BBN Technologies/GTE Internet-working, Cambridge, MA 02138, USA, <http://quo.bbn.com/>

- [21] Friedemann Mattern, *Vom Verschwinden des Computers - Die Vision des Ubiquitous Computing*, Total Vernetzt, Springer Verlag 2003
- [22] Shivajit Mohapatra, Nalini Venkatasubramanian, *PARM: Power Aware Reconfigurable Middleware*, IEEE International Conference on Distributed Computing Systems, 2002
- [23] Brian D. Noble, *Mobile Data Access*, Phd. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 1998
- [24] Object Management Group, *Common Object Request Broker Architecture: Core Specification 3.02*, <http://www.omg.org/>
- [25] Object Management Group, *Common Secure Interoperability (CSIv2)*, <http://www.omg.org/>
- [26] Object Management Group, *Fault-tolerant Corba*, <http://www.omg.org/>
- [27] Object Management Group, *Realtime Corba Specification*, <http://www.omg.org/>
- [28] Object Management Group, *Unified Modelling Language*, <http://www.omg.org/>
- [29] Peer to Peer Pervasive Computing, <http://3pc.info>
- [30] Jarno Rajahalme, Telma Mota, Frank Steegmans, Per F. Hansen, Fernanda Fonseca, *Quality of Service Negotiation in TINA*, IEEE Global Convergence of Telecommunication and Distributed Object Computing, 1997
- [31] Saltzer, Reed, Clark, *End-to-End Arguments in System Design*, IEEE International Conference on Distributed Computing Systems, 1981
- [32] Michael Samulowitz, *Kontextadaptive Dienstrutzung in Ubiquitous Computing Umgebungen*, Dissertation, Fakultät für Mathematik, Informatik und Statistik, Ludwig-Maximilians-Universität, München
- [33] Softwired AG, *ELECTRA - An Object Request Broker for Reliable Distributed Systems*, Schweiz, <http://www.softwired.ch/people/maffeis/electra.html>
- [34] Sun Microsystems, *Java 2 Platform Enterprise Edition Specification*, <http://java.sun.com/>
- [35] Sun Microsystems, *Java 2 Micro Edition Specification*, <http://java.sun.com/>

- [36] Marc Weiser, *How computers will be differently used in the next twenty years*, 1999
- [37] Marc Weiser, *Some Computer Science Issues in Ubiquitous Computing*, Communications of the ACM, Juli 1993
- [38] Mark Weiser, *The Computer for the twenty-first Century*, Scientific American, 1991

*Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst
und bei der Erstellung nur die angegebenen Quellen verwendet habe.*

Alexander Rau