Universität Stuttgart
Fakultät Informatik

Prüfer	Prof. Dr. Kurt Rothermel
Betreuer	Dr. Christian Becker
	Dipl.-Inf. Gregor Schiele
Beginn	1. Februar 2003
Ende	31. Juli 2003
CR-Klassifikation	C.2.4, D.2.11, D.3.3

Diplomarbeit 2074

Entwicklung eines Komponentensystems für Anwendungen in ubiquitären Rechnersystemen

Marcus Handte

Institut für
Parallele und Verteilte Systeme

Abteilung
Verteilte Systeme

Kurzfassung

Ubiquitäre Rechnersysteme stellen hohe Anforderungen an Anwendungen. Die Heterogenität und Mobilität der Rechner in ubiquitären Rechnersystemen sowie der steigende Bedarf an kontextuellen Informationen erfordern Abstraktionen, die in der Lage sind, diese Anforderungen zu reduzieren. Softwarekomponenten stellen eine geeignete Abstraktion dar, mit deren Hilfe die Komplexität von Anwendungen für ubiquitäre Rechnersysteme gesenkt werden kann.

Im Laufe dieser Arbeit werden die Anforderungen an Verteilungsinfrastrukturen zur Entwicklung von Anwendungen für ubiquitäre Rechnersysteme erarbeitet. Entlang dieser werden bestehende Arbeiten vorgestellt und verglichen. Dabei zeigt sich, dass existierende Arbeiten nicht alle Anforderungen zufrieden stellend abdecken.

Im Anschluss an die Diskussion existierender Arbeiten wird ein Ansatz für ein Komponentensystem vorgeschlagen, mit dem auf die Anforderungen eingegangen wird. Dieser dient als Grundlage für die darauf folgende Ausarbeitung der Konzepte und deren Implementierung.

Ausgehend von der Implementierung werden die Kosten für den Einsatz von komponentenbasierten Abstraktionen abgeschätzt und mit denen für den Einsatz von dienstbasierten verglichen. Zum Abschluss der Arbeit werden weitere Entwicklungsmöglichkeiten vorgestellt und zukünftige Arbeiten motiviert.

-

Diese Arbeit widme ich meinen Eltern.

Vielen Dank für Eure unermüdliche Unterstützung.

-

Frickenhausen, Juli 2003

Inhaltsverzeichnis

1 Einleitung 1
1.1 Übersicht 1
1.2 Grundlagen 1
1.2.1 Komponentensoftware 1
1.2.2 Ubiquitous Computing 3
1.3 Motivation 5
2 Anforderungen 8
2.1 Heterogenität 8
2.1.1 Verständlichkeit 8
2.1.2 Minimalität 9
2.1.3 Interoperabilität 9
2.2 Mobilität 10
2.2.1 Integration 10
2.2.2 Adaption 11
2.2.3 Administration 11
2.3 Kontext 12
2.3.1 Spezifikation 12
2.3.2 Verbreitung 13
2.3.3 Verarbeitung 13
3 Existierende Ansätze 14
3.1 Klassische Verteilungsinfrastrukturen 14
3.1.1 Objekt- bzw. Dienstorientiert 14
3.1.2 Komponentenbasiert 15
3.2 Ressourcenoptimierte Verteilungsinfrastrukturen 16
3.2.1 Jini 16
3.2.2 Puppeteer 16
3.2.3 Universally Interoperable Core 18
3.3 Verteilungsinfrastrukturen für Ubiquitous Computing 19
3.3.1 one.world 19
3.3.2 BASE 20
3.3.3 Gaia 21
3.3.4 Aura 23
3.3.5 Sparkle 24

3.4 Vergleich 25
4 Lösungsansatz 28
4.1 Diskussion 28
4.2 Zielsetzung 32
4.3 Beispiel 34
4.4 Abgrenzung 37
4.5 Einschränkungen 38
5 Konzeption 40
5.1 Plattform 40
5.2 Komponentenmodell 43
5.2.1 Anwendungsmodell 44
5.2.2 Programmeinheit 48
5.2.3 Beschreibung 52
5.2.4 Verhandlung 58
5.2.5 Adaption 61
5.3 Erweiterungen 63
6 Implementierung 65
6.1 Schichtenmodell 65
6.2 Statisches Modell 66
6.2.1 Instance 67
6.2.2 Factory 69
6.2.3 Container 70
6.2.4 Contract 71
6.3 Dynamisches Modell 73
6.3.1 Aufruf 73
6.3.2 Verhandlung 74
6.3.3 Instanzierung 76
6.3.4 Fehlerbehandlung 77
7 Evaluation 79
7.1 Implementierung 79
7.1.1 Messung 80
7.1.2 Auswertung 80
7.2 Speicherbedarf 81
7.2.1 Messung 82
7.2.2 Auswertung 83

7.3 Geschwindigkeit 84
7.3.1 Messung 84
7.3.2 Auswertung 85
8 Zusammenfassung 86
8.1 Ergebnisse 86
8.2 Ausblick 88
8.3 Fazit 90
9 Literaturverzeichnis 92

Abbildungsverzeichnis

Abbildung 1 – Puppeteer Architektur 17
Abbildung 2 – BASE Architektur 20
Abbildung 3 – Gaia Architektur 22
Abbildung 4 – Aura Architektur 23
Abbildung 5 – Sparkle Architektur 24
Abbildung 6 – Komposition von Anwendungen 34
Abbildung 7 – Einbettung kontextueller Information 35
Abbildung 8 – Umgang mit Mobilität 36
Abbildung 9 – Identifikation von Anwendungen 45
Abbildung 10 – Zyklische Abhängigkeiten 45
Abbildung 11 – Richtung von Aufrufen 47
Abbildung 12 – Lebenszyklus für Komponenteninstanzen 51
Abbildung 13 – Struktur von Beschreibungen 57
Abbildung 14 – Ineffiziente Bestimmung von Parametern 60
Abbildung 15 – Schichtenmodell 66
Abbildung 16 – Umsetzung von Komponenteninstanzen 67
Abbildung 17 – Umsetzung von Factories 69
Abbildung 18 – Umsetzung des Containers 70
Abbildung 19 – Umsetzung von Beschreibungen 72
Abbildung 20 – Aufruf mit Adressraumwechsel 73
Abbildung 21 – Aufruf ohne Adressraumwechsel 74
Abbildung 22 – Verhandlungen 75
Abbildung 23 – Instanzierung von Komponenten 76
Abbildung 24 – Bindungsspezifische Fehlerbehandlung 77

Tabellenverzeichnis

Tabelle 1 – Umfang und Komplexität.....80
Tabelle 2 – Speicherbedarf.....83
Tabelle 3 – Geschwindigkeit.....85

1 Einleitung

Die besonderen Anforderungen von Ubiquitous Computing machen angepasste Verteilungsinfrastrukturen unabdingbar. Die vorliegende Arbeit dokumentiert die Entwicklung eines Komponentensystems für Anwendungen in ubiquitären Rechnersystemen.

Die Arbeit ist Teil des Projekts *Communication in Adhoc Networks for Ubiquitous Computing (CANU)*, das am *Institut für Verteilte und Parallele Systeme* der *Universität Stuttgart* durchgeführt wird und sich vorrangig mit Informationsausbreitung, Routing und Anwendungsarchitekturen ubiquitärer Rechnersysteme beschäftigt. Der Fokus liegt dabei auf mobilen ad hoc Netzen und Smart Environments.

1.1 Übersicht

Nachdem in diesem Kapitel zunächst die grundlegenden Begriffe erläutert werden und die Motivation für die Entwicklung eines Komponentensystems für ubiquitäre Rechnersysteme gegeben wird, werden im darauf folgenden Kapitel die Anforderungen an ein solches System skizziert. Daraufhin werden in Kapitel drei bestehende Ansätze vorgestellt, klassifiziert und entlang den Anforderungen verglichen. Im vierten Kapitel wird der Lösungsansatz dieser Arbeit vorgestellt, motiviert und dessen Ziele erläutert. Das fünfte Kapitel beschreibt notwendige Konzepte und diskutiert mögliche Alternativen. Im sechsten Kapitel wird auf die technische Realisierung eingegangen. Kapitel sieben evaluiert die Umsetzung entlang technischer Parameter. Im letzten Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst, zukünftige Arbeiten motiviert und zum Abschluss der Arbeit wird ein Fazit gezogen.

1.2 Grundlagen

Da sich diese Arbeit mit der Entwicklung eines Komponentensystems für Anwendungen in ubiquitären Rechnersystemen beschäftigt, müssen zunächst die grundlegenden Begriffe aus den Bereichen Komponentensoftware und Ubiquitous Computing genauer beleuchtet werden. Dabei wird neben konkreten Definitionen auch ein kurzer Überblick über den Hintergrund der Begriffe gegeben.

1.2.1 Komponentensoftware

Software, die aus *Softwarekomponenten* zusammengesetzt ist, heißt *Komponentensoftware*. Abkürzend werden Softwarekomponenten in der Regel auch ein-

fach als *Komponenten* bezeichnet. Szyperski definiert Softwarekomponenten wie folgt [Szyperski98]:

„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Komponentensoftware ist demnach Software, die aus Teilen besteht, die nur explizite Abhängigkeiten und vertraglich spezifizierte Schnittstellen haben. Aus diesem Grund können sie unabhängig voneinander entwickelt, gewartet, installiert und zusammengesetzt werden.

Damit Softwarekomponenten trotz hoher Unabhängigkeit interoperabel sind, bedarf es standardisierter Schnittstellen, Mechanismen und Methoden, die ein reibungsloses Zusammenspiel ermöglichen. Nach Szyperski wird die Definition eines solchen Standards *Komponentenframework* genannt.

„A component framework is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level."

Ein *Komponentensystem* ist die konkrete Umsetzung von einem oder mehreren Komponentenframeworks, wobei die Regeln für das Zusammenwirken der einzelnen Komponentenframeworks festgelegt werden. Da sich das Komponentensystem dieser Arbeit vorwiegend mit Aspekten der Verteilung beschäftigt, wird es im Folgenden auch als *komponentenbasierte Verteilungsinfrastruktur* bezeichnet.

Aus heutiger Sicht ist Komponentensoftware eine logische Folgerung des Software Engineerings, auch wenn viele Schritte in Richtung Komponenten durch die Entwicklung neuer Programmiersprachen vorangetrieben wurden. Komponentensoftware ist ein Resultat jahrzehntelanger Versuche, die Produktivität der Softwareentwicklung durch getrennte Entwicklung, Wartung und Test von Programmteilen zu steigern.

In den Achtzigern wurde mit der Einführung von Modulen in Ada und Modula-2 die Möglichkeit zur Entwicklung modularer Software geschaffen. Ein wesentlicher Vorteil der Modularisierung ging aus der getrennten Kompilierbarkeit hervor. Anstatt das gesamte Programm nach jeder Änderung neu zu kompilieren, konnte es entlang von Modulgrenzen geändert werden. Mit dieser Neuerung ermöglichten Module die Distribution von Bibliotheken in binärer Form.

Modularisierung war zwar ein Schritt in Richtung Softwarekomponenten. Jedoch führten vor allem die unklare Definition von Schnittstellen und die daraus resultierende Inkompatibilität dazu, dass Module im Allgemeinen nicht getrennt entwickelt werden konnten und nicht getrennt wartbar waren.

Modernere Programmiersprachen wie Eiffel, versuchen diesen Misstand auf Basis vertraglich spezifizierter Schnittstellen zu beseitigen. Allerdings unterstützt Eiffel ausschließlich Verträge die Syntax und Verhalten spezifizieren. Nicht funktionale Anforderungen, wie beispielsweise Dienstgüte, können bisweilen nicht in Eiffel ausgedrückt werden.

Rund zehn Jahre nach der Einführung von Modulen wurde Objektorientierung mit der Entwicklung der Programmiersprache Java zu einem beliebten Paradigma. Konzepte, wie Vererbung, erlaubten zwar neue Formen der Wiederverwendung, erforderten aber gleichzeitig detaillierte Kenntnisse und hohe Kontrolle der Beschaffenheit von Superklassen (Fragile Baseclass Problem).

Softwarekomponenten umgehen die Probleme von Klassen und Modulen durch eine kompositionale Form der Wiederverwendung und der Forderung nach expliziten, vertraglich spezifizierten Schnittstellen und Abhängigkeiten. Objektorientierte Vererbung wird in diesem Zusammenhang häufig als Whitebox-Wiederverwendung bezeichnet. Bei Softwarekomponenten spricht man hingegen von einem Blackbox-Ansatz. Da Softwarekomponenten einzig aufgrund ihrer spezifizierten Schnittstellen und Abhängigkeiten zusammengesetzt werden können, umschiffen sie rein theoretisch die Probleme, die bei getrennter Wartung entstehen.

Um Komponentensoftware zu ermöglichen, muss ein Komponentensystem, über integrierte Komponentenframeworks, Standards im Hinblick auf Komponenteninteraktion festschreiben. Ohne solche Standards kann nicht gewährleistet werden, dass Komponenten, die unabhängig voneinander entwickelt wurden, kooperieren können.

Die Definition dieser Standards ist eine Gratwanderung. Werden an Softwarekomponenten zu viele Anforderungen gestellt, kann es zu unnötigen Einschränkungen kommen. Dagegen kann ein ungenauer oder unvollständiger Standard zu inkompatiblen Interpretationen führen.

1.2.2 Ubiquitous Computing

Ubiquitous Computing [Weiser91] bezeichnet ein Gebiet der Informatik, das sich mit allgegenwärtigen integrierten Rechnern beschäftigt. Ziel ist es, Rechner unsichtbar in eine möglichst große Zahl von Alltagsgegenständen einzubetten. Durch die Kooperation dieser allgegenwärtigen Rechner sollen Nutzer durchgängig bei ihren Aufgaben unterstützt werden. Natürliche Interaktion zwischen Mensch und Rechner sorgt dafür, dass der Gebrauch von Rechnern weitgehend unsichtbar wird. Rechner erweitern die Realität somit nahtlos um eine virtuelle Dimension.

Zu Begin der neunziger Jahre formulierte Mark Weiser die Vision des Ubiquitous Computing [Weiser91]. Die Idee, alltägliche Aufgaben mit Rechnersys-

temen zu unterstützen, ohne dabei von der Aufgabe abzulenken, war zu diesem Zeitpunkt sicherlich keine Revolution sondern eher Evolution. Den fundamentalen Unterschied zwischen damaligen Entwicklungen wie intelligenten Agenten oder virtueller Realität und der Vision der ubiquitären Rechnersysteme formulierte Mark Weiser beeindruckend verständlich und plakativ.

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

Bei Ubiquitous Computing geht es daher nicht um den Versuch, neue oder effizientere Formen der Interaktion zu entwickeln, sondern Bekanntes und Bewährtes zur unsichtbaren Integration von Rechnern zu nutzen.

Betrachtet man die heutige Entwicklung aus dieser Perspektive, steht fest, dass große Teile von Weisers Vision bereits Einzug in die Realität gehalten haben. Unsichtbare, eingebettete Systeme sind überall. Im Automobil, in der Mikrowelle oder der Kaffeemaschine unterstützen sie Nutzer allgegenwärtig bei der Ausführung von Aufgaben.

Trotzdem gibt es zwischen Weisers Vision und dem derzeitigen Stand der Technik zumindest eine große Differenz. Ubiquitous Computing zielt auf eine möglichst lückenlose und umfassende Unterstützung von Nutzern ab. Am Beispiel seiner damaligen Rechnerprototypen unterstreicht Weiser die Bedeutung rechnerübergreifender Kooperation zur Erfüllung dieser Forderung.

„The real power of the concept comes not from any of these devices; it emerges from the interaction of all of them."

Damit ist klar, dass wir selbst ein Jahrzehnt nach dem Startschuss der neuen Ära noch weit von Ubiquitous Computing entfernt sind. Zwar sind wir in der Regel von Rechnern umgeben, doch eine Kooperation dieser ist in den meisten Fällen nicht oder nur bedingt möglich. Selbst die intuitive Interaktion zwischen Mensch und Rechner ist in vielen Fällen noch Wunschdenken.

Ubiquitous Computing verspricht eine Revolution der Nutzung von Informationstechnologie. Die technischen Anforderungen zur Realisierung sind überwältigend und durchdringen nahezu alle Gebiete der Informatik [Weiser93a].

Die Integration von Rechnern in Alltagsgegenstände fordert die Anpassung von Rechnerarchitekturen und Fertigungsverfahren. Ubiquitous Computing erfordert dabei nicht nur billigere oder schnellere Technologie. Für Szenarien, in denen kleine, mobile Gegenstände wie Plastikkarten oder Verpackungen erweitert werden, wird winzige energiesparende Hardware benötigt.

Intuitive Bedienung stellt hohe Ansprüche im Bereich der Mensch-Rechner-Interaktion. Natürliche Schnittstellen oder die Bereitstellung und Nutzung

kontextueller Informationen sind nur einige Anforderungen, die Ubiquitous Computing stellt [Abowd00].

Kooperation zwischen verschiedenen Rechnern kann nicht ohne eine entsprechende Infrastruktur realisiert werden. Unterstützung von ad hoc Netzwerken und heterogenen Geräten in hoch dynamischen Umgebungen sind Aufgaben der Netzwerktechnik [Weiser93b] und Middleware [Becker03a].

Mobilität und spontane Vernetzung, eine unweigerliche Folge ubiquitärer Rechnersysteme, erfordern gleichermaßen neue Anwendungsarchitekturen. Heutige Anwendungen allokieren Ressourcen wie beispielsweise Aus- und Eingabegeräte üblicherweise statisch. Fallen Ressourcen zur Laufzeit aus, führt dies zu Fehlern, die in der Regel nicht abgefangen werden können. In hoch dynamischen Umgebungen werden solche statischen Ressourcenallokationen zum Problem.

1.3 Motivation

Ubiquitous Computing hat das erklärte Ziel, die Rechnerlandschaft von morgen grundlegend zu verändern. Während heutzutage Rechenleistung vor allem durch eine vergleichsweise kleine Zahl leistungsfähiger Personalcomputer, Laptops, Cluster oder Mainframes bereitgestellt wird, werden in Zukunft alle Aufgaben des Alltags von einer Vielzahl heterogener Rechnersysteme unterstützt.

Rechner kooperieren spontan miteinander, um die Aufgaben ihrer Nutzer bestmöglich zu unterstützen. Dabei verfügen sie über unterschiedliche, sich ergänzende Ressourcen. Einfachste Aktuatoren und Sensoren mit wenig Rechenleistung, minimalem Speicher und preiswerter Bluetooth Hardware vernetzen sich mit Kaffeemaschinen, Mobiltelefonen oder Personal Digital Assistants mit schnellen Prozessoren und Wireless LAN Anbindungen. Diese sind ihrerseits mit Laptops, Personalcomputern oder Mainframes verbunden, die über Hochleistungsprozessoren, praktisch unbegrenzten Speicher und breitbandige Internetzugänge verfügen.

Diesem Bild zufolge wird damit die gesamte Welt zu einem verteilten System. Jedoch sind im Gegensatz zu herkömmlichen verteilten Systemen Benutzer und Geräte in hohem Maße mobil. Ad hoc Netzwerke ermöglichen die Kommunikation und damit die Kooperation dieser heterogenen Geräte, um eine optimale Ressourcenausnutzung zu gewährleisten.

Bis zu einem gewissen Grad ist die Transformation heutiger Rechnersysteme zu Ubiquitous Computing Umgebungen mit der Einführung des Personalcomputers vergleichbar. Wie ubiquitäre Rechnersysteme stellten auch Personalcomputer neue Anforderungen an Hard- und Software.

Während Mainframes zunächst Unternehmen vorbehalten waren, ermöglichte der Personalcomputer die Durchdringung privater Haushalte. Damit erweiterte sich die berufliche Anwendung von Informationstechnik um eine private Dimension. Ubiquitous Computing hat den Anspruch, eine ähnliche Expansion einzuleiten, indem es in Zukunft alle Aufgaben des Alltags erleichtert.

Genauso wie Mainframes nicht von Personalcomputern abgelöst wurden, sollen ubiquitäre Rechnersysteme derzeitige Technik nicht schlagartig ersetzen. Vielmehr kann mittelfristig die Koexistenz beider Welten und langfristig ihre Fusion erwartet werden. Rechenintensive Anwendungen ubiquitärer Rechnersysteme werden schließlich weiterhin leistungsfähige Geräte benötigen, und Personalcomputer sowie Mainframes sind wegen ihrer hohen Marktdurchdringung geradezu prädestiniert, diese Anforderungen zu erfüllen.

Ubiquitous Computing unterscheidet sich jedoch in zwei wesentlichen Punkten von der Einführung von Personalcomputern. Während Mainframes und Personalcomputer über ähnliche Mensch-Rechner-Schnittstellen verfügen, ist die Anpassung dieser Schnittstelle per Definition eine Voraussetzung ubiquitärer Rechnersysteme. Fortschreitendes Eindringen von Rechnern ohne gleichzeitige Vereinfachung der Interaktion würde zwangsläufig zu hoher kognitiver Belastung führen. Minimale und natürliche Interaktion ist die Voraussetzung für die unsichtbare Erweiterung von Alltagsgegenständen mit Rechnern.

Zudem sind ubiquitäre Rechnersysteme in höchstem Maße dynamisch. Mobilität von Nutzern und Geräten stellen neue Anforderungen an Anwendungen. Dynamische Vernetzung und schwankende Ressourcenverfügbarkeit durch Mobilität sorgen für explosionsartige Komplexitätssteigerungen. Bei der Entwicklung werden Analysten, Designer, Programmierer und Tester mit einer Vielzahl bekannter und neuartiger Probleme konfrontiert.

Wie bereits erwähnt, sind ubiquitäre Rechnersysteme inhärent verteilt. In Folge ziehen sie die gesamte Komplexität verteilter Anwendungsentwicklung nach sich.

Anwendungen für ubiquitäre Rechnersysteme müssen in die Lage versetzt werden, entfernte Ressourcen zu nutzen. Diese können zur Laufzeit aufgrund von Hard- oder Softwarefehlern ausfallen. Stabile Anwendungen müssen daher permanent eine weitere, nicht triviale Klasse von Sonderfällen behandeln.

Gleichzeitig steigt die Komplexität der Installation und Administration. Soll eine neue Anwendung installiert oder eine bestehende aktualisiert werden, müssen mehrere Systeme simultan verändert werden.

Erschwerend kommt hinzu, dass ubiquitäre Rechnersysteme heterogen sind. Um Interoperabilität zu gewährleisten, müssen Anwendungen die Kommunikation mit verschiedenen Rechnertypen über unterschiedliche Protokolle erlauben.

Des Weiteren bilden sich Netzwerke im ubiquitären Umfeld ad hoc. Damit müssen Anwendungen schwankende Ressourcenverfügbarkeit erkennen und darauf entsprechend reagieren. Während derzeitige Anwendungen üblicherweise fest auf die permanente Verfügbarkeit benötigter Ressourcen ausgerichtet sind, müssen Anwendungen in ubiquitären Rechnersystemen mit wechselnder Ressourcenverfügbarkeit rechnen.

Ubiquitous Computing erfordert einen Paradigmenwechsel. Abstraktionen herkömmlicher Verteilungsinfrastrukturen, wie sie beispielsweise CORBA oder DCOM für die Entwicklung verteilter Anwendungen liefern, müssen an die geänderten Voraussetzungen angepasst werden. In dieser Arbeit wird eine komponentenbasierte Verteilungsinfrastruktur entwickelt, die speziell auf die besonderen Anforderungen ubiquitärer Rechnersysteme eingeht.

2 Anforderungen

Anwendungen für ubiquitäre Rechnersysteme stellen eine Reihe besonderer Anforderungen an die zugrunde liegenden Infrastrukturen [Becker03b, Henricksen01]. Dabei handelt es sich im Wesentlichen um eine Obermenge der Anforderungen herkömmlicher Verteilungsinfrastrukturen, die sich unmittelbar aus den Eigenschaften ubiquitärer Rechnersysteme ableiten lässt.

Zum einen müssen Infrastrukturen für ubiquitäre Rechnersysteme in der Lage sein, mobile und damit spontan vernetzte, heterogene Geräte zu unterstützen. Zum anderen müssen sie ihren Anwendungen kontextuelle Informationen bereitstellen.

2.1 Heterogenität

Ubiquitäre Rechnersysteme sind zwangsläufig heterogen. Dies liegt vor allem daran, dass die unsichtbare Integration von Rechnern in Alltagsgegenstände spezialisierte Hardware erfordert. Zudem ist mittelfristig nicht damit zu rechnen, dass der gesamte Markt ubiquitärer Rechnersysteme von wenigen oder kooperierenden Anbietern abgedeckt wird. Damit wird die Unterstützung heterogener Systeme zu einer wesentlichen Voraussetzung für jede Art von Infrastruktur für ubiquitäre Rechnersysteme. Als direkte Folge lassen sich drei konkrete Anforderungen ableiten.

2.1.1 Verständlichkeit

Eine Verteilungsinfrastruktur für die Entwicklung von Anwendungen in ubiquitären Rechnersystemen muss ein solides Fundament für Anwendungen beliebigen Umfangs bilden. Gerade im Hinblick auf ressourcenarme Geräte, auf denen nur kleine, einfache Anwendungen ausgeführt werden, ist es erforderlich, dass die Infrastruktur mit einer kleinen Zahl verständlicher Konzepte arbeitet und möglichst schmale Schnittstellen bietet.

Während die Einarbeitung in komplexe Modelle und die Nutzung breiter Schnittstellen für komplexe und umfangreiche Anwendungen lohnenswert sein kann, nimmt die Effizienz kleiner Entwicklungen mit zunehmender Komplexität der Infrastruktur drastisch ab. Eine optimale Verteilungsinfrastruktur muss ihre komplexeren Schnittstellen vor einfachen Anwendungen verbergen und erlaubt umfangreichen die fein-granulare, wenn auch möglicherweise komplizierte Steuerung.

Die Forderung nach Verständlichkeit setzt gerade im Bereich ubiquitärer und damit heterogener Rechnersysteme voraus, dass eine Verteilungsinfrastruktur bis zu einem gewissen Grad von den Unterschieden zugrunde liegender Platt-

formen abstrahiert. Der Nutzen dieser Forderung nimmt mit einer steigenden Zahl unterschiedlicher Geräte zu, wodurch die Abstraktion von plattformabhängigen Mechanismen generell als zukunftsorientierte Investitionssicherung betrachtet werden kann.

2.1.2 Minimalität

Um Zugriff auf möglichst viele Ressourcen unterschiedlichster Geräte mit derselben Infrastruktur zu erschließen, darf diese keine übermäßigen Anforderungen an Hard- oder Software stellen. Sie sollte mit minimalen Ressourcen auskommen. Hoher Verbrauch von Rechenleistung, Speicher oder hohe Anforderungen an Netzwerkbandbreite würden zwangsläufig ganze Klassen wichtiger Zielgeräte ausschließen.

So ist es unter dem Gesichtspunkt der Wirtschaftlichkeit ausgesprochen unwahrscheinlich, dass ein einfacher Temperatursensor, GPS-Empfänger oder Lichtschalter in naher Zukunft über große Mengen an Speicher verfügen wird. Jede Verteilungsinfrastruktur, die entsprechende Anforderungen an Zielsysteme stellt, schließt solche Geräte möglicherweise über Jahre hinweg kategorisch aus.

Die Anforderung, verschiedenste Geräte zu unterstützen, kann zum einen durch minimalistische Infrastrukturen erfüllt werden. Alternativ lässt sie sich auch mittels flexibel konfigurierbarer Infrastrukturen erfüllen. Über geräteabhängige Konfigurationen wird erreicht, dass kleine Rechner nur über einen erforderlichen Ausschnitt der Funktionalität des Gesamtsystems verfügen. Damit werden Anforderungen an Hard- und Software praktisch konfigurationsabhängig.

Extrem flexible Konfigurierbarkeit führt jedoch zwangsläufig zu höherer Komplexität. Damit steht sie zu einem gewissen Grad in Widerspruch mit der Forderung nach schmalen, verständlichen Schnittstellen. Bei der Konzeption muss demnach bewusst zwischen Konfigurierbarkeit und Verständlichkeit abgewogen werden.

2.1.3 Interoperabilität

Während Verständlichkeit und Minimalität auf die Unterstützung einzelner Geräte abzielen, muss darüber hinaus sichergestellt sein, dass Geräte verschiedenen Typs unabhängig von der verwendeten Software interoperabel sind. Durch die Heterogenität ubiquitärer Rechnersysteme kann nicht generell davon ausgegangen werden, dass jedes Gerät über dieselben Kommunikationsmechanismen verfügt. Während ressourcenarme Geräte wie Kaffeemaschinen oder Mikrowellen möglicherweise nur ein Kommunikationsproto-

koll unterstützen werden, implementieren Laptops oder Personal Digital Assistants bereits heute mehrere Protokolle.

Verteilungsinfrastrukturen für ubiquitäre Rechnersysteme müssen daher über möglichst flexible Kommunikationsmechanismen verfügen. Auch an dieser Stelle ist es wünschenswert, einfache Anwendungen nicht mit der Wahl eines Kommunikationsmechanismus zu belasten. Komplexeren kann optional die Beeinflussung dieser Wahl gestattet sein.

Gängige CORBA Implementierungen [Omg98] und neue Middleware wie beispielsweise BASE [Becker03a], liefern bereits heute flexible Kommunikationsmechanismen. CORBA basierte Systeme unterstützen in der Regel mehrere Protokolle, die zur Laufzeit ausgewählt werden. BASE geht einen Schritt weiter, indem es darüber hinaus Protokoll und Aufruf entkoppelt.

2.2 Mobilität

Neben der heterogenen Zusammensetzung ubiquitärer Rechnersysteme bringt die Mobilität von Geräten eine Reihe spezieller Anforderungen mit sich. Heutige Verteilungsinfrastrukturen setzen zumeist statische Netzwerke voraus. Ressourcen registrieren sich zu Beginn und sind während der gesamten Laufzeit verfügbar. Der Ausfall einzelner Ressourcen wird als ungewöhnlicher Fehlerzustand eingestuft und dementsprechend behandelt.

In ubiquitären Rechnersystemen ist die Verfügbarkeit von Ressourcen zu weiten Teilen dynamisch. Nutzer bewegen sich zusammen mit einigen ihrer Geräte. Aus Sicht dieser Geräte befindet sich ihr Umfeld in ständigem Fluss. Neue Ressourcen werden verfügbar und genutzte Ressourcen sind nicht länger verfügbar.

Aus diesem Grund muss sich die Infrastruktur dynamisch an die Umgebung anpassen. Um unter anwendungsspezifischen Gesichtspunkten optimale Funktion ermöglichen zu können, darf schwankende Ressourcenverfügbarkeit nicht als seltener Fehlerfall betrachtet werden. Damit ergeben sich die im Folgenden beschriebenen Anforderungen.

2.2.1 Integration

Damit sich ein System an schwankende Ressourcenbestände anpassen kann, muss es über einen Mechanismus zur Bestimmung verfügbarer Ressourcen verfügen. Ohne die Möglichkeit, die Menge aktuell verfügbarer Ressourcen zu bestimmen, müssen alle Entscheidungen hinsichtlich der Ressourcenverteilung reaktiv getroffen werden.

Durch spontane Bildung von Netzen wird die Bestimmung des Ressourcenbestandes erschwert. Bevor Ressourcen lokalisiert werden können, muss die

Menge erreichbarer Geräte bestimmt werden. Die Mechanismen zum Auffinden von Geräten sind dabei eventuell von den zugrunde liegenden Netzwerkprotokollen abhängig.

Die hohe Dynamik einiger Netzwerkteile macht es praktisch unmöglich, eine zentrale Instanz für die Verwaltung von Ressourcen einzurichten. Da die ständige Verbindung aller Geräte mit einem zentralen Gerät nicht garantiert werden kann, muss eine effiziente Ressourcenverwaltung zwangsläufig dezentral organisiert sein.

2.2.2 Adaption

Selbstverständlich ist es nicht ausreichend, wenn eine Verteilungsinfrastruktur lediglich feststellt, dass sich Ressourcenbestände geändert haben. Flexible Adaptionsmechanismen sind eine weiter Anforderung, um Anwendungen vor der Komplexität ubiquitärer Rechnersysteme zu schützen.

Wie Grimm et. al [Grimm02a] feststellen, ist es nicht immer vorteilhaft, wenn Anwendungen von allen Unterschieden verteilter Programmierung geschützt werden. Häufig führt die vollständige Isolierung von Anwendungen zu Hindernissen bei der Bestimmung oder Durchsetzung nichtfunktionaler Anforderungen.

Als Beispiel werden von Grimm et al. *Remote Procedure Call* Pakete angeführt, bei denen Anwendungen keine Möglichkeit haben, zwischen lokalen und entfernten Aufrufen zu unterscheiden. Anwendungen, die entfernte Ressourcen auf Basis dieser Pakete nutzen, verspielen die Möglichkeit auf spezielle Fehlerfälle entfernter Aufrufe gesondert zu reagieren. Für viele Anwendungen mag dies keine große Einschränkung darstellen. Kritische Anwendungen werden damit jedoch kategorisch ausgeschlossen.

Daher sollte klar sein, dass eine vollständig automatisierte und damit unsichtbare Adaption nicht immer wünschenswert ist. Flexible Infrastrukturen müssen es Anwendungen ermöglichen, die Parameter der Adaption selbst zu spezifizieren. Darüber hinaus könnte es für manche Anwendungen wünschenswert sein, wenn sie die Adaption in Eigenregie durchführen könnten. Daher sollte eine optimale Verteilungsinfrastruktur sowohl automatisierte als auch manuelle Adaption ermöglichen.

2.2.3 Administration

Die Administration dynamischer Systeme ist komplex. Dies gilt in besonderer Weise für ubiquitäre Rechnersysteme. Durch die spontane Vernetzung von Geräten unterschiedlicher Administrationsdomänen können Verfügbarkeit oder Eigenschaften bestimmter Anwendungen nicht zugesichert werden.

Die zugrunde liegende Verteilungsinfrastruktur muss daher sicherstellen, dass Probleme, die beispielsweise von Versionskonflikten herrühren, abgefangen werden. Da Fehler aufgrund unterschiedlicher Versionen in der Regel schwer nachvollziehbar sind, ist eine klare Fehlersemantik die absolute Minimalforderung.

Darüber hinaus sind die automatische Verteilung von Anwendungen sowie ihre Migration zur Vereinfachung der Administration in einigen Fällen durchaus wünschenswert. Vor allem unter dem Aspekt der Systemsicherheit muss dieser Wunsch, wenn überhaupt, äußerst vorsichtig realisiert werden.

2.3 Kontext

Während die Unterstützung heterogener, mobiler Systeme offensichtlich und unmittelbar aus der Definition ubiquitärer Rechnersysteme hervorgeht, ist die Forderung nach durchgängiger Unterstützung kontextueller Informationen subtiler.

Ubiquitous Computing hat definitionsgemäß den Anspruch, die kognitive Belastung bei der Bedienung von Rechnersystemen drastisch zu reduzieren. Ohne diese Reduktion ist die Integration von Rechnern in Alltagsgegenstände sinnlos und führt in letzter Konsequenz zu einer überladenen und damit unzugänglichen Umgebung.

Anwendungen für ubiquitäre Rechnersysteme müssen daher in höherem Maße auf kontextuelle Informationen zurückgreifen [Hendricksen01]. Anstatt Nutzer durch ständiges Nachfragen zu belästigen, muss kontextuelle Information definiert oder gesammelt, verbreitet und verarbeitet werden.

2.3.1 Spezifikation

Um kontextuelle Informationen unterstützen zu können, muss die Verteilungsinfrastruktur einen Rahmen definieren. Da der Kontext einer Anwendung in höchstem Maße anwendungsspezifisch ist [Abowd02], muss dieser Rahmen genügend Flexibilität bieten.

Betrachtet man die hohe Vielfalt von Kontextinformationen, wird deutlich, dass diese Anforderung nicht einfach zu realisieren ist. Kontextinformation wie beispielsweise Positionsangaben unterscheiden sich strukturell von Kontextinformationen wie geforderter Dienstgüte.

Darüber hinaus wirken sie an unterschiedlichen Stellen. In manchen Fällen, wie zum Beispiel bei der geforderten Dienstgüte, müssen sie sogar Einfluss auf die Verteilungsinfrastruktur selbst nehmen.

2.3.2 Verbreitung

Um adäquate Unterstützung von Anwendungen in ubiquitären Rechnersystemen zu gewährleisten, muss neben der Definition kontextueller Informationen auch ihre Verbreitung ermöglicht werden.

Die Verbreitung und der Austausch kontextueller Informationen werden wesentlich durch die Art der Information beeinflusst. Während Kontextinformation über Dienstgüte in der Regel bereits zu Beginn einer Kommunikation ausgetauscht werden müssen, könnten andere Arten wie Positionsinformationen auch asynchron ausgetauscht werden.

Eine flexible Verteilungsinfrastruktur muss geeignete Abstraktionen einführen, um den Austausch kontextueller Informationen zu vereinheitlichen und damit zu vereinfachen. Ohne eine Vereinheitlichung der Verbreitungsmechanismen kontextueller Informationen wird Interoperabilität entscheidend erschwert.

2.3.3 Verarbeitung

Gerade hinsichtlich kontextueller Information, die auf zugrunde liegende Verteilungsinfrastruktur wirkt, ist eine automatisierte Verarbeitung notwendig. Dies ist vor allem darauf zurückzuführen, dass kontextuelle Informationen die Ausprägungen ansonsten variabler Parameter festlegen.

Im Bezug auf Dienstgütemanagement muss beispielsweise die automatische Aushandlung von Parametern möglich sein. Da viele Qualitätsgarantien selbst auf unteren Schichten der Infrastruktur wirken, ist es nicht ausreichend, wenn kontextuelle Information ausschließlich auf Anwendungsebene abgehandelt wird. Vielmehr muss jede Schicht selbst dafür Sorge tragen, dass ausgehandelte Garantien durchgesetzt werden.

3 Existierende Ansätze

Da es sich bei den Anforderungen an Verteilungsinfrastrukturen für ubiquitäre Rechnersysteme um eine Obermenge der Anforderungen an klassische Verteilungsinfrastrukturen handelt, stellt sich natürlich die Frage, inwiefern diese bereits durch bestehende Systeme erfüllt werden können.

Darüber hinaus gibt es bereits heute eine Reihe von Systemen, die sich speziell den Eigenschaften ressourcenarmer und ubiquitärer Rechnersysteme widmen.

Im Folgenden werden exemplarisch einzelne Vertreter der drei Klassen herausgegriffen. Dabei werden die jeweiligen Konzepte grob skizziert und die gebotene Funktionalität mit den Anforderungen verglichen.

3.1 Klassische Verteilungsinfrastrukturen

Unter dem Punkt klassische Verteilungsinfrastrukturen werden Systeme gruppiert, die nicht unmittelbar für ressourcenbeschränkte oder ubiquitäre Rechnersysteme konzipiert wurden. Dies schließt nicht aus, dass es für einzelne Systeme Anpassungen oder Erweiterungen gibt, die sich speziell den Eigenschaften solcher Rechnersysteme annehmen.

Die Beispiele sind bewusst unvollständig gehalten. Systeme mit niedrigem Abstraktionsgrad, wie beispielsweise die direkte Programmierung auf der Ebene von *Sockets* oder einfachen *Remote Procedure Call* Mechanismen, wurden ausgeschlossen. Grund dafür ist vor allem die unzulängliche Eignung dieser Abstraktionen für die Entwicklung umfangreicher Systeme in heterogenen Umgebungen.

3.1.1 Objekt- bzw. Dienstorientiert

Objektorientierte Verteilungsinfrastrukturen, wie beispielsweise CORBA [Omg98] oder RMI [Sun01a] vereinfachen die Nutzung entfernter Objekte (häufig werden diese Objekte auch als Dienste bezeichnet).

Dazu werden, ausgehend von einer Schnittstellenbeschreibung der entfernten Objekte, Stellvertreterobjekte generiert. Diese Stellvertreter ermöglichen es, dass Anwendungen entfernte Objekte weitestgehend wie ihre lokalen Pendants nutzen können. Die Stellvertreterobjekte übernehmen in der Regel die gesamten Aufgaben, die für die verteilte Kommunikation notwendig sind.

Damit Anwendungen mit dem passenden Stellvertreter versorgt werden, verfügen die Infrastrukturen über verschiedene Standarddienste. Häufig ermöglichen zentral zugängliche Namens- und Verzeichnisdienste den Zugriff sowohl nach dem Yellow- als auch nach dem Whitepage-Paradigma. Aufgrund

dieser zentralen Dienste eignen sich konventionelle objektorientierte Verteilungsinfrastrukturen in der Regel nur für relativ statische Rechnernetze.

Die Zustellung von Aufrufen an Stellvertreterobjekte wird in der Regel von einer Instanz, die auf jedem Gerät ausgeführt wird, übernommen. Diese Instanz wird häufig *Object Request Broker* oder auch *Object Bus* genannt und ist für Anwendungen zumeist unsichtbar.

Neben diesen grundlegenden und notwendigen Eigenschaften verfügen moderne Infrastrukturen über eine ganze Reihe von weiteren zusätzlichen Fähigkeiten. Beispielsweise können aktuelle Versionen von RMI Stellvertreterobjekte zur Laufzeit generieren. Standardkompatible CORBA Implementierungen liefern eine große Zahl von Diensten, die unterschiedlichste Aufgaben bewältigen können, wie beispielsweise die Migration von Objekten.

3.1.2 Komponentenbasiert

Komponentenbasierte Verteilungsinfrastrukturen, wie beispielsweise CORBA Components [Omg02], Enterprise JavaBeans [Sun02b] oder .NET [Chappell02], basieren nach eigener Aussage auf dem Konzept der Softwarekomponenten. Da der Begriff Softwarekomponente in der Vergangenheit relativ vielseitig verwendet und häufig unterschiedlich definiert wurde [Szyperski98], ist die Abgrenzung objektorientierter und komponentenbasierter Verteilungsinfrastrukturen bisweilen lediglich eine Frage der Definition.

Während objektorientierte Verteilungsinfrastrukturen Anwendungen auf der Ebene von Objekten betrachten, gehen komponentenbasierte Verteilungsinfrastrukturen von Anwendungen aus, die aus Softwarekomponenten zusammengesetzt sind. Diese Komponenten sind in der Regel aus mehreren lokalen, interagierenden Objekten zusammengesetzt. Ihre Nutzung folgt in der Regel dem Vorbild objektorientierter Infrastrukturen. Teilweise basieren die komponentenbasierten Verteilungsinfrastrukturen auf objektorientierten (beispielsweise basiert Enterprise JavaBeans auf RMI).

Ein wichtiges Merkmal komponentenorientierter Infrastrukturen ist der zentrale Verwaltungsmechanismus, der in der Regel *Container* genannt wird. Der Container bildet die Laufzeitumgebung der Softwarekomponenten und sorgt dafür, dass Abhängigkeiten zwischen Softwarekomponenten aufgelöst werden. Darüber hinaus übernimmt der Container in der Regel eine Vielzahl unterschiedlicher Querschnittsaufgaben. Diese reichen von der Verwaltung des Komponentenlebenszyklus über Persistenz- und Transaktionsverwaltung bis hin zur Zugriffsverwaltung.

3.2 Ressourcenoptimierte Verteilungsinfrastrukturen

Neben den klassischen Verteilungsinfrastrukturen, die in der Regel für statische Rechnersysteme mit praktisch unbegrenzten Ressourcen ausgelegt sind, gibt es eine Reihe von Infrastrukturen, die an spezielle Szenarien angepasst sind.

Aufgrund der Eigenschaften ubiquitärer Rechnersysteme sind vor allem Verteilungsinfrastrukturen interessant, die an die Besonderheiten mobiler Netze angepasst sind oder die für kleine, ressourcenarme Systeme optimiert wurden.

3.2.1 Jini

Jini [Sun01b] ist eine objektorientierte Verteilungsinfrastruktur, die sich auf spontane Vernetzung fokussiert. Wie bei RMI oder CORBA existieren Stellvertreter für entfernte Objekte. Die ausschließliche Eignung herkömmlicher Infrastrukturen für statische Rechnernetze wird im Wesentlichen durch zwei Mechanismen aufgebrochen.

Zunächst sorgt ein Discovery-Protokoll dafür, dass neue Rechnersysteme innerhalb des Netzes erkannt werden können. Damit sorgt es dafür, dass Netzwerke ohne manuelle Konfiguration spontan gebildet werden können.

Weiterhin enthält ein auf jedem Rechnersystem verfügbarer Verzeichnisdienst alle entfernt nutzbaren Objekte und deren Stellvertreter. Dieser spezielle Dienst ermöglicht neben der typbasierten Suche den Austausch von Stellvertretern.

Die Kombination von Discovery-Protokoll und föderativem Verzeichnisdienst ermöglichen die Lokalisierung entfernter Objekte in ad hoc Netzen. Der Austausch von Stellvertreterobjekten gestattet darüber hinaus die Nutzung neuer, zuvor unbekannter Dienste.

3.2.2 Puppeteer

Puppeteer [DeLara01] ist ein System, das die Entwicklung adaptiver Komponentensoftware unter Verwendung existierender Komponentesysteme ermöglicht. Ziel ist es, bestehende Komponentensoftware entlang ihrer Komponentengrenzen für mobile Umgebungen zu optimieren. Dabei sollen bestehende Komponenten ohne Änderungen genutzt werden.

Um diesen Anforderungen gerecht zu werden, wird Puppeteer zwischen die Softwarekomponenten der Anwendung geschoben. Anstatt direkt miteinander zu kommunizieren, wird jede Verbindung der Softwarekomponenten mit Hilfe von Proxies durch Puppeteer geleitet. Diese Indirektion erlaubt die Op-

timierung der Kommunikation unter anwendungsspezifischen Gesichtspunkten.

Die Architektur von Puppeteer setzt sich dabei aus *Kernel-*, *Driver-*, *Transcoder-* und *Policymodulen* zusammen.

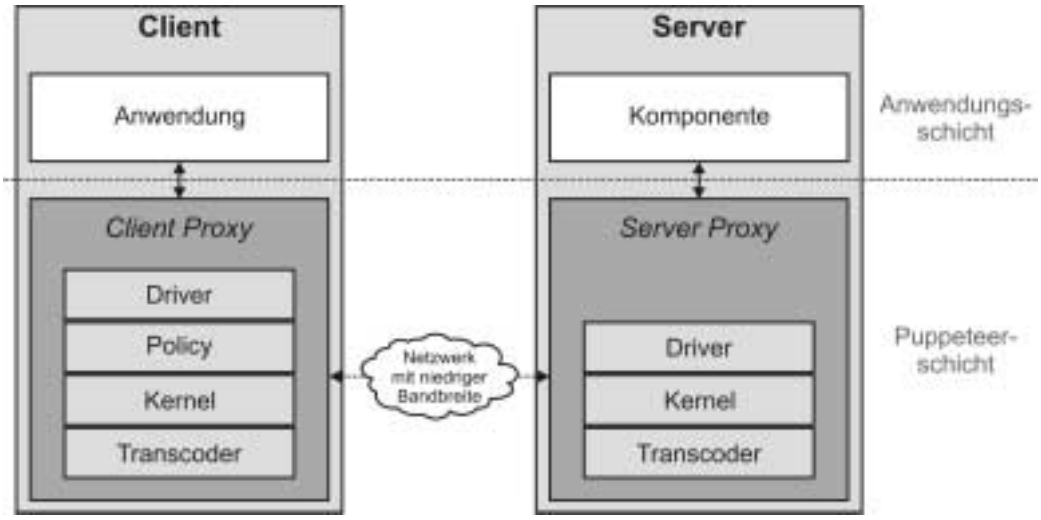

Abbildung 1 – Puppeteer Architektur

Kernelmodule implementieren das so genannte *Puppeteerprotokoll*, das rechnerübergreifende Kommunikation von Proxies ermöglicht. Drivermodule besitzen die Fähigkeit, zwischen den Daten eines bestimmten Komponententyps und einem komponentenunabhängigen Format zu mitteln und ermöglichen damit die Abstraktion von speziellen Komponententypen. Transcodermodule sorgen schließlich für die Reduktion der Datenmenge, die zwischen Proxies ausgetauscht werden muss. Diese Reduktion kann mit anwendungsunabhängigen Mittel wie beispielsweise Kompression oder durch anwendungsabhängige Transformationen erfolgen. Für die Wahl geeigneter Transcodermodule sorgen die Policies. Policymodule geben damit den Rahmen der Adaption vor.

Verwendet eine Anwendung eine mit Puppeteer angebundene Komponente, so wird beim Start der entsprechende Proxy initialisiert. Dabei werden gleichzeitig die benötigten Komponenten des Servers bestimmt. Spätere Aufrufe vom Klienten zum Server werden von Drivermodulen in ein komponentenunabhängiges Format konvertiert, das entlang einer Policy mit Hilfe von Transcodermodulen komprimiert wird. Auf der Seite des Servers werden ankommende Aufrufe decodiert und in das komponentenabhängige Ausgangsformat transformiert. Die Optimierung der Antwort des Servers erfolgt analog.

3.2.3 Universally Interoperable Core

Universally Interoperable Core [Román01a] ist eine reflektive Middleware, die vor allem auf den Einsatz in heterogenen Rechnersystemen abzielt. Primäres Ziel ist die Erstellung einer konfigurierbaren Middleware die sich möglichst einfach an die Gegebenheiten heterogener Systeme anpassen lässt. Basis der Arbeit ist die Middleware *DynamicTAO* [Kon00].

DynamicTAO ist eine CORBA Implementierung und zeichnet sich besonders durch dynamische Konfigurierbarkeit aus. Interne Eigenschaften, wie beispielsweise die Strategie des Schedulers oder die Vorgehensweise beim Verbindungsmanagement lassen sich zur Laufzeit ändern. Um die Konsistenz des gesamten Systems zu wahren, verfügt DynamicTAO über ein Netzwerk so genannter *Configurators*. Sie bilden die Basis des Reflektionsmechanismus, mit dessen Hilfe jederzeit festgestellt werden kann, welche Teile des Systems voneinander abhängig sind. Auf dieser Grundlage können ladbare Bibliotheken dynamisch hinzugefügt oder ersetzt werden, um darüber Eigenschaften der Middleware zu ändern.

Universally Interoperable Core definiert ein Gerüst für eine modulare Zerlegung von DynamicTAO. Damit ist es möglich, den Umfang von DynamicTAO an unterschiedlichste Einsatzgebiete anzupassen. Minimale Konfigurationen für Personal Digital Assistants enthalten beispielsweise nur klientenseitige Schnittstellen zur Interaktion mit einer Middleware, wohingegen Konfigurationen für Laptops gleichzeitig mehrere Middleware Plattformen klienten- und serverseitig unterstützen. Zur Modularisierung von DynamicTAO werden die Abstraktionen *Personality* und *Configuration* verwendet.

Personality bezeichnet eine konkrete Zusammensetzung von Modulen, die die Interaktion mit anderen Systemen ermöglicht. Eine Personality kann zum Beispiel die klientenseitige Interaktion mit CORBA Systemen ermöglichen, während eine andere die serverseitige mit DCOM erlaubt.

Eine Configuration spezifiziert hingegen die Form einer Zusammensetzung. Sie kann entweder statisch, dynamisch oder hybrid sein. Die statische Configuration eines Systems fixiert die Wahl der Module. Während eine dynamische Configuration Module zur Laufzeit nachladen oder austauschen kann, werden bei einer statischen Configuration Module zum Zeitpunkt der Übersetzung unwiderruflich verbunden. Die hybride Configuration ist eine Mischung zwischen dynamischer und statischer. Die offensichtlich höhere Flexibilität einer dynamischen Configuration führt zu umfangreicheren Systemen. Daher ist die Entscheidung für eine statische oder dynamische Configuration eine Abwägung zwischen Größe und Flexibilität.

Universally Interoperable Core erlaubt darüber hinaus die gleichzeitige Nutzung mehrerer Personalities. Damit ermöglicht er dieselbe Flexibilität wie DynamicTAO.

3.3 Verteilungsinfrastrukturen für Ubiquitous Computing

Neben den Verteilungsinfrastrukturen, die für bestimmte Ressourcenverfügbarkeit optimiert sind, gibt es auch eine ganze Reihe Verteilungsinfrastrukturen, die speziell für die Gegebenheiten ubiquitärer Rechnersystem entwickelt wurden.

Aufgrund der Vielzahl unterschiedlicher Anstrengungen wird an dieser Stelle nur eine kleine Auswahl aktueller Systeme und Projekte vorgestellt. Trotz der willkürlichen Zusammenstellung geben die Beschreibungen einen recht guten Überblick über die derzeitige Systemlandschaft und die darin enthaltenen Konzepte.

3.3.1 one.world

one.world [Grimm00, Grimm01, Grimm02a, Grimm02b] ist ein System, das auf die Erweiterung von Programmierabstraktionen auf Systemebene zur Unterstützung ubiquitärer Rechnersysteme abzielt. Ausgangspunkt ist die Idee, Besonderheiten der Anwendungsentwicklung in verteilten bzw. ubiquitären Systemen nicht vor Programmierern zu verbergen, sondern sie bei dieser komplexen Aufgabe durchgängig zu unterstützen. Um dies zu ermöglichen, müssen Applikationen in one.world strikt zwischen Daten und Anwendung trennen.

Das zentrale Konzept sind so genannte *Environments*, die den Zustand und die Daten von Anwendungen gruppieren. Environments können zusätzlich auch hierarchisch andere Environments enthalten. Ein *Checkpointing* Mechanismus, der es erlaubt, den Zustand eines Baums von Environments in einem *Tuppelspace* zu speichern, sorgt dafür, dass Environments konsistent von einem Rechner zum anderen migrieren können.

Neben Environments bietet one.world automatisches *Discovery*, mit dem entfernte Systeme entdeckt werden können. Die Kommunikation zwischen Rechnern erfolgt auf Basis von *Remote Event Passing*, womit unter anderem Nachrichten an Rechner gesendet werden können, die vom Discoverymechanismus entdeckt wurden.

Durch das Zusammenspiel aller Mechanismen können Anwendungen das System, auf dem sie ausgeführt werden, beliebig verändern. Damit werden unter anderem Anwendungen ermöglicht, die dem Aufenthaltsort ihrer Nutzer „folgen“.

3.3.2 BASE

BASE [Becker03a, Becker03b, Gubbels02] ist eine Middleware, die speziell für die Anforderungen ubiquitärer Rechnersysteme konzipiert ist. Wie bei anderen objektorientierten Verteilungsinfrastrukturen werden Stellvertreterobjekte verwendet, um auf entfernte Objekte zuzugreifen.

Ausgangspunkt ist ein mikrokernbasierter Object Request Broker, der so genannte *InvocationBroker*. Er ist für die Zustellung von Aufrufen verantwortlich. Um diesen Dienst zu erfüllen, verwendet er zwei Registrierungen, die *ServiceRegistry* und die *DeviceRegistry*. Die ServiceRegistry ist für die Verwaltung lokaler Objekte verantwortlich wohingegen die DeviceRegistry die Verwaltung entfernter Geräte übernimmt.

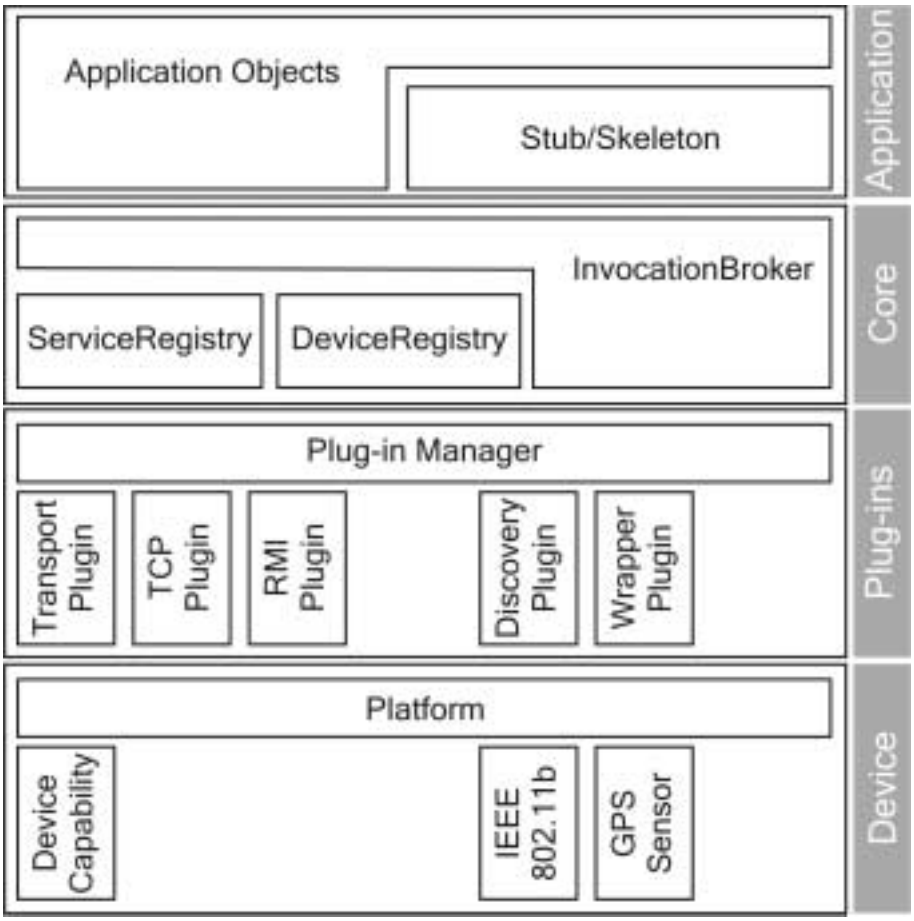

Abbildung 2 – BASE Architektur

Der wesentliche Unterschied zwischen klassischen, objektorientierten Verteilungsinfrastrukturen besteht darin, dass der eigentliche Versand von Aufrufen (Marshalling und Transport) durch *Plug-ins* von der Funktionalität des InvocationBrokers getrennt ist. Plug-ins, die eine Kommunikation mit einem bestimmten Gerät ermöglichen, werden für jeden Kommunikationsversuch

dynamisch ausgewählt. Die dynamische Auswahl gestattet es BASE, unterschiedliche Plug-ins zum Versenden und Empfangen desselben Aufrufs zu verwenden, was besonders für die unstetigen Netzwerkeigenschaften mobiler Geräte vorteilhaft ist.

Ähnlich wie Jini verfügt BASE über eine Schnittstelle zur automatischen Entdeckung von Geräten. Während Jini jedoch einen fest eingebauten Mechanismus verwendet, lässt BASE die Möglichkeit zu, Discovery-Mechanismen über Plug-ins nachzurüsten. Wird ein neues Gerät entdeckt, werden die Informationen über vorhandene Plug-ins ausgetauscht, wodurch die spätere Auswahl von Plug-ins zur Kommunikation ermöglicht wird.

Darüber hinaus können Plug-ins auch verwendet werden, um auf gerätespezifische Fähigkeiten zuzugreifen. Die Abstraktion von gerätespezifischen Eigenschaften über so genannte *Wrapper-Plugins* vereinfacht die Entwicklung plattformunabhängiger Anwendungen.

Neben dem flexiblen Plug-in-Mechanismus zeichnet sich BASE durch geringen Ressourcenverbrauch aus. Damit ist BASE auf ressourcenarmen Geräten ohne Einschränkungen lauffähig, was vor allem im Hinblick auf die Heterogenität ubiquitärer Rechnersysteme wichtig ist.

3.3.3 Gaia

Gaia [Román00] ist eine Architektur, die die Umsetzung so genannter *Active Spaces* ermöglicht. Active Spaces sind ein Modell für die Abbildung von Konzepten realer, physikalischer Räume auf Software. Sie sollen die Software eines Rechnersystems in die Lage versetzen, mit der Umgebung zu verschmelzen und abhängig vom Kontext zu agieren. In Analogie zu realen Räumen steht ein Active Space für einen physikalischen Raum, der bestimmte Entitäten enthält und gewissen Aufgaben dient.

Diese Form der Kontextinformationen wird Anwendungen zugänglich gemacht. Ziel sind dabei mobile Rechnersysteme, die aufgrund ihres umgebenden Active Spaces kontextabhängig agieren können. Gaia zielt damit auf eine spezielle Ausprägung ubiquitärer Rechnersysteme ab, in der ressourcenstarke, intelligente Umgebungen mit kleinen und mobilen Rechnern kooperieren.

Um die Kluft zwischen bestehenden Betriebssystemen und dem Konzept der Active Spaces nicht mit jeder Entwicklung aufs Neue überbrücken zu müssen, basieren Anwendungen auf Gaia OS. Bei Gaia OS handelt es sich um eine Infrastruktur, die gemeinsame Funktionalität aus Anwendungen extrahiert.

Gaia OS setzt sich aus dem *Unified Object Bus* [Román01b] und *Gaia Services* zusammen. Während der Unified Object Bus die Kommunikationsmechanismen vereinheitlicht und rechnerübergreifende Interaktionen ermöglicht, bie-

ten Gaia Services verschiedene Funktionen, die die Entwicklung für Active Spaces vereinfacht.

Der Unified Object Bus ist ein Kommunikationssystem zur Integration unterschiedlicher Komponentensysteme. Über eine zweistufige Hierarchie von so genannten *Component Managern* abstrahiert der Unified Object Bus von den Details beim Zugriff auf unterschiedliche Komponentensysteme.

Damit die Komponenten interoperabel sind, vereinheitlicht der Unified Object Bus ihren Lebenszyklus, Namensraum und ihre Verwaltung. Diese Vereinheitlichung ist Basis für die Entwicklung von Namens-, Trader- oder Verwaltungsdiensten, die unabhängig vom Komponentensystem einheitlichen Zugriff auf Komponenten ermöglichen.

Die eigentliche Kommunikation mit entfernten Komponenten erfolgt mittels bestehender Middleware. Da sich diese beliebig austauschen lässt, ist der Unified Object Bus sowohl für umfangreiche Rechner als auch für ressourcenarme geeignet.

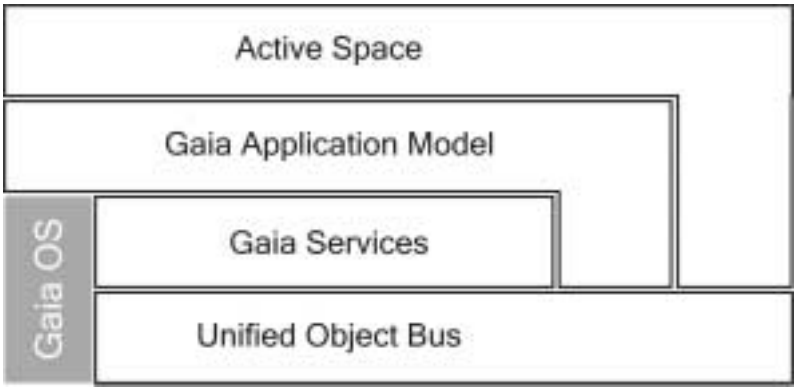

Abbildung 3 – Gaia Architektur

Gaia Service bauen auf dem Unified Object Bus auf und liefern eine Reihe von Diensten zur Steuerung und Koordination. Beispiele sind Dienste für Sicherheit oder Dienstgüte. Sie basieren auf *2K* [Román99], einem komponentenbasierten, verteilten Betriebssystem, das je nach Anforderungen dynamisch konfiguriert wird. Mittels dieser dynamischen Konfiguration wird sichergestellt, dass nur benötigte Gaia Services Ressourcen beanspruchen.

Gemeinsam mit dem Unified Object Bus unterstützen Gaia Services die Entwicklung von Anwendungen im Active Space Modell, denn Active Spaces werden in Gaia OS letztlich als Dienste modelliert. Als solche bieten sie die Möglichkeit, kontexutelle Informationen abzurufen, zu speichern und zu verändern.

Um die Strukturen weiter zu vereinheitlichen, enthält Gaia darüber hinaus einen Vorschlag für Anwendungsarchitekturen. Folgt man diesem Vorschlag, werden Anwendungen entlang des *Model-View-Controller Design Patterns* [Gamma95] zerlegt. Diese Zerlegung erleichtert die Entwicklung interaktiver Anwendungen, da zwischen Daten, Sicht und Interaktion getrennt werden kann.

3.3.4 Aura

Das Forschungsprojekt *Aura* [Cheng02, Garlan02] definiert ein architektonisches Rahmenwerk für Ubiquitous Computing. Ausgangspunkt sind explizite Beschreibungen von Anwendungsarchitekturen. Diese Anwendungsarchitekturen können zur Laufzeit entlang vordefinierter Regeln angepasst werden.

Die Anpassung der Architektur einer Anwendung kann zum einen durch Änderungen des Systemzustands ausgelöst werden. So kann beispielsweise eine kleiner werdende Netzwerkbandbreite dazu führen, dass entfernte Aufrufe komprimiert werden.

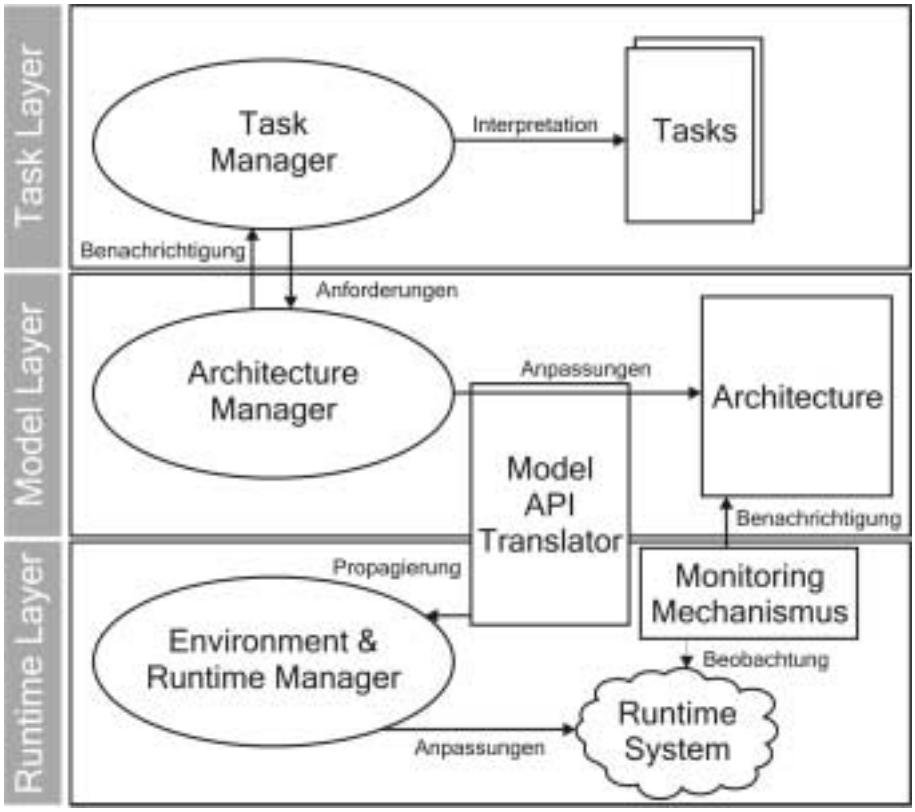

Abbildung 4 – Aura Architektur

Zum anderen kann die Architektur auch aufgrund eines sich ändernden *User Tasks* angepasst werden. User Tasks sind allgemeine Beschreibungen von

Aufgaben, die von Nutzern durchgeführt werden können. Beispiele hierfür wären das Abspielen eines Videos oder die Bearbeitung eines Texts.

Die daraus resultierenden adaptiven Anwendungsarchitekturen werden zur Laufzeit umgesetzt, um den gewünschten User Task zu unterstützen. Aura definiert hierfür ein dreischichtiges Modell, das aus *Task Layer*, *Model Layer* und *Runtime Layer* besteht.

Der Task Layer besteht aus dem *Task Manager*, der mit Hilfe der aktuell ausgeführten User Tasks die benötigten Anwendungen bestimmt und aufgabenspezifische Attribute wie beispielsweise Dienstgüteparameter vorgibt.

Der *Architecture Manager* im Model Layer nimmt die Anforderungen des Task Managers entgegen und passt damit die Architektur an. Der *Model API Translator* propagiert Änderungen der Architektur an den *Environment & Runtime Manager*.

Der Environment & Runtime Manager im Runtime Layer passt schließlich das *Runtime System* entsprechend an. Die Änderungen der Umbebung werden von einem *Monitoring Mechanismus* beobachtet, der diese an die darüberliegende Schicht weiterleitet.

3.3.5 Sparkle

Sparkle [Belaramani03] ist ein System zur Entwicklung von komponentenbasierten Anwendungen für ubiquitäre Rechnersysteme. Der Fokus von Sparkle liegt auf der automatischen Verteilung einer bestimmten Art von Softwarekomponenten, den so genannten *Facets*, in heterogenen, mobilen Systemen.

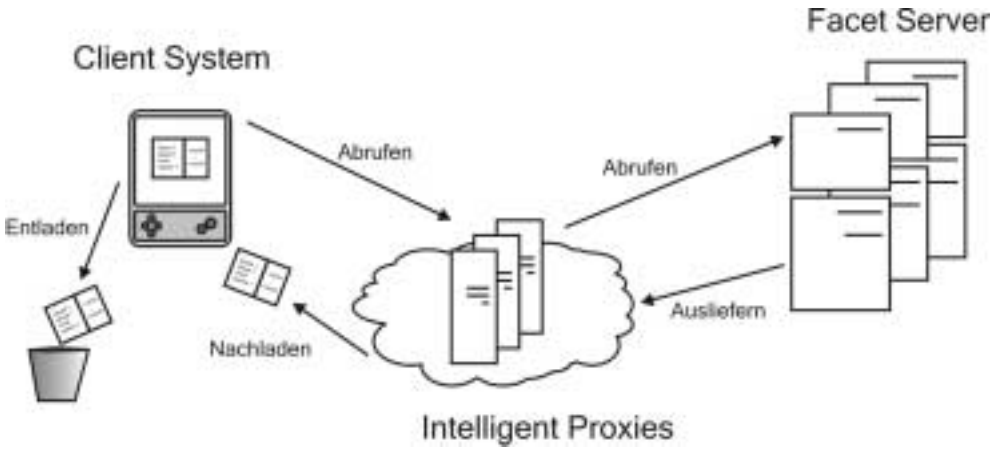

Abbildung 5 – Sparkle Architektur

Anwendungen in Sparkle setzten sich aus einer potentiell großen Zahl von Facets zusammen, die in einem *Container* ausgeführt werden. Sparkle geht

dabei von der Annahme aus, dass aus Effizienzgründen nicht alle Facets auf jedem System vorgehalten werden können. Deshalb wird es in der Regel notwendig sein, neue Facets nachzuladen und bereits ausgeführte zu entladen.

Facets kapseln genau eine Funktionalität und sind vollkommen zustandslos, wodurch gewährleistet wird, dass Aufrufsequenzen unabhängig voneinander sind. Diese Unabhängigkeit ermöglicht es, Facets zu jedem Zeitpunkt der Ausführung beliebig auszutauschen oder zu entladen. Um die automatische Verteilung zu ermöglichen, besitzt jede Facet neben dem *Code Segment* einen *Shadow*, eine XML-basierte Beschreibung der Funktionalität.

Wird eine Anwendung auf einem System ausgeführt, kann dieses benötigte Facets über so genannte *Intelligent Proxies* bei einem *Facetserver* abrufen. Intelligent Proxies optimieren die Selektion des Facetservers und sorgen mit Hilfe der Shadows dafür, dass die korrekten Facets abgerufen werden. Neue Facets werden zur Laufzeit mit Hilfe des *Lightweight Mobile Code Systems* in den Container und damit in die Anwendung integriert.

3.4 Vergleich

Im Hinblick auf die Anforderungen an Verteilungsinfrastrukturen für ubiquitäre Rechnersysteme ist klar, dass vor allem herkömmliche objektorientierte oder komponentenbasierte Verteilungsinfrastrukturen besonders schlecht angepasst sind.

Betrachtet man CORBA Implementierungen, gibt es zwar durchaus Systeme, die auf ressourcenarmen Geräten lauffähig sind[1], aber die durchgängige Unterstützung kontextueller Informationen und die Eignung für ad hoc Netze ist in der Regel nicht vorhanden oder letztlich unzureichend. Notwendige Dienste für Discovery oder automatischer Wechsel von Kommunikationskanälen sind in der Regel nicht vorhanden. Stattdessen sind Angaben über entfernte Rechner statisch und ein Ausfall dieser Rechner wird als seltener Systemfehler behandelt.

Ähnliches gilt für komponentenbasierte Verteilungsinfrastrukturen. Nicht erst seit der Einführung des *.NET Compact Frameworks* gibt es Komponentensysteme für ressourcenarme Geräte. Jedoch wurde auch bei diesem relativ neuen Komponentensystem nur wenig auf die speziellen Anforderungen von ad hoc Netzen oder auf die durchgängige Unterstützung kontextueller Informationen eingegangen.

[1] Ein Beispiel hierfür ist *minimumTAO*, http://www.cs.wustl.edu/~schmidt/TAO.html.

Ressourcenoptimierte Verteilungsinfrastrukturen bieten je nach Einsatzgebiet unterschiedliche Verbesserungen. Diese Verbesserungen zielen letztlich jedoch nur auf einen Aspekt oder wenige Aspekte ubiquitärer Rechnersysteme ab und vernachlässigen dabei andere.

Die Anpassungen von Jini ermöglichen den Einsatz in ad hoc Netzen ohne manuelle Netzkonfiguration. Auf Änderungen der Eigenschaften des Netzes wie Bandbreite oder Latenz geht Jini jedoch nicht ein. Damit eignet es sich vor allem für Umgebungen, in denen sich Geräte nur selten bewegen.

Puppeteer dagegen zielt gerade auf diese Art der Adaption ab und vernachlässigt dabei den Aspekt, der von Jini abgedeckt wird. Darüber hinaus ist fraglich, inwiefern sich Puppeteer für den Einsatz in einem heterogenen Rechnersystem eignet.

Diese Heterogenität ist dagegen die Domäne von UIC. Durch flexible Konfigurationsmöglichkeiten kann UIC mit minimalen Ressourcen zurechtkommen. Auf die Anforderungen mobiler Systeme oder die Unterstützung kontextueller Informationen wird dabei jedoch nicht eingegangen.

Auch bei den Verteilungsinfrastrukturen für Ubiquitous Computing decken nicht alle Systeme die gesamte Palette der Anforderungen ab. Bei einem Vergleich muss jedoch beachtet werden, dass es aufgrund des derzeitigen Status einiger Projekte und teilweise unvollständiger oder prototypischer Implementierungen schwierig ist, deren Eigenschaften abschließend zu beurteilen.

one.world liefert mit seinen relativ niedrigen Abstraktionen vor allem Unterstützung für die Mobilität und Heterogenität ubiquitärer Rechnersysteme. Durch die strikte Trennung von Daten und Programmen ist es möglich, Programme unterschiedlicher Rechnersysteme auf derselben Datenbasis auszuführen. Die Migrationsfähigkeit von Daten erlaubt es, in Verbindung mit der automatischen Erkennung von Rechnersystemen, mobile Nutzer durchgängig zu unterstützen. Die Unterstützung kontextueller Informationen muss bei one.world über den Eventmechanismus auf Anwendungsebene implementiert werden, da keine automatisierte Unterstützung vorgesehen ist.

Ähnlich wie one.world bietet auch BASE vorrangig Unterstützung für mobile und heterogene Systeme. Die Entwicklung von Anwendungen folgt jedoch dem bewährten Vorbild objektorientierter Verteilungsinfrastrukturen. Auch BASE liefert derzeit keine durchgängige Unterstützung kontextueller Informationen.

Das Modell der Active Spaces in Gaia bietet dagegen die Möglichkeit, Anwendungen kontextuelle Informationen zugänglich zu machen. Da Gaia darüber hinaus hinsichtlich der zugrunde liegenden Middleware äußerst flexibel ist, können auch die Anforderungen im Bezug auf Heterogenität und Mobili-

tät erreicht werden. Da das Konzept der ActiveSpaces jedoch hauptsächlich auf so genannte intelligente Umgebungen abzielt, bleibt die Frage offen, wie leistungsfähig Gaia in Umgebungen mit dynamischeren Eigenschaften ist.

Neben Gaia hat auch Aura das Potential, alle Anforderungen zu erfüllen. Durch Monitoring- und Adaptionsmechanismen verfügt Aura über weitreichende Konzepte zur Unterstützung kontextueller Informationen. Zusätzlich ermöglicht die Adaption von Anwendungsarchitekturen sowohl Anpassungen für heterogene Systeme, als auch Anpassungen an einen variierenden Grad an Mobilität. Im Bezug auf Aura bleibt lediglich die Frage offen, inwiefern die Kluft zwischen architektonischer Beschreibung und Umsetzung effizient überbrückt werden kann.

Sparkle geht wiederum eher auf Mobilität und Heterogenität ubiquitärer Rechnersysteme ein. Automatisches Laden und Entladen von Softwarekomponenten ermöglicht selbst Systemen mit relativ wenig Speicher die Ausführung größerer Anwendungen. Inwieweit dieser Ansatz im Hinblick auf andere Ressourcen wie beispielsweise Rechenleistung skaliert, bleibt abzuwarten.

4 Lösungsansatz

Der verbleibende Teil der Arbeit beschreibt die Entwicklung einer komponentenbasierten Verteilungsinfrastruktur für Anwendungen in ubiquitären Rechnersystemen. Wie das vorhergehende Kapitel deutlich macht, gibt es bereits eine Reihe von sowohl objekt- bzw. dienstorientierten als auch komponentenbasierten Verteilungsinfrastrukturen, die speziell für dieses Einsatzgebiet entwickelt wurden.

Daher stellt sich zunächst die Frage, warum eine komponentenbasierte Verteilungsinfrastruktur einer objektorientierten vorzuziehen ist. Nachdem diese Frage beantwortet ist, werden die Ziele dieser Arbeit vorgestellt. Um die zugrunde liegende Idee zu verdeutlichen, wird sie anhand einer Beispielanwendung vorgestellt, wonach diskutiert wird, was die Ziele dieser Arbeit von den Zielen bestehender Verteilungsinfrastrukturen für Ubiquitous Computing unterscheidet. Im letzten Teil dieses Kapitels werden Einschränkungen vorgestellt, die im Zuge der Entwicklung beschlossen wurden um den Umfang der Arbeit zu begrenzen.

4.1 Diskussion

Am Beispiel von Sparkle und Aura wird deutlich, dass es bereits existierende Verteilungsinfrastrukturen für ubiquitäre Rechnersysteme gibt, die sich direkt oder indirekt mit der verteilten Nutzung von Softwarekomponenten auseinandersetzen. Während das Systemmodell von Sparkle spezialisierte Softwarekomponenten direkt verwendet, werden im Modell von Aura adaptierbare architektonische Modelle auf Softwarekomponenten abgebildet.

Beide Ansätze nutzen dabei die definitionsgemäßen Vorteile von Softwarekomponenten gegenüber Objekten im Sinne der objektorientierten Programmierung aus. Wie bei der Definition des Begriffs Komponentensystem vorgestellt, zeichnen sich Komponenten vor allem durch vertraglich spezifizierte Schnittstellen und explizite kontextuelle Abhängigkeiten aus.

Zusammen bilden diese beiden Eigenschaften die Grundlage für die getrennte Entwicklung, Wartung, Installation und Nutzung. Damit wird erreicht, dass Anwendungen durch Komposition unabhängiger Softwarekomponenten entwickelt werden können.

Sparkle macht sich die Komposition von Anwendungen zu nutze, um zustandslose Komponenten bei Bedarf in ausgeführte Anwendungen einzubinden. Durch die Forderung nach zustandslosen Softwarekomponenten kann Sparkle diese jederzeit entladen bzw. austauschen, ohne die Korrektheit einer ausgeführten Anwendung zu gefährden. Für die Lokalisierung benötigter

Softwarekomponenten verwendet Sparkle die XML-basierte komponenteneigene Beschreibung.

Ähnlich wie Anfragen an intelligente Proxies in Sparkle, kann ein architektonisches Anwendungsmodell von Aura dazu verwendet werden, benötigte Softwarekomponenten auf Basis ihrer explizit beschriebenen Funktionalität auszuwählen. Durch die Komposition unabhängiger Softwarekomponenten kann eine Anwendung zusammengestellt werden, die alle Anforderungen korrekt erfüllt.

Der Vorteil von komponentenbasierten Systemen gegenüber objektorientierten liegt demnach zum einen in der automatisierten und flexiblen Komposition von Anwendungen aus kleinen, a priori unbekannten, aber wohldokumentierten Programmteilen. Zum anderen ermöglicht gerade die automatische Komposition von Anwendungen eine graduelle Automatisierung von wiederkehrenden Aufgaben.

Kommerzielle Komponentensysteme wie Enterprise JavaBeans oder .NET, respektive DCOM und COM+, bieten schon seit mehreren Jahren die Möglichkeit, eine Reihe von meist orthogonalen Aufgaben halb- oder vollautomatisiert vom Komponentensystem ausführen zu lassen. Dazu gehören unter anderem:

- *Verwaltung des Lebenszyklus*; Softwarekomponenten werden je nach Bedarf geladen, ausgeführt, gestoppt und entladen. Da das Entladen zustandsbehafteter Softwarekomponenten dazu führen kann, dass die Korrektheit eines Programms beeinträchtigt wird, ist die Steuerung des Lebenszyklus häufig mit einem Persistenzmechanismus gekoppelt (Stateful Enterprise JavaBeans).
- *Verwaltung von Persistenz*; Zustand von Softwarekomponenten kann automatisch in nichtflüchtigen Speicher geschrieben werden. Teilweise können solche Mechanismen auch mit Datenbanken oder ähnlichen, abstrakteren Speichermechanismen gekoppelt werden (Entity Enterprise JavaBeans).
- *Unterstützung von Transaktionen*; Interaktionen zwischen Softwarekomponenten folgen auf Wunsch den ACID Prinzipien (atomicity, consistency, isolated execution, durability) von Transaktionen [Date00].
- *Verwaltung von Zugriffsberechtigungen*; Systemadministratoren können den Zugriff auf Softwarekomponenten beschränken und beispielsweise mit den nutzerspezifischen Berechtigungen des Betriebssystems verknüpfen, ohne den Code bestehender Softwarekomponenten zu verändern (COM+).

- *Verwaltung von Ressourcen*; Systemadministratoren können die Ressourcen bestimmter Softwarekomponenten konfigurieren, ohne bestehende Softwarekomponenten zu verändern, um die Systemnutzung bestimmten Kriterien anzupassen. Beispiel hierfür ist die Konfiguration von Parametern zum Instancepooling oder Loadbalancing (COM+).
- *Mechanismen zur Fehlerbehandlung*; Systemadministratoren können die Reaktion des Systems auf Fehler konfigurieren. Softwarekomponenten können beispielsweise nach einem Fehler automatisch neu gestartet werden (COM+).

Die Automatisierung von Aufgaben erleichtert die Entwicklung neuer Softwarekomponenten und die Komposition von Anwendungen und bietet damit, bereits für verteilte Anwendungen in relativ statischen Rechnernetzen, erhebliche Vereinfachungen.

Darüber hinaus führt diese Form der Automatisierung auch zu einer Standardisierung von weiten Teilen der genutzten Plattform. Bietet eine Verteilungsinfrastruktur mächtige Mechanismen an, kann in der Regel von plattformabhängigen abstrahiert werden.

Alle bereits genannten Vorteile werden durch die Eigenschaften ubiquitärer Rechnersysteme um ein Vielfaches verstärkt. Dazu kommen noch einige neue Vorteile, die aus den speziellen Anforderungen ubiquitärer Rechnersysteme abgeleitet werden können.

Wie aus der Definition von Ubiquitous Computing ersichtlich ist, werden Anwendungen in ubiquitären Rechnersystemen mit einer Vielzahl unterschiedlicher Geräte kooperieren müssen, um ihre Aufgaben effizient erfüllen zu können. Die Kooperation zwingt Anwendungen dazu, dass sie Geräte nutzen, die sich in ihrer Umgebung befinden.

Softwarekomponenten bieten eine bewährte Aufteilung der Funktionalität von Anwendungen, die sich einfach und für Entwickler weitgehend unsichtbar, über Rechnergrenzen hinweg verteilen lässt. Darüber hinaus sorgt die Beschreibung von Softwarekomponenten dafür, dass auch die Funktionalität zuvor unbekannter Geräte nahtlos in bestehende Anwendungen integriert werden kann.

Die Heterogenität der verschiedenen Rechnersysteme und ihre Mobilität machen die explizite Beschreibung von angebotener Funktionalität eines Gerätes praktisch unumgänglich. Die ausschließlich diensttypbasierten Abstraktionen, wie sie gängige Middleware bietet, sind für die Anforderungen ubiquitärer Rechnersysteme unzulänglich.

Ein Beispiel hierfür ist die Unzulänglichkeit von RMI Systemen, zwischen unterschiedlichen Interfaceversionen zu unterscheiden. Der Mangel, gewisse

nicht-funktionale Parameter nicht ausdrücken zu können, mag innerhalb geschlossener Administrationsdomänen erträglich sein. Ubiquitäre Rechnersysteme umspannen jedoch per Definition mehrere Administrationsdomänen. Unvollständige Funktionsbeschreibungen können in einer solchen Umgebung zu unvorhersehbaren Ergebnissen führen. Damit wird eine unvollständige Beschreibung zum K.O. Kriterium.

Gerade bei der Beschreibung bieten Softwarekomponenten den entscheidenden Vorteil. Die Forderung nach der Spezifikation aller kontextueller Abhängigkeiten und vertraglich spezifizierten Schnittstellen führt dazu, dass sie entlang ihrer Parameter nach belieben kombiniert werden können, ohne die Korrektheit des resultierenden Programms zu gefährden.

Auch die Einführung von Mechanismen zur Automatisierung von häufig wiederkehrenden Aufgaben und die damit einhergehende Standardisierung der Plattform sind gerade für ubiquitäre Systeme besonders vorteilhaft. Durch die Standardisierung der Mechanismen können Komponenten einfacher auf unterschiedliche Plattformen portiert werden. Zusätzlich kann die automatische Abhandlung wiederkehrender Funktionen die Komplexität und den Umfang von Softwarekomponenten deutlich reduzieren.

Darüber hinaus bieten die wohldefinierten Verbindungspunkte und Abhängigkeiten zwischen Softwarekomponenten einen Ansatzpunkt, um die Unterstützung kontextueller Informationen durchgängig in die Verteilungsinfrastruktur zu integrieren. So kann beispielsweise von allen Programmteilen, die zu einer Softwarekomponente gehören, gefordert werden, dass sie sich korrekt im Sinne ihrer vertraglich spezifizierten Schnittstellen verhält, sobald ihre Abhängigkeiten erfüllt werden. Damit kann sich eine komponentenbasierte Verteilungsinfrastruktur auf die allgemeine Durchsetzung und Verteilung kontextueller Informationen außerhalb der Softwarekomponentengrenze beschränken.

Zusätzlich sind komponentenbasierte Verteilungsinfrastrukturen gut geeignet, um höhere Abstraktionen aufzubauen. Vor allem im Hinblick auf Anwendungsarchitekturen und deren Beschreibungen gibt es eine Reihe von Ansätzen, die die Vorteile von Softwarekomponenten ausnutzen, um den Entwicklungsprozess zu vereinfachen. Projekte wie Aura zeigen, dass solche Ansätze auch für Anwendungen in ubiquitären Rechnersystemen möglich und sinnvoll sind.

Neben dieser Vielzahl von Vorteilen komponentenbasierter Verteilungsinfrastrukturen soll natürlich nicht verschwiegen werden, dass die Nutzung von Softwarekomponenten einige Schwierigkeiten mit sich bringt.

Um eine komponentenbasierte Verteilungsinfrastruktur entwickeln zu können, müssen zunächst ein oder mehrere Komponentenframeworks definiert

werden. Da diese Komponentenframeworks gewissermaßen den Rahmen für die Aufteilung von Anwendungen vorgeben, ist es äußerst wichtig, dass die Abstraktionen der Komponentenframeworks Entwicklungen nicht einschränken. Damit wird die Gestaltung der Komponentenframeworks ein kritischer Erfolgsfaktor.

Zusätzlich sorgt die Einführung von Mechanismen zur Automatisierung wiederkehrender Aufgaben für eine drastische Steigerung der Komplexität auf Seiten der Verteilungsinfrastruktur. Die Verantwortung für die Umsetzung der Mechanismen verschiebt sich vom Entwickler der Anwendung hin zum Entwickler der Verteilungsinfrastruktur. Durch komplexe Mechanismen wird demnach zwar die Portierungsaufwand von Softwarekomponenten verringert, im Gegenzug wird jedoch der Aufwand für die Portierung der Verteilungsinfrastruktur größer.

Die Komplexität der Mechanismen ist gleichzeitig Anlass für das nächste Argument, das gegen einen komponentenbasierten Ansatz vorgebracht werden kann. Im Allgemeinen kann man davon ausgehen, dass der größere Funktionsumfang eine aufwendigere, d.h. umfangreichere Implementierung erfordert. Damit hat ein automatisierter, komponentenbasierter Ansatz tendenziell einen größeren Bedarf an Systemressourcen als ein weitgehend manueller und objektorientierter. Dieses Problem entschärft sich jedoch, falls die Mechanismen der Verteilungsinfrastruktur genutzt werden, denn würde die Verteilungsinfrastruktur die Mechanismen nicht anbieten, dann müssten die Softwarekomponenten selbst die zusätzliche Funktionalität liefern.

Durch die genannten Vor- und Nachteile von komponentenbasierten Verteilungsinfrastrukturen wird deutlich, dass die Konzeption einer Verteilungsinfrastruktur für Anwendungen in ubiquitären Rechnersystemen eine Gratwanderung ist, bei der Entscheidungen nicht unabhängig von Anwendungen getroffen werden können. Vor allem im Hinblick auf das Komponentenmodell und den Grad an Automatisierung und Standardisierung von Mechanismen gibt es eine Reihe von Abwägungen, deren Auswirkungen nicht ohne Annahmen über mögliche Anwendungen vorher bestimmt werden können.

4.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer komponentenbasierten Vereilungsinfrastruktur. Durch den komponentenbasierten Ansatz soll es möglich werden, dass Anwendungen aus interagierenden Komponenten zur Laufzeit zusammengesetzt werden können. Diese Komposition soll die Aufteilung von Funktionen über Rechnergrenzen hinweg ermöglichen.

Bei der Ausführung von Anwendungen sollen keinerlei Anforderungen an die Umgebung der Geräte gestellt werden, da diese a priori nicht bekannt sein

kann. Zentrale Dienste, wie sie in klassischen Verteilungsinfrastrukturen eingesetzt werden, sind für ubiquitäre Rechnersysteme inadäquat. Generell gilt, dass alle Anforderungen bezüglich Heterogenität und Mobilität ubiquitärer Rechnersysteme erfüllt werden müssen, wobei ermöglicht werden muss, dass Anwendungen mit kontextuellen Informationen versorgt werden können.

Im Bezug auf Heterogenität bedeutet das zunächst, dass möglichst von plattformabhängigen Mechanismen abstrahiert werden soll. Da diese Abstraktion das Potential hat, Portierungsaufwand und Performance negativ zu beeinflussen, muss ein Kompromiss zwischen beiden Extrema gefunden werden. Konkret führt das dazu, dass überall dort von plattformabhängigen Mechanismen abstrahiert werden soll, wo das ohne großen Einfluss auf Performance oder Portierbarkeit möglich ist.

Darüber hinaus fordert die heterogene Systemlandschaft die Beschränkung des Ressourcenbedarfs. Daraus folgt, dass bei der Automatisierung von wiederkehrenden Aufgaben darauf geachtet werden muss, dass sie nicht zu einem übermäßigen Bedarf an Ressourcen führt. Diese Arbeit berücksichtigt daher nur Mechanismen, die für eine komponentenbasierte Verteilungsinfrastruktur zwingend notwendig sind. Bei diesen wird dann darauf geachtet, dass nur grundlegende Funktionen direkt umgesetzt werden und weitergehende Funktionen nachgerüstet werden können. Schmale verständliche Schnittstellen sollen die Fälle abdecken, die häufig anzutreffen sind.

Weiterhin soll die komponentenbasierte Verteilungsinfrastruktur, vor allem im Bezug auf unterschiedliche Kommunikationsmechanismen, erweiterbar sein. Sowohl die Heterogenität als auch die Mobilität ubiquitärer Rechnersysteme machen flexible Erweiterbarkeit unterstützter Protokolle und Discoverymechanismen unumgänglich.

Da entfernte Ressourcen gerade in Umgebungen mit mobilen Geräten jederzeit ausfallen können, muss eine Verteilungsinfrastruktur für ubiquitäre Rechnersysteme Anwendungsentwickler bei der Verwaltung von Abhängigkeiten und der Behandlung von Fehlern unterstützen. Auch bei diesen Mechanismen muss darauf geachtet werden, dass sie die häufigsten Fälle abdecken und für Sonderfälle entsprechend erweitert werden können.

Die Mobilität der Geräte führt letztlich dazu, dass eine komponentenbasierte Verteilungsinfrastruktur für ubiquitäre Rechnersysteme keinerlei Annahmen über die Aufteilung von Administrationsdomänen treffen darf. Die Beschreibung der Softwarekomponenten muss gewährleisten, dass vollkommen unbekannte Komponenten korrekt agieren können. Damit muss die Beschreibung sowohl von funktionalen als auch von nicht-funktionalen Parametern möglich sein.

Der Bedarf an kontextuellen Informationen von Anwendungen für ubiquitäre Rechnersysteme erfordert die Definition von Mechanismen zur Verbreitung und Verarbeitung. In der zu entwickelnden, komponentenbasierten Verteilungsinfrastruktur werden daher Schnittstellen definiert werden, um den Umgang mit kontextuellen Informationen zu vereinheitlichen.

4.3 Beispiel

Die folgende Grafik soll die angestrebte Komposition am Beispiel einer Anwendung verdeutlichen, die den Nutzer akustisch über bevorstehende Termine benachrichtigt. Die Benachrichtigungsanwendung besteht aus einer Softwarekomponente, die auf einem Mobiltelefon ausgeführt wird, das der Nutzer bei sich trägt. Um ihren Dienst, die akustische Benachrichtigung bei bevorstehenden Terminen, korrekt anbieten zu können, benötigt die Benachrichtigungskomponente eine Audiokomponente, um Töne wiederzugeben, und eine Kalenderkomponente, um Termine abzurufen.

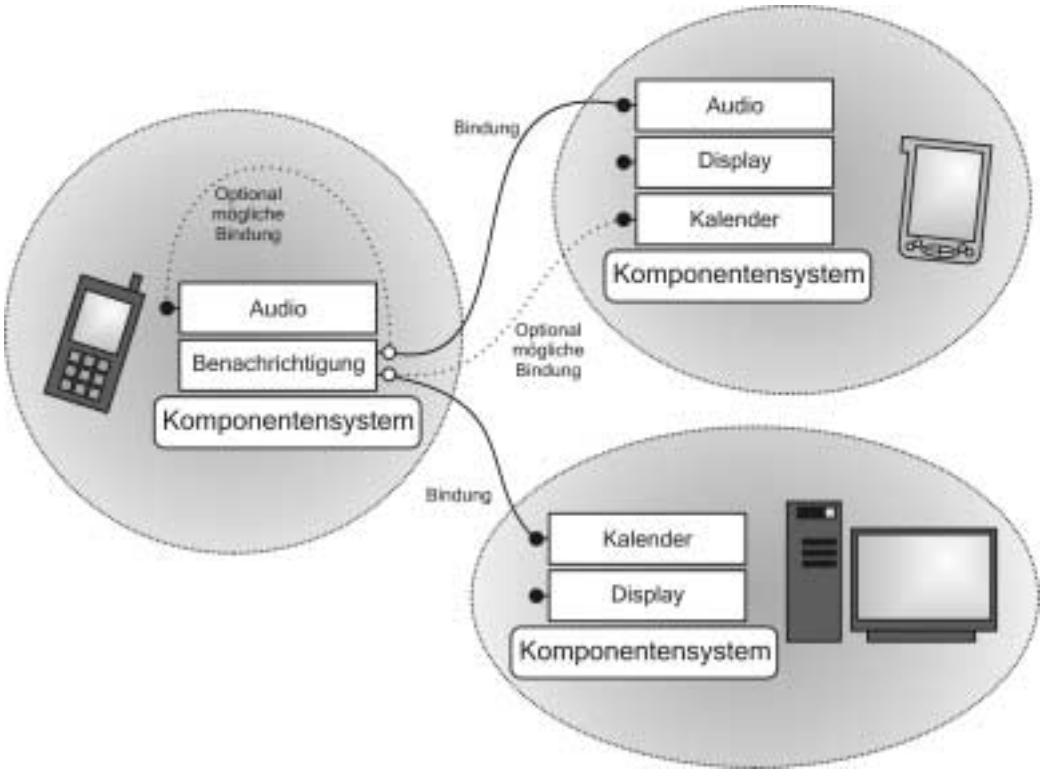

Abbildung 6 – Komposition von Anwendungen

In diesem Beispiel ist das Mobiltelefon zunächst mit einem PDA und einem PC verbunden. Auf dem PDA befinden sich einige Softwarekomponenten, die verschiedene Funktionalitäten des PDA kapseln. Die Audiokomponente erlaubt es, Töne wiederzugeben, die Displaykomponente kann das Display des

PDA ansteuern und die Kalenderkomponente bietet die Funktionalität, Termine abzurufen und zu verwalten. Der PC liefert ebenfalls eine Kalenderkomponente und eine Displaykomponente für den Bildschirm des PCs.

Beim Start der Benachrichtigungskomponente sorgt die zu entwickelnde, komponentenbasierte Verteilungsinfrastruktur dafür, dass die benötigten Softwarekomponenten bereitgestellt werden, sofern diese verfügbar sind. In obigem Beispiel hat die Infrastruktur mehrere Möglichkeiten, um die Abhängigkeiten aufzulösen. Da sowohl der PC als auch der PDA über eine Kalenderkomponente verfügen, kann wahlweise eine dieser Komponenten verwendet werden. Gleiches gilt für die Audiokomponente, die sowohl vom PDA als auch vom Mobiltelefon selbst angeboten wird.

In diesem Beispiel werden die Audiokomponente des PDA und die Kalenderkomponente des PC verwendet, womit die Anwendung aus der Benachrichtigungskomponente des Mobiltelefons, der Kalenderkomponente des PC und der Audiokomponente des PDA zusammengesetzt ist.

Die verschiedenen Softwarekomponenten werden durch die Verteilungsinfrastruktur mit kontextuellen Informationen versorgt. Die tatsächlichen Informationen können sowohl anwendungs- als auch geräteabhängig sein. Ihre Bereitstellung erfolgt entlang der Komponentengrenzen.

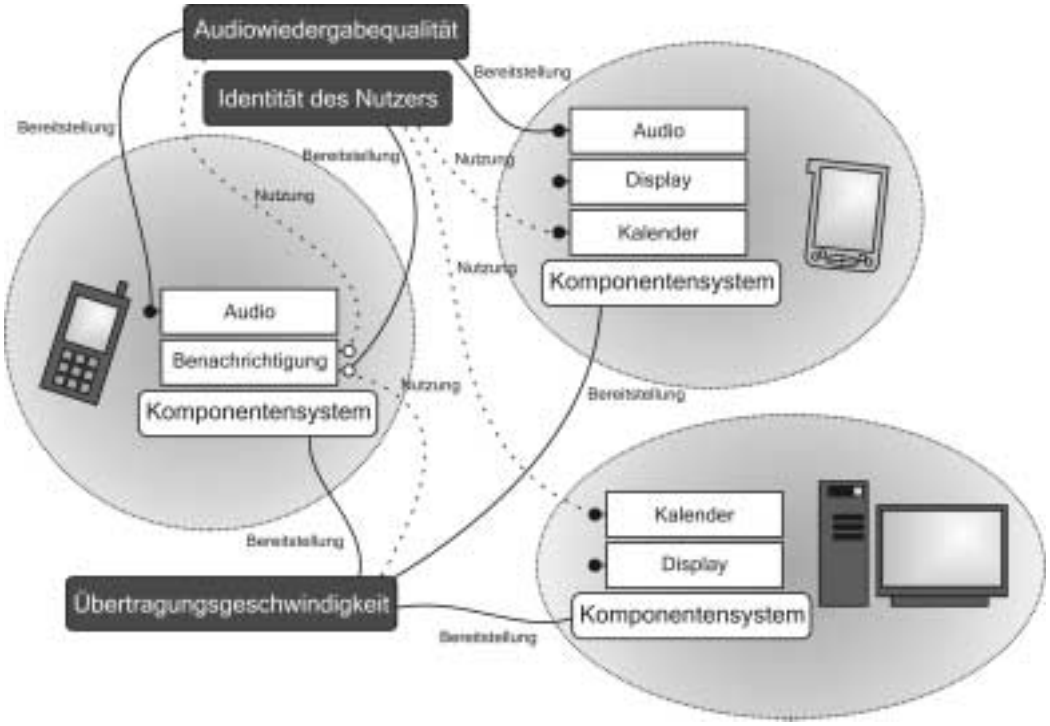

Abbildung 7 – Einbettung kontextueller Information

Für diese fiktive Beispielanwendung könnte die Benachrichtigungskomponente Informationen über die Identität des Nutzers bereitstellen, wodurch die Kalenderkomponente den richtigen Kalender abrufen könnte. Weiterhin könnte die Audiokomponente Informationen über die Qualität der Audioausgabe an die Benachrichtigungskomponente weiterleiten. In Verbindung mit Informationen über die Übertragungsgeschwindigkeit des Netzes könnte die Benachrichtigungskomponente die Qualität und damit die Größe der akustischen Nachrichten bestimmen.

Bricht nun beispielsweise die Verbindung zum PC ab, muss das Komponentensystem auf die neue Situation reagieren. Eine Möglichkeit hierfür wäre der automatische Wechsel zur Kalenderkomponente des PDA, wonach sich die Anwendung gemäß der folgenden Grafik zusammensetzen würde.

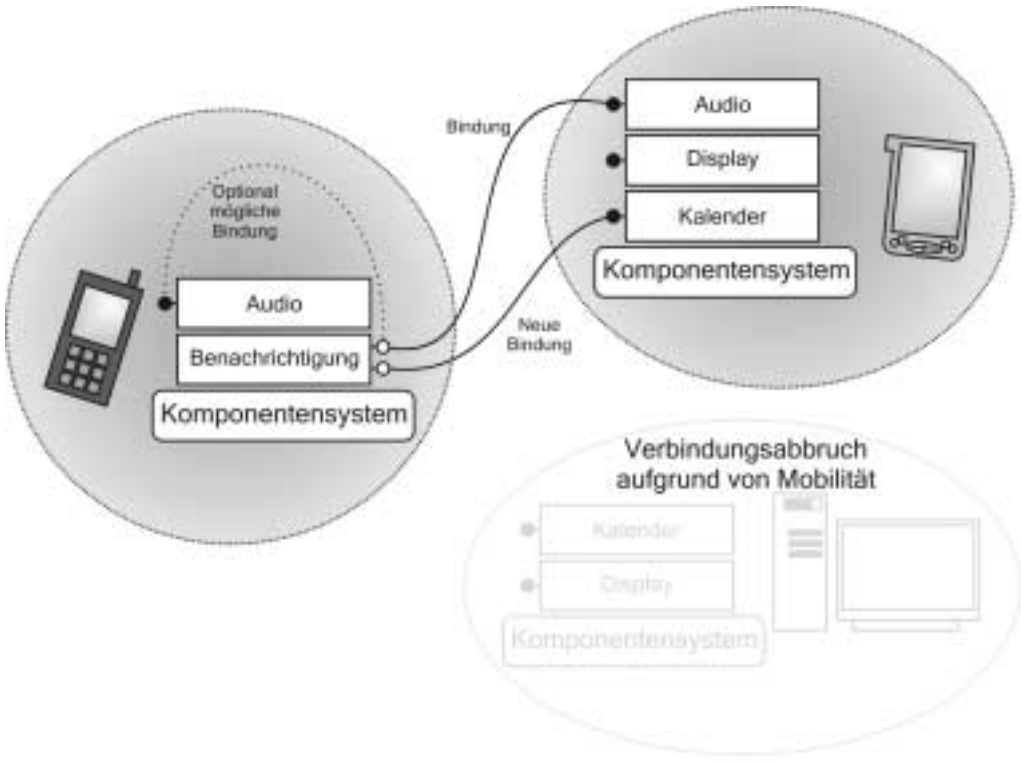

Abbildung 8 – Umgang mit Mobilität

Die Infrastruktur verfügt über die Möglichkeit, Anwendungen entlang bestimmter Parameter selbst zur Laufzeit neu zu gestalten. Hierin liegt ein wesentlicher Unterschied zu herkömmlichen Ansätzen. Im Gegensatz zu klassischen Verteilungsinfrastrukturen soll ein Verbindungsabbruch nicht als harter Fehler angesehen werden, der die Ausführung beendet. Die automatische Reparatur von Anwendungen entlang vordefinierter Parameter soll dafür

sorgen, dass Anwendungen auch in hochdynamischen Umgebungen lauffähig sind.

4.4 Abgrenzung

Bei der Vielfalt bestehender Verteilungsinfrastrukturen für ubiquitäre Rechnersysteme, die teilweise sogar komponentenbasierte Ansätze verfolgen, stellt sich die Frage, inwiefern sich die Ziele dieser Arbeit von den Zielen existierender Arbeiten unterscheiden.

Betrachtet man one.world, kann man zunächst feststellen, dass one.world auf Abstraktionen auf Systemebene abzielt, die nicht dem Paradigma der Softwarekomponenten folgen. one.world zeichnet sich vor allem durch die Trennung von Programmen und Daten und das Konzept der Environments aus. Die Möglichkeit der konsistenten Migration von Daten nutzt one.world, um die gewünschte Funktionalität immer in der Nähe des Nutzers zu bieten.

Im Gegensatz dazu versucht die komponentenbasierte Verteilungsinfrastruktur Sparkle durch mobilen Code, die gewünschte Funktionalität auf einem ressourcenarmen, mobilen System zu liefern, das sich immer in der Nähe des Nutzers befindet.

Die Verteilungsinfrastruktur, die in dieser Arbeit entwickelt wird, zielt auf die spontane Nutzung vorhandener Ressourcen ab. Während one.world Daten und Sparkle Code in die Nähe des Nutzers bringen, stellt diese Arbeit einen Ansatz vor, der es ermöglicht, die in der Umgebung des Nutzers befindlichen Ressourcen zu nutzen. In dieser Hinsicht ähnelt der Ansatz dieser Arbeit eher den Konzepten von Gaia und Aura.

Anders als bei Gaia wird jedoch nicht von einer festen Umgebung ausgegangen, die es ermöglicht, zentrale Elemente wie ActiveSpaces zu liefern. Damit beschränkt sich diese Arbeit nicht nur auf intelligente Umgebungen, die ressourcenarmen Geräten feste Mechanismen anbieten können. Dennoch muss auch die Verteilungsinfrastruktur, die in dieser Arbeit entwickelt wird, die Nutzung kontextueller Informationen unterstützen.

Im Hinblick auf die verbleibenden Ansätze wird deutlich, dass die Konzepte und damit auch die Ziele dieser Arbeit zwischen denen von BASE und Aura eingeordnet werden können.

Wie one.world bietet BASE vergleichsweise niedrige, dem Dienst- bzw. Objektparadigma folgende Abstraktionen. Die gängigen Elemente zur Unterstützung von spontaner Vernetzung, wie beispielsweise Discovery- und föderativer Namensdienst sorgen dafür, dass diese Konzepte auch in Umgebungen mit mobilen Geräten genutzt werden können. Zusätzlich sorgen Plug-ins für flexible Erweiterbarkeit und Plattformunabhängigkeit. Damit können die

bekannten und bewährten Konzepte objektorientierter Verteilungsinfrastrukturen auf ubiquitäre Rechnersysteme übertragen werden. Im Hinblick auf die Unterstützung kontextueller Information hat BASE, zumindest in seinem derzeitigen Stadium, einige Schwachpunkte. Mit Hilfe der für Softwarekomponenten notwendigen Beschreibungen zielt diese Arbeit auf eine durchgängigere Unterstützung kontextueller Informationen ab.

Die adaptiven Anwendungsarchitekturen von Aura sind deutlich abstrakter als die Softwarekomponenten, auf die diese Arbeit abzielt. Damit bietet Aura grundsätzlich höhere Flexibilität bei der Komposition von Anwendungen. Gleichzeitig wird jedoch ein effizienter Mechanismus erforderlich, der Anwendungsarchitekturen letztlich umsetzt. Bei der Entwicklung einer komponentenbasierte Verteilungsinfrastruktur wird ein solcher Mechanismus zunächst nicht benötigt. Das führt dazu, dass eine komponentenbasierte Verteilungsinfrastruktur im Allgemeinen mit weniger Ressourcen auskommen kann.

Diese Arbeit zielt jedoch letztlich auf die spätere Anbindung von Mechanismen ab, die in der Lage sind, mehr Annahmen über ihre Umgebung zu treffen und mit anwendungsspezifischem Wissen ausgestattet werden können. Damit muss die hier vorgestellte Verteilungsinfrastruktur, zumindest im Bezug auf die Auswahl Softwarekomponenten und die Zusammensetzung von Anwendungen, Schnittstellen bieten, die es erlauben, Systeme wie Aura anzubinden.

4.5 Einschränkungen

Die komponentenbasierte Verteilungsinfrastruktur, die in dieser Arbeit entwickelt wird, soll die Grundlage für spätere Arbeiten legen. Aus diesem Grund zielt die Arbeit vor allem auf die zentralen Bestandteile ab, die zwingend erforderlich sind, um die soeben vorgestellten Ziele zu erreichen.

Weiterführende Konzepte und Überlegungen, wie beispielsweise die Migration von Softwarekomponenten oder der Umgang mit zustandsbehafteten Softwarekomponenten, werden bewusst, soweit es sich mit den Zielen vereinbaren lässt, vernachlässigt, um nicht den Rahmen dieser Arbeit zu sprengen. Trotzdem werden im Verlauf die Ansatzpunkte für eventuelle Erweiterungen in diese oder ähnliche Richtungen an der jeweiligen Stelle vorgestellt.

Ähnliches gilt auch für den Bereich der Anwendungsarchitekturen. Obgleich bereits heute abzusehen ist, dass die Adaption von Architekturen mächtige Möglichkeiten zur Anpassung und Konfiguration von Anwendungen bietet, geht diese Arbeit nicht weiter als erforderlich auf das Thema ein. Dennoch wird im Folgenden darauf geachtet, dass die zu entwickelnde Verteilungsinfrastruktur die Anbindung von architekturbasierten Ansätzen ermöglicht.

Darüber hinaus beschäftigt sich diese Arbeit nicht mit der Definition oder gar einer Klassifizierung verschiedener Arten von kontextuellen Informationen, auch wenn dies im Hinblick auf das Ziel, die Verbreitung und die Verarbeitung von kontextuellen Informationen durchgängig zu unterstützten, zunächst nahe liegend scheint. Vielmehr wird in dieser Arbeit ein Rahmen vorgegeben, in den kontextuelle Informationen integriert werden können. Aufgrund der Tatsache, dass kontextuelle Informationen, wie beispielsweise Dienstgüte, bereits innerhalb der Verteilungsinfrastruktur wirken müssen, kann auch im Bezug auf die Verarbeitung kontextueller Informationen nur ein Rahmen für die spätere Weiterentwicklung definiert werden.

Auch wenn im Verlauf der Arbeit Dienstgüte häufiger als Beispiel für nichtfunktionale Parameter gebraucht wurde und wird, beschäftigt sich diese Arbeit nicht mit der Realisierung einer komponentenbasierten Verteilungsinfrastruktur, die sich für die generische Durchsetzung harter Dienstgüteanforderungen eignet. Zwar ist die spätere Erweiterung der Ergebnisse in diese Richtung nicht grundsätzlich ausgeschlossen, der Bereich der Dienstgüteverwaltung ist jedoch kein primäres Ziel.

5 Konzeption

Um die im vorhergehenden Kapitel beschriebenen Ziele zu erreichen, müssen zunächst offene Fragen bezüglich der zugrunde liegenden Plattform geklärt werden. Aus der Fülle verschiedener Programmiersprachen und Infrastrukturen muss ein möglichst optimaler Satz an Werkzeugen gewählt werden, um bestehende Konzepte optimal einzubetten und gleichzeitig den Entwicklungsaufwand zu verringern.

Darüber hinaus müssen die notwendigen Abstraktionen des Komponentenmodells diskutiert und ausgewählt werden. Bei der Diskussion werden vor allem bestehende Komponentensysteme wie Enterprise JavaBeans und Corba Components zum Vergleich herangezogen. Die Auswahl folgt letztlich, wie im vorherigen Kapitel erläutert, dem Prinzip der Minimalität.

Im letzten Teil dieses Kapitels werden zusätzliche Konzepte vorgestellt, die unabhängig vom Komponentenmodell sind. Mit ihrer Hilfe soll eine hohe Modularität der Verteilungsinfrastruktur erzielt werden und Anhaltspunkte für deren Strukturierung gegeben werden.

5.1 Plattform

Möglichst weitgehende Plattformunabhängigkeit ist ein Ziel vieler Softwareprojekte. Die komponentenbasierte Verteilungsinfrastruktur, die in dieser Arbeit entwickelt wird, zielt ebenfalls darauf ab. Um plattformunabhängige Software zu entwickeln sind zwei Ansätze üblich. Zum einen kann bei der Auswahl der genutzten Abstraktionen darauf geachtet werden, dass die Zahl plattformabhängiger Abstraktionen minimiert wird und zum anderen kann eine Abstraktionsschicht verwendet werden, die von den Details der Plattform abstrahiert.

Moderne Sprachen wie Java [Joy00] und C# [ECMA02] bieten plattformunabhängige Abstraktionen bereits auf der Ebene der Programmiersprache. Bei der Entwicklung von Programmen für heterogene Systeme macht sich Plattformunabhängigkeit auf Ebene der Programmiersprache bezahlt, da der Aufwand für die Portierungen vermieden werden kann. Aufgrund dieser Tatsache könnte man geneigt sein, plattformunabhängige den plattformabhängigen Sprachen generell vorzuziehen. Ein solcher Schluss wäre allerdings mit Blick auf ubiquitäre Rechnersysteme verfrüht. Im Gegensatz zu plattformabhängigen machen plattformunabhängige Programmiersprachen häufig Gebrauch von relativ umfangreichen Laufzeitumgebungen. Da ubiquitäre Rechnersysteme auch ressourcenarme Rechner enthalten können, ist die Nutzung mächtiger Laufzeitumgebungen wie J2SE [Sun03] oder .Net Framework [Microsoft03a] ausgeschlossen.

Diese Fakten führten in der jüngsten Zeit dazu, dass sowohl Sun Microsystems als auch Microsoft Cooperation schlankere Versionen ihrer Laufzeitumgebungen veröffentlichten. Mit der Java 2 Micro Edition [Sun02a] und dem .Net Compact Framework [Microsoft03b] sind Laufzeitumgebungen für Java und C# verfügbar, die für ressourcenarme Geräten geeignet sind. Damit stehen dieser Arbeit prinzipiell Umgebungen für C# und Java zur Verfügung, die Plattformunabhängigkeit und Komfort neuer Programmiersprachen mit schlanken Laufzeitumgebungen verbinden.

Die Entscheidung für die eine oder andere Plattform ist im Bezug auf die sprachlichen Unterschiede von C# und Java für diese Arbeit beliebig. Es stellt sich also die Frage, welche zusätzlichen Abstraktionen benötigt werden, um eine komponentenbasierte Verteilungsinfrastruktur mit den gewünschten Eigenschaften effizient zu entwickeln.

In seiner Vision von Ubiquitous Computing betont Weiser den Einfluss und die Notwendigkeit der Vernetzung verschiedener Geräte. Damit wird deutlich, dass der Kommunikationsmechanismus in jeder Infrastruktur für Anwendungen in ubiquitären Rechnersystemen eine zentrale Rolle spielt.

Im Bezug auf die Eignung verschiedener Kommunikationsmechanismen für den Einsatz in ubiquitären Rechnersystemen lohnt es sich, einen genaueren Blick auf die existierenden Ansätze zu werfen. Bei dieser Betrachtung wird klar, dass zumindest die Verteilungsinfrastrukturen für Ubiquitous Computing über gemeinsame Funktionalität verfügen. Discovery- und Namens- oder Verzeichnisdienste sind zentrale Bestandteile aller Ansätze.

Es ist offensichtlich, dass die Fähigkeit, entfernt verfügbare Funktionalitäten zu entdecken, auf die eine oder andere Weise umgesetzt werden muss, um die Anforderungen, die sich aus der Mobilität ubiquitärer Rechnersysteme ergeben, zu erfüllen. Im Hinblick auf die Einbettung bestehender Konzepte ist es demnach wünschenswert, wenn ein Kommunikationsmechanismus diese Fähigkeit liefert.

Diese Anforderung schränkt die möglichen Kommunikationsmechanismen stark ein. Gängige Konzepte mit relativ niedrigem Abstraktionsgrad wie Sockets und klassische Remote Procedure Call Pakete, werden weitgehend ausgeschlossen. Tatsächlich bleiben von den hier vorgestellten Ansätzen, unter Ausschluss der komponentenbasierten, nur noch der eventbasierte Kommunikationsmechanismus von one.world und die objektorientierten von Jini, BASE und Gaia übrig.

Der Vergleich der Mechanismen zeigt, dass das objektorientierte Paradigma eine höhere Entkopplung von Anwendung und entfernter Kommunikation bietet. Tatsächlich argumentieren die Entwickler von one.world für die geringe Entkopplung von Anwendung und Kommunikationsmechanismus. Nied-

rige Kopplung könne die Umsetzung von Anwendungen erschweren, falls diese den Kommunikationsmechanismus, beispielsweise aufgrund von harten Dienstgüteanforderungen, direkt beeinflussen müssen [Grimm02a]. Diese Arbeit zielt jedoch nicht auf die Umsetzung solcher Dienstgüteanforderungen ab. Demnach ist die niedrigere Kopplung der höheren vorzuziehen.

Damit kann die Gruppe der in Frage kommenden Kommunikationsmechanismen auf die objektorientierten Ansätze von Jini, BASE und Gaia eingeschränkt werden. Von diesen drei Ansätzen ist BASE am besten für die Entwicklung einer komponentenbasierten Verteilungsinfrastruktur geeignet. Hinsichtlich der Flexibilität der Kommunikationsprotokolle bietet der Plug-in Mechanismus von BASE eine höhere Flexibilität als Jini. Gleichzeitig basiert BASE auf der Java 2 Micro Edition und ist plattformunabhängig, im Gegensatz zu Gaia, das auf einer in C++ implementierten Version von DynamicTAO basiert und daher plattformabhängig ist. BASE verbindet flexible Kommunikation, die nur lose mit Anwendungen gekoppelt ist, mit den Vorteilen der Plattformunabhängigkeit.

Durch die Wahl von BASE als Kommunikationsmechanismus wird letztlich auch die Entscheidung für den Einsatz der Java 2 Micro Edition gegen den Einsatz des .NET Compact Frameworks getroffen.

Auf Basis der Programmiersprache sind folgende Abstraktionen verfügbar:

- *Typsicheres Typsystem*; Java ist eine typsichere Sprache (strong typed). Das Typsystem unterscheidet zwei Arten von Typen, Basis- und Objekttypen. Basistypen sind eine feste Menge von Typen, die beim Methodenaufruf als Wertparameter übergeben werden (call-by-value). Objekttypen können durch Klassen oder Interfaces definiert werden und werden beim Methodenaufruf als Referenzparameter übergeben (call-by-reference)
- *Methoden, Interfaces, Klassen und Objekte*; Code wird in Java immer in Methoden gekapselt, die innerhalb von Klassen definiert werden. Mit Ausnahme von statischen Methoden können Methoden nur auf Basis von Instanzen dieser Klassen, den so genannten Objekten, aufgerufen werden. Methoden können über eine beliebige Anzahl fester Parameter verfügen und wahlweise einen Wert zurückgeben. Die Signatur einer Methode wird durch den Methodenname, die Parameter und den Rückgabewert definiert. Interfaces definieren eine Menge von Feldern und Methoden. Klassen können Interfaces implementieren, wodurch sie typkompatibel mit den implementierten Interfaces werden.
- *Vererbung*; Mit Ausnahme der Basisklasse Object erbt jede Klasse von genau einer Klasse. Die Methoden der Basisklasse können überschrieben werden. Die Subklasse ist typkompatibel zur Basisklasse. Inter-

faces können eine beliebige Anzahl an Interfaces implementieren. Dadurch wird das implementierende Interface typkompatibel zu den implementierten Interfaces.

- *Multithreading, Exceptions, Garbage Collection*; Java unterstützt Multithreading und monitor-style-programming auf Ebene der Sprache. Dazu kann jedes Objekt mittels eines Sprachkonstrukts als Monitor zur Synchronisation verwendet werden. Weiterhin bietet Java Konstrukte zum Auslösen und Behandeln von Exceptions. Exceptions selbst sind als Klassen bzw. Objekte modelliert, die ein bestimmtes Interface namens Throwable implementieren. Java verfügt darüber hinaus über automatische Garbage Collection.

Auf Basis des Kommunikationsmechanismus sind zusätzlich folgende Abstraktionen verfügbar:

- *Dienste, Diensttyp und Dienstinstanzen*; Dienste sind Klassen, die von einer bestimmten Klasse erben, die den Lebenszyklus des Dienstes definiert, und über eine bestimmte Beschreibung verfügen. Die Beschreibung enthält im Wesentlichen Informationen über die Implementierung und den Typ des Dienstes, der durch genau ein Interface definiert wird. Um einen Dienst nutzen zu können, muss zunächst eine Instanz gebildet werden, d.h., die Klasse, die den Dienst implementiert, muss instanziert werden.
- *Dienstregistrierung*; Dienstinstanzen können andere Dienstinstanzen nutzen. Zur Suche von Dienstinstanzen muss die Dienstregistrierung genutzt werden. Bei der Dienstregistrierung werden alle lokal verfügbaren Dienste mit ihrem Typ registriert. Die Dienstregistrierung erlaubt sowohl typ- als auch namensbasierte Suchen auf dem lokalen Gerät und allen entfernt verfügbaren Geräten. Die Dienstregistrierung sorgt für die Generierung von Stellvertreterobjekten, die Aufrufe unabhängig von der Lokation der Dienstinstanz erlauben.

5.2 Komponentenmodell

Neben der Plattform hat das Komponentenmodell den größten Einfluss auf die Gestalt einer komponentenbasierten Verteilungsinfrastruktur und folglich auf die Anwendungen, die damit entwickelt werden. Die Wahl der Eigenschaften kann den Aufwand bei der Entwicklung von Anwendungen erhöhen oder verringern.

Um geeignete Eigenschaften für das Komponentenmodell zu definieren, muss zunächst das zugrunde liegende Anwendungsmodell diskutiert werden. Dabei werden Einschränkungen gemacht, die die Komplexität der Infrastruktur

deutlich reduzieren und dabei die Anwendungsentwicklung möglichst gering beeinflussen.

Ein zentraler Teil von Softwarekomponenten sind ihre Programmeinheiten. Ihre Struktur liefert die grundlegenden Strukturierungsmöglichkeiten für die Anwendungen, die auf der Basis der Softwarekomponenten entwickelt werden. Daher werden nach der Vorstellung des Anwendungsmodells zunächst die Abstraktionen ausgearbeitet, die von den Programmeinheiten geliefert werden müssen.

Im Anschluss daran wird auf den zweiten, zentralen Teil eingegangen, die Beschreibung der Softwarekomponenten. Sie ermöglicht die angestrebte Automatisierung von wiederkehrenden Aufgaben. Gleichzeitig wird mit ihrer Hilfe die durchgängige Unterstützung von kontextuellen Informationen ermöglicht.

Sowohl die Programmeinheiten als auch die Beschreibung der Softwarekomponenten sind zunächst statisch. Daher muss zusätzlich zur statischen Form und Funktion, das dynamische Verhalten zur Laufzeit der Verteilungsinfrastruktur beschrieben werden. Der Fokus liegt dabei auf der automatischen Bindung von Softwarekomponenten und dem Verhalten der Verteilungsinfrastruktur im Hinblick auf die Dynamik ubiquitärer Rechnersysteme.

5.2.1 Anwendungsmodell

Wie bereits aus dem Beispiel im vorhergehenden Kapitel deutlich wird, sind Anwendungen aus Softwarekomponenten zusammengesetzt. Diese Komposition der Softwarekomponenten zur Bildung von Anwendungen erfolgt entlang von Angebot und Anforderungen, die von jeder Komponente einzeln spezifiziert werden.

Eine Anwendung kann demnach zur Laufzeit als zusammenhängender, gerichteter Graph aufgefasst werden, wobei ein Knoten des Graphs eine Softwarekomponente repräsentiert und eine Kante des Graphs eine Anforderung darstellt, die durch ein Angebot erfüllt wird. Dieser Graph kann sich während der Ausführung der Anwendung verändern, da sich die Knoten durch spontane Vernetzung und Mobilität von Rechnern ändern können.

Die dynamische Struktur von Anwendungen wirft einige Fragen auf. Sieht man die Anwendung in einem zeitlichen Ablauf, stellt sich zunächst die Frage, wie Anwendungen gestartet, beendet und identifiziert werden können. Die Identifikation von Anwendungen ist besonders für die Behandlung von Fehlern wichtig. Mit einem kurzen Beispiel lässt sich dieser Sachverhalt am einfachsten verdeutlichen.

Angenommen, eine Anwendung besteht aus zwei Softwarekomponenten A und B, wobei A die Komponente B benötigt und B die Komponente A. Bricht nun die Verbindung zwischen beiden Komponenten ab, gibt es mehrere Möglichkeiten auf den Fehler zu reagieren. Zum einen könnte die Komponente A versuchen, eine neue Komponente B′ zu finden, zum anderen könnte auch die Komponente B versuchen, eine neue Komponente A′ zu finden. Ein Zustand in dem beide Komponenten versuchen, einen Ersatz für die verloren gegangene Komponente zu finden, ist nicht wünschenswert, da dies im Erfolgsfall zur Ausführung von zwei Anwendungen führen würde.

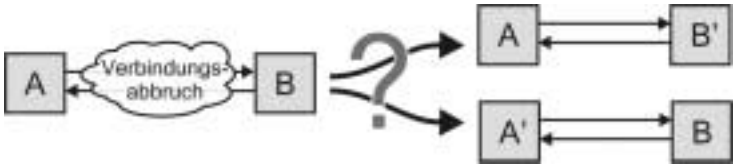

Abbildung 9 – Identifikation von Anwendungen

Um dieses Problem zu vermeiden und für alle möglichen Fälle eindeutig entscheiden zu können, welcher Teil des Graphs fortgeführt werden soll, wird in dieser Arbeit eine Komponente verwendet, um die Anwendung eindeutig zu identifizieren. Wird diese Komponente gestartet, ist das gleichzusetzen mit dem Start der Anwendung. Bricht die Ausführung der Komponente ab, wird der Rest der Anwendung nicht länger ausgeführt. Verliert der Graph durch einen Verbindungsabbruch den Zusammenhang, wird der Teil des Graphs fortgeführt, der die Komponente enthält, die stellvertretend für die Anwendung steht.

Neben dem Identifikationsproblem können zyklische Graphen Schwierigkeiten verursachen. Zyklische Abhängigkeiten von Softwarekomponenten erfordern bei der Komposition von Anwendungen einen Erkennungsmechanismus. Können Zyklen nicht erkannt werden, kann das dazu führen, dass die Komposition nicht terminiert.

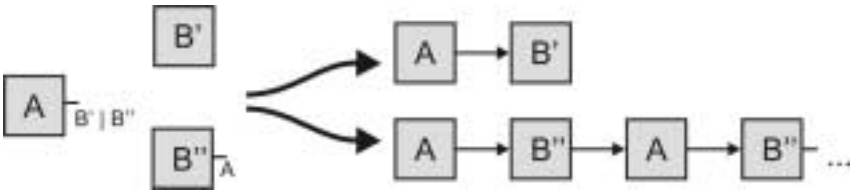

Abbildung 10 – Zyklische Abhängigkeiten

Angenommen, eine Anwendung besteht aus einer Komponente A, die entweder eine Komponente B′ oder B′′ benötigt. B′ hat keine weiteren Abhängig-

keiten und B'' benötigt eine Komponente A. Wird bei der Komposition der Anwendung die Komponente B'' an A gebunden, wobei der Zyklus von A und B'' nicht entdeckt wird, wird versucht eine Komponente A zu finden. Wird jedoch die Komponente B' gebunden, so ist die Komposition abgeschlossen. Die permanente Bindung neuer Komponenten A und B'' führt theoretisch[2] dazu, dass die Komposition der Anwendung nicht terminiert.

Um diesem Problem zu begegnen, gibt es zwei mögliche Ansätze. Zum einen kann der Algorithmus zur Komposition von Anwendungen auf die Erkennung von Zyklen ausgelegt werden. Zum anderen kann das Zyklenproblem per Definition gelöst werden, indem man Zyklen generell verbietet. Dieser Weg wird in dieser Arbeit verfolgt, da zyklische Abhängigkeiten letztlich zur mehrfachen Nutzung von Komponenten führen.

Diese mehrfache Einbettung und Wiederverwendung des Angebots einer zustandsbehafteten Komponente macht die Synchronisation zwischen Nutzern erforderlich. Bei zustandsbehafteten Komponenten können die Ergebnisse von Anfragen von ihrer Reihenfolge abhängen. Im Extremfall kann die Korrektheit der Berechnung von einer bestimmten Sequenz der Aufrufe abhängen. Wird eine Komponente gleichzeitig von mindestens zwei Komponenten verwendet, müssen diese Komponenten die Sequenzen ihrer Aufrufe abstimmen.

Um die Synchronisation zwischen Nutzern zu vermeiden, wird in dieser Arbeit die Zahl der gleichzeitigen Nutzungen von Komponenten auf eine Nutzung beschränkt. Da diese Einschränkung auch die Mächtigkeit des Modells einschränkt, muss sie im Zuge weiterer Arbeiten nochmals bewertet und eventuell gelockert werden.

Betrachtet man nun den Graph unter der Einschränkung, dass eine Komponente maximal von einer Komponente genutzt wird und die Anwendung über eine Komponente identifiziert werden kann, stellt man fest, dass Anwendungen nun nicht mehr durch einen beliebigen Graphen, sondern durch einen Baum repräsentiert werden. Dabei ist die Wurzel des Baums genau die Komponente, die eine Anwendung repräsentiert. In Anbetracht dessen kann eine Anwendung einfach in Schichten gegliedert werden, wobei jeweils eine Ebene des Baums eine Schicht repräsentiert.

Entlang dieser Schichten kann nun geklärt werden, welche Richtung von Aufrufen unterstützt werden soll. Grundsätzlich ist klar, dass Softwarekomponenten, die andere Softwarekomponenten benötigen, der Funktionalität dieser Komponenten bedürfen. Daher ist es sinnvoll, die direkte Kommunikati-

[2] In einer realen Umgebung, in der Ressourcen begrenzt sind, würde die Ausführung nach Konsum aller verfügbaren Ressourcen abbrechen.

on, ausgehend von einer anfordernden Komponente hin zu einer angeforderten zu ermöglichen. Innerhalb des vorgestellten Schichtenmodells würde dies einem Downcall entsprechen. Downcalls sind in allen vorgestellten komponentenbasierten Verteilungsinfrastrukturen enthalten und stellen in der Regel die primäre Richtung für Aufrufe dar.

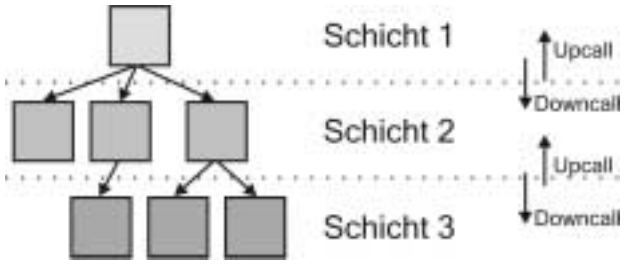

Abbildung 11 – Richtung von Aufrufen

Wenn die Richtung vom Nutzer hin zum Anbieter unterstützt wird, stellt sich die Frage, inwiefern es sinnvoll ist, die Gegenrichtung, so genannte Upcalls, zu unterstützten. Upcalls geben unteren Schichten die Möglichkeit, Änderungen ihres Zustands an darüberliegende Schichten zu signalisieren. Damit dienen sie primär zur Schonung von Ressourcen, da darüberliegende Schichten keine Ressourcen verwenden müssen, um untere Schichten regelmäßig zu prüfen. In einem verteilten System, in dem die verschiedenen Schichten möglicherweise auf unterschiedlichen Rechnern liegen, ist der positive Einfluss von Upcalls noch größer, da entfernte Aufrufe mehr Ressourcen benötigen. In kommerziellen komponentenbasierten Verteilungsinfrastrukturen werden Upcalls in der Regel mit Hilfe von Eventmechanismen realisiert.

Um die Richtungen von Aufrufen vollständig zu beleuchten, müssen noch Aufrufe bewertet werden, die zwischen Komponenten derselben Schicht und zwischen Komponenten in nicht direkt angrenzenden Schichten gemacht werden könnten. Solche Aufrufe sind generell nicht wünschenswert, da sie letztlich gegen das Konzept von Softwarekomponenten verstoßen. Ein Aufruf von einer Komponente zur anderen würde bedeuten, dass es eine Abhängigkeit zwischen Komponenten gibt, die nicht spezifiziert ist. Die Definition des Begriffs verlangt ausdrücklich die Spezifikation aller Abhängigkeiten.

In dieser Arbeit werden daher Downcalls und Upcalls zwischen nutzender Komponente und genutzter Komponenten unterstützt. Andere Aufrufrichtungen sind nicht erwünscht und werden daher nicht unterstützt. Der Ausschluss anderer Richtungen bedeutet nicht gleichzeitig, dass diese nicht in höheren Schichten nachgebildet werden können. Damit ist lediglich gemeint, dass die Verteilungsinfrastruktur keine Mechanismen vorsieht, um solche Aufrufe direkt abzubilden.

Zusammenfassend besteht eine Anwendung damit aus einem Baum von Komponenten. Die Aufrufe von einer Komponente zur anderen werden innerhalb dieses Baumes nur unterstützt, wenn die aufrufende Komponente entweder Kind oder Vater der aufgerufenen Komponente ist.

5.2.2 Programmeinheit

Aus dem zuvor beschriebenen Anwendungsmodell ergibt sich die Forderung, dass sich Anwendungen aus Komponenten zusammensetzen müssen. Die Komponenten können eigene Funktionalität anbieten und Funktionalität anderer Komponenten nutzen. Die Kommunikation soll sowohl mit dem Nutzer der eigenen Funktionalität, als auch mit den Anbietern der genutzten Funktionalität möglich sein.

Um diese Forderungen umzusetzen, muss zunächst genauer betrachtet werden, was der Begriff Funktionalität in diesem Zusammenhang bedeutet. Alle vorgestellten komponentenbasierten Verteilungsinfrastrukturen gehen in diesem Punkt denselben Weg, wobei die Funktionalität einer Komponente durch eine Menge von Methoden beschrieben wird. Da diese Methoden nur syntaktische Schnittstellen darstellen, setzten die vorgestellten Verteilungsinfrastrukturen voraus, dass mit der syntaktischen Schnittstelle implizit eine bestimmte Semantik verbunden ist. Durch diese Vereinfachung entfällt die Notwendigkeit, die Semantik einer Schnittstelle zu beschreiben. In dieser Arbeit wird auf dieselbe Vereinfachung zurückgegriffen, um die Entwicklung von Komponenten nicht durch eine formale Beschreibung der Semantik einer Komponente zu erschweren.

Um eine Menge von Methoden zu gruppieren, bietet Java zwei Abstraktionen, Interfaces und Klassen. Im Hinblick auf die Beschreibung von Funktionalität sind Interfaces besser geeignet als Klassen, da Klassen neben dem syntaktischen Interface die Implementierung enthalten. Die Beschreibung der Implementierung ist für die Definition von Funktionalität nicht notwendig und würde dazu führen, dass die Komposition von Komponenten unnötig unflexibler würde. Auch die vorgestellten Verteilungsinfrastrukturen weichen in diesem Punkt nicht voneinander ab. Alle verwenden ein sprachliches Interfacekonstrukt, um Methoden zu gruppieren und die Funktionalität von Komponenten zu beschreiben.

Unterschiede zwischen den Verteilungsinfrastrukturen gibt es im Hinblick auf die Zuordnung von Interfaces und Komponenten. Während Corba Components und DCOM/COM+ eine 1-zu-n Zuordnung zwischen Komponenten und Interfaces ermöglichen, ordnen Enterprise JavaBeans einer Komponente genau ein Interface zu. Da in vielen Sprachen die Bildung neuer Interfaces aus einer Menge bestehender Interfaces möglich ist, stellt die Zuordnung von ei-

nem Interface zu einer Komponente keine Einschränkung im Bezug auf die Strukturierungsmöglichkeiten dar. Die 1-zu-n Zuordnung birgt jedoch keinen großen Aufwand und führt nicht, wie die 1-zu-1 Zuordnung, zur Einführung unnötiger Interfaces zur Gruppierung von Funktionalität. Durch die Möglichkeit mehrere Interfaces anzubieten, wird darüber hinaus ermöglicht, dass Komponenten auch Teile ihrer Funktionalität anbieten können. In dieser Arbeit wird daher eine 1-zu-n Zuordnung von Komponente und Interfaces verwendet.

Nachdem nun geklärt ist, wie Funktionalitäten beschrieben werden, muss ihre Nutzung genauer betrachtet werden. Nutzer von Komponenten basieren auf deren Funktionalität. Der Ausfall einer genutzten Komponente kann dazu führen, dass die nutzende Komponente nicht mehr ausgeführt werden kann. Damit ist klar, dass die Funktionalität von angeforderten Komponenten in der Regel zuverlässig genutzt werden soll, schließlich wurde die Bindung der Komponente vom Nutzer initiiert und ist damit ein direktes Resultat der Anforderungen des Nutzers.

Im Hinblick auf Kommunikation in die Gegenrichtung, d.h., auf Kommunikation, die von der genutzten Komponente initiiert wurde, gilt dies nicht. Um eine möglichst hohe Flexibilität bei der Komposition von Anwendungen zu gewährleisten, sollte die genutzte Komponente keinerlei Anforderungen an die nutzende Komponente stellen. Ein auf beliebigen Methoden basierendes Konzept ist zu allgemein, um zu gewährleisten, dass die genutzte Komponente keinerlei Anforderungen an das Verhalten ihres Nutzers stellt. Um die Unabhängigkeit einer genutzten Komponente von der nutzenden Komponente zu gewährleisten, darf eine Komponente nicht auf den Ergebnissen von Aufrufen basieren, die sie zu ihrem Nutzer schickt.

In der Verteilungsinfrastruktur, die in dieser Arbeit entwickelt wird, werden daher Aufrufe vom Nutzer zur genutzten Komponente synchron durchgeführt. Um die Konsistenz zustandsbehafteter Komponenten zu gewährleisten, werden Aufrufe maximal einmal ausgeführt. Theoretisch wären auch alternative Aufrufsemantiken und asynchrone Aufrufe mit Callbacks oder Future Objekten denkbar. Da in dieser Arbeit versucht wird, ein minimales System zu entwickeln, werden diese jedoch nicht umgesetzt. Kommunikation in Richtung der Bindung entspricht damit weitestgehend einem lokalen Methodenaufruf.

Für die Gegenrichtung, d.h. von der genutzten Komponente zum Nutzer, wird hingegen asynchrone Kommunikation ohne Callbacks und Future Objekte verwendet. Auch für diese Kommunikation ist es wünschenswert, wenn Aufrufe maximal einmal ausgeliefert werden. Um asynchrone Aufrufe ohne Callbacks oder Future Objekte zu ermöglichen, dürfen Methoden keinerlei Ergebnisse zurückgeben. Weiterhin müssen Parameter entweder kopiert wer-

den oder es muss sichergestellt werden, dass nur lesend darauf zugegriffen wird. Damit entspricht die Semantik der Aufrufe weitgehend der Semantik von klassischen Eventmechanismen. Im Unterschied dazu gibt es jedoch nur einen Konsumenten, den Nutzer der Komponente.

Beide Aufrufrichtungen müssen mit Hilfe von BASE, dem zugrunde liegenden Kommunikationsmechanismus, umgesetzt werden. Da BASE als Vertreter des objekt- bzw. dienstorientierten Paradigmas Aufrufe durch Stellvertreterobjekte generiert und damit eine hohe Abstraktion von der Umsetzung der Kommunikation erlaubt, werden für die Kommunikation von Komponenten ebenfalls Stellvertreterobjekte verwendet. Damit müssen Komponenten neben den Methoden zur Bereitstellung ihrer Funktionalität und zum Empfang von Aufrufen genutzter Komponenten auch über Methoden zur Verwaltung von Stellvertreterobjekten verfügen. Diese Methoden werden dann von der Verteilungsinfrastruktur angesteuert, um die Komponenten mit den benötigten Stellvertreterobjekten zu versorgen.

Durch die Nutzung von Java Interfaces zur Beschreibung von Funktionalität ergibt sich ein technisches Problem. Die Implementierung eines Interface erfolgt innerhalb einer Klasse. Dabei kann ein Interface keine statischen Methoden vorgeben, was dazu führt, dass zur Nutzung der Implementierung eines Interface letztlich ein Objekt erzeugt werden muss. Dadurch wird es notwendig, Instanzen von Klassen zu bilden, die die Schnittstellen der Komponenten anbieten. Dem Paradigma der Objektorientierung folgend, kann zwischen Komponente und Komponenteninstanz unterscheiden werden. Die Programmeinheit einer Komponente besteht damit aus einer Menge von Klassen, unter denen es eine spezielle Klasse gibt, die alle angebotenen Interfaces der Komponente implementiert. Die Programmeinheit der Komponenteninstanz entsteht durch Erzeugen einer Instanz dieser Klasse.

Grundsätzlich stellt sich damit die Frage, wann eine Instanz der Komponente gebildet werden soll. Zum einen könnte eine Instanz beim Start der Verteilungsinfrastruktur gebildet werden. Zum anderen könnten Instanzen auch bei Bedarf erzeugt werden. Die bedarfsgesteuerte Bildung von Instanzen hat eine Reihe von Vorzügen. Zunächst werden Ressourcen besser ausgenutzt, da keine ungebundenen Instanzen gebildet werden müssen. Weiterhin erlaubt sie die mehrfache Bildung von Instanzen derselben Komponente, wodurch ein erheblicher Nachteil des Anwendungsmodells gemildert werden kann. Im vorgestellten Anwendungsmodell wurde die mehrfache Bindung von Komponenten ausgeschlossen, um die gemeinsame Nutzung von Komponenten und den daraus resultierenden Bedarf an Synchronisationsmechanismen zu vermeiden. Durch die Einführung von Komponenteninstanzen erfolgt die Bindung nicht mehr auf Basis der Komponenten selbst, sondern auf Basis ihrer Instanzen. Die Einschränkung, mehrfache Bindungen von Komponenten

zu verbieten, bezieht sich damit nicht länger auf Komponenten sondern auf Komponenteninstanzen. Erlaubt man demnach die Bildung mehrer Komponenteninstanzen, können Komponenten de facto mehrfach verwendet werden.

Damit wird deutlich, dass die bedarfsgesteuerte Bildung mehrerer Instanzen der Bildung einer Instanz beim Start der Verteilungsinfrastruktur vorzuziehen ist. Allerdings wird durch diese Wahl ein Mechanismus zum Starten von Komponenteninstanzen, oder genauer gesagt zur Verwaltung des Lebenszyklus notwendig. Im Hinblick auf die Zustände eines Lebenszyklus von Komponenten sind minimal die Zustände, „Instanz erzeugt“ und „Instanz nicht erzeugt bzw. entladen“, notwendig. In einer minimalen Implementierung könnten diese Zustände durch sprachliche Konstrukte und Mechanismen getriggert werden. Wird das Objekt erzeugt, das die Interfaces der Komponente implementiert, befindet es sich im Zustand „Instanz erzeugt“. Wird dieses Objekt vom Garbage Collector entladen, wird der Zustand gewechselt. Die minimale Implementierung hat jedoch einige gravierende Nachteile. Zunächst gibt es in Java keinerlei Möglichkeit, den Zeitpunkt für das Entladen eines Objektes festzulegen. Dadurch könnte es vorkommen, dass Komponenteninstanzen lokale Ressourcen zu lange in Anspruch nehmen. Darüber hinaus sind durch die automatische Bindung benötigter Komponenten Initialisierungen notwendig, die nicht generell im Konstruktor eines Objekts abgehandelt werden können und daher einen weiteren Zustand wünschenswert machen.

Abbildung 12 – Lebenszyklus für Komponenteninstanzen

Ein sinnvoller und dennoch überschaubarer Lebenszyklus sollte daher vier Zustände besitzen. Der erste und letzte Zustand entspricht dabei den Zuständen des minimalen Lebenszyklus. Dazwischen werden noch die Zustände „Instanz gestartet“ und „Instanz gestoppt“ eingeführt. Diese Aufrufe werden von der Verteilungsinfrastruktur getriggert. Der Zustand „Instanz gestartet“ wird nach Abschluss aller notwendigen Initialisierungen aufgerufen und signalisiert, dass die Komponenteninstanz für den Betrieb notwendige Ressourcen reservieren soll. In diesem Zustand muss die Komponenteninstanz die angebotene Funktionalität liefern. Nachdem die Komponenteninstanz nicht mehr benötigt wird, sorgt die Verteilungsinfrastruktur dafür, dass sie den Zustand wechselt. Im Zustand „Instanz gestoppt“ können alle Ressourcen freigegeben werden. Im Gegensatz zum Garbage Collector kann die Infrastruktur diesen Zustand zu jedem gewünschten Zeitpunkt herbeiführen, wodurch die komponentenabhängigen Ressourcen nur für einen minimalen Zeitraum belegt sind. Durch diese Erweiterung des Lebenszyklus kann die Verteilungsinfrastruktur die Nutzung komponentenabhängiger Ressourcen direkt steuern. Aufgrund der automatischen Garbage Collection kann lediglich der Speicher, der von Java für die Erzeugung von Objekten verwendet wird, nicht direkt von der Verteilungsinfrastruktur freigegeben werden.

Insgesamt setzten sich die Programmeinheiten von Komponenten damit aus vier verschiedenen Teilen zusammen. Zunächst gibt es eine Gruppe von Methoden, die aufgrund implementierter Interfaces vorhanden sind. Diese Methoden stellen die angebotene Funktionalität der Komponente bereit. Darüber hinaus verfügt die Komponente über Methoden, die Aufrufe von genutzten Komponenten entgegen nehmen. Weiterhin gibt es eine Reihe von Methoden, die genutzt werden, um auf Stellvertreterobjekte zuzugreifen, die von der Verteilungsinfrastruktur bereitgestellt werden. Die Stellvertreterobjekte repräsentieren dabei entweder die nutzende Komponenteninstanz oder genutzte Komponenteninstanzen. Zur Erzeugung von Komponenteninstanzen müssen Methoden zur Verwaltung des Lebenszyklus bereitgestellt werden.

5.2.3 Beschreibung

Der Definition von Komponenten zufolge, verfügen Komponenten über vertraglich spezifizierte Schnittstellen und kontextuelle Abhängigkeiten sind ausschließlich explizit. Die bereits vorgestellten Programmeinheiten alleine bieten durch die explizite Spezifikation von unterstützter Funktionalität bereits einen hohen Detaillierungsgrad im Bezug auf Beschreibung ihrer Funktionalität. Die Beschreibung von Abhängigkeiten ist hingegen relativ schwach ausgeprägt. Im Hinblick auf kontextuelle Abhängigkeiten bieten die Programmeinheiten keine Möglichkeiten zur Definition. Diese Unzulänglichkei-

ten werden anhand einer zusätzlichen Beschreibung ausgemerzt. Dazu werden Inhalt und Form erarbeitet.

Die Definition von Komponenten verlangt zwar vertraglich spezifizierte Schnittstellen, es wird jedoch nicht auf den Inhalt, die Form oder die Art dieser Verträge eingegangen. Daher ist es zunächst erforderlich, die verschiedenen Arten von Verträgen zu klassifizieren und auf Basis dieser Klassifikation zu bestimmen, welche Vertragstypen sinnvoll für die Erstellung einer komponentenbasierten Verteilungsinfrastruktur für Anwendungen in ubiquitären Rechnersystemen sind.

In [Beugnard99] werden Verträge für Komponenten entlang ihres Inhalts in die folgenden vier Klassen aufgeteilt.

- *Syntax*; Verträge zur Beschreibung der Syntax spezifizieren das Angebot einer Komponente syntaktisch. Dazu gehören in der Regel die Operationen, die eine Komponente ausführen kann, sowie die Ein- und Ausgabeparameter dieser Operationen und eventuell mögliche Exceptions, die durch die Ausführung einer Operation ausgelöst werden können.
- *Verhalten*; Verträge zur Beschreibung des Verhaltens einer Komponente spezifizieren die Vor- und Nachbedingungen einzelner Operationen, die von der Komponente angeboten werden. Darüber hinaus spezifizieren sie die globalen Invarianten der Komponente.
- *Synchronisation*; Verträge zur Beschreibung der Synchronisation spezifizieren globale Eigenschaften von Komponenten im Bezug auf die Synchronisation von Methodenaufrufen. Dabei werden beispielsweise Aufrufsequenzen vereinbart, die zur korrekten Funktion einer Komponente eingehalten werden müssen.
- *Dienstgüte*; Verträge zur Beschreibung der Dienstgüte spezifizieren die nicht-funktionalen Parameter die von einer Komponente geliefert werden. Nicht-funktionale Parameter können dabei entweder statisch vorgegeben oder dynamisch durch Verhandlungen vereinbart werden.

Im Hinblick auf die Auflösung von Abhängigkeiten, die von der komponentenbasierten Verteilungsinfrastruktur automatisiert werden soll, ist es offensichtlich, dass die syntaktische Spezifikation von Komponenten erforderlich ist. Die syntaktische Spezifikation der Funktionalität ermöglicht der Verteilungsinfrastruktur, Angebot und Anforderung von Komponenten zu vergleichen. Im Gegensatz zu der vorangestellten Klassifizierung ist jedoch die einzelne Auflistung der Operationen einer Komponente sowie deren Ein- und Ausgabeparameter und Exceptions nicht notwendig, da die Programmeinheiten ihre Funktionalität auf Basis von Interfaces anbieten. Geht man davon aus,

dass diese Interfaces unveränderlich sind[3], enthält die Spezifikation der Interfaces bereits die genaue Spezifikation der Syntax angebotener Operationen. Um die Bindung abhängiger Komponenten zu automatisieren, muss die Beschreibung der Komponenten demnach zumindest die benötigten und angebotenen Interfaces enthalten. Darüber hinaus müssen die Methoden zum Empfang von Aufrufen genutzter Komponenten, sowie die Aufrufe an nutzende Komponenten syntaktisch beschrieben werden. Als letztes muss noch die Syntax aller Methoden zur Verwaltung von Stellvertreterobjekten beschreiben werden.

Die Beschreibung des Verhaltens angebotener und benötigter Funktionalität wird in die Komponentenbeschreibungen dieser Arbeit nicht integriert. Aus softwaretechnischen Gesichtspunkten ist die Integration von Verhaltensbeschreibungen zwar durchaus sinnvoll, da sie einen Schutz vor Programmierfehlern ermöglichen, der über den gewöhnlicher Typsysteme hinausgeht. Gegen die Einbettung solcher Beschreibungen spricht jedoch die Tatsache, dass die Entwicklung der Verteilungsinfrastruktur durch die Prüfung von Vor- und Nachbedingungen oder Invarianten erheblich komplexer wird, wenn solche Mechanismen nicht bereits von der Plattform angeboten werden[4]. Darüber hinaus wirken sich zusätzliche Prüfungen negativ auf die Performance der Infrastruktur und den Verbrauch von Ressourcen aus. In dieser Arbeit wird daher vereinfachend davon ausgegangen, dass die syntaktische Definition eines Interfaces gleichzeitig das Verhalten der Funktionen beschreibt.

Eine ähnliche Vereinfachung kann auch für die Beschreibung von Synchronisationseigenschaften gemacht werden. Der tatsächliche Verlust durch diese Einschränkung hält sich hingegen in Grenzen, was zum größten Teil daran liegt, dass das Anwendungsmodell vereinfachend davon ausgeht, dass Komponenten, genauer gesagt Komponenteninstanzen, nicht von mehreren Nutzern gleichzeitig genutzt werden. Daher würde sich die Beschreibung von Synchronisationseigenschaften lediglich auf die exklusive Nutzung der Komponente beschränken. Dies würde letztlich bedeuten, dass ausschließlich die Synchronisation zwischen den verschiedenen Methoden einer Schnittstelle spezifiziert würde. Um die Entwicklung der Verteilungsinfrastruktur zu vereinfachen, wird diese Definition von Synchronisationseigenschaften einfach als Teil der Spezifikation syntaktischer Schnittstellen angesehen.

---

[3] Java Interfaces sind nicht a priori unveränderlich und eindeutig, was in realen Systemen zu Versionskonflikten führt. Da diese Versionskonflikte jedoch letztlich ein Problem der Programmiersprache sind, werden sie an dieser Stelle zunächst ignoriert, es zeigt sich jedoch, dass unterschiedliche Versionen über die Mechanismen zur Unterstützung nicht-funktionaler Parametern nachgebildet werden können.

[4] Programmiersprachen wie beispielsweise Eiffel unterstützen sowohl die Definition als auch die Prüfung von Verhaltensspezifikationen.

Die Unterstützung nicht-funktionaler Parameter soll hingegen nicht ausgegrenzt werden. Generell sollen möglichst viele nicht-funktionalen Parameter unterstützt werden, da auf Basis dieser Parameter kontextuelle Information eingebettet und die Auswahl von Komponenten optimiert werden kann. Beispielsweise könnte jede Komponente ihren Energieverbrauch spezifizieren. Ein energieoptimierender Selektionsalgorithmus zur Auflösung von Abhängigkeiten der Komponenten könnte dann auf Basis dieser Informationen die Komponenten mit dem niedrigsten Energieverbrauch binden. Damit liegt der Fokus der Beschreibungen klar auf syntaktischen und allgemeinen nicht-funktionalen Parametern.

Nachdem der Inhalt der Komponentenbeschreibungen geklärt ist, richtet sich der Blick auf die konkrete technische Umsetzung der Beschreibungen. Dabei stellt sich zunächst die Frage, ob für Beschreibung und Entwicklung der Komponenten dieselbe Sprache verwendet werden sollte.

Für den Einsatz von Java zur Beschreibung von Komponenten spricht die Tatsache, dass bei geschickter gewählten Vereinbarungen die Beschreibungen ohne Verarbeitung nach dem Laden der Komponente verwendet werden können. Darüber hinaus werden Programmierer durch die Nutzung einer Sprache zum Schreiben und Beschreiben von Komponenten nicht gezwungen, mehrere Sprachen gleichzeitig zu beherrschen. Weiterhin erlaubt der Einsatz von Java als Sprache zur Beschreibung von Komponenten die Nutzung selbst auf kleinen Geräten, die beispielsweise nicht über ein Dateisystem verfügen und daher keine externen Beschreibungen laden können.

Demgegenüber gibt es Gründe, die gegen die Nutzung von Java sprechen. Java ist eine kompilierte Sprache. Dementsprechend müssten auch Komponentenbeschreibungen vor der ersten Nutzung kompiliert werden. Während das auf dem Rechner eines Komponentenentwicklers sicherlich keine große Einschränkung darstellt, kann es bei einem Administrator zu einem Nachteil werden. Dies gilt vor allem im Bezug auf nicht-funktionale Parameter, die möglicherweise erst bei der Installation von Komponenten auf einem bestimmten Gerät bestimmt werden können. Darüber hinaus können die Vorteile einer direkten Nutzung von Java Code durch entsprechende Werkzeuge kompensiert werden. So könnte beispielsweise ein Übersetzer zwischen Beschreibungssprache und Code die direkte Nutzung von Beschreibungen auf kleinsten Geräten ermöglichen, ohne sich negativ auf die Flexibilität auszuwirken, falls Geräte mit großen Ressourcenbeständen zum Einsatz kommen. Falls darüber hinaus eine weit verbreitete Sprache zur Beschreibung von Komponenten verwendet wird, so relativiert sich auch der Vorteil, dass durch die einheitliche Nutzung von Java die Komponentenentwicklung vereinfacht wird.

Da sich die Vorteile der Nutzung von Java durch die Implementierung eines einfachen Übersetzers erreichen lassen, wird in dieser Arbeit eine weitere, interpretierte Sprache verwendet werden. Als Sprache zur Beschreibung von Komponenten dient XML [W3C00]. XML genießt eine hohe Verbreitung, was sich sowohl in der guten Editor- und Werkzeugunterstützung als auch in der Verfügbarkeit einer großen Zahl unterschiedlicher Parser ausdrückt. Zusätzlich ermöglichen die Parser in der Regel die automatische Prüfung der Struktur einer XML Datei auf Basis von Schemata, wodurch die Validierung von Beschreibungen einfacher wird.

Durch die Nutzung von XML zur Beschreibung von Komponenten wird die Syntax des Markups relativ beliebig. Im Folgenden sind daher vor allem die Bestandteile der Beschreibung wichtig und nicht die konkrete sprachliche Umsetzung. Die Bestandteile ergeben sich größtenteils aus der Struktur der Programmeinheiten und aus einigen technischen Details. Sie können insgesamt in die folgenden drei Klassen unterteilt werden.

- *Implementierung*; Dieser Teil der Komponentenbeschreibung beschreibt die Implementierung der Komponente. Dazu gehört beispielsweise der Name der Klasse, die die Komponentenimplementierung enthält oder ein frei wählbarer Name, der die Komponente auf dem lokalen System eindeutig identifiziert.
- *Angebot*; Dieser Teil der Komponentenbeschreibung definiert das Angebot der Komponente. Das Angebot einer Komponente besteht dabei aus der angebotenen Funktionalität in Form einer Reihe von Interfaces und den angebotenen Aufrufen zur Benachrichtigung von nutzenden Komponenten. Die Angebotsbeschreibung beinhaltet sowohl funktionale Parameter als auch nicht-funktionale.
- *Anforderung*; Dieser Teil der Komponentenbeschreibung definiert die Anforderungen der Komponente. Die Anforderungen können in Anforderungen bezüglich weiterer Komponenten und Anforderungen gegenüber der Plattform unterschieden werden. Die Anforderungen an weitere Komponenten setzten sich, analog zur Beschreibung des Angebots, aus einer Beschreibung der benötigten Funktionalität und der benötigten Benachrichtigungen zusammen. Wie die Angebotsbeschreibung setzt sich auch die Anforderungsbeschreibung aus funktionalen und nicht-funktionalen Parametern zusammen.

Um diese Klassen zu repräsentieren, wird in dieser Arbeit ein XML Dokument mit der folgenden Struktur genutzt. Zunächst gibt es, wie in jedem XML Dokument, einen Wurzelknoten (*Contract*). Dieser Wurzelknoten enthält genau drei Knoten. Der erste Knoten (*Implementation*) enthält die Implementierungsdetails. Der zweite Knoten (*Provision*) enthält das Angebot in Form einer

Reihe von Knoten, die angebotene Interfaces (*Interface*) oder Benachrichtigungen (*Event*) beschreiben. Der dritte Knoten (*Demand*) enthält ebenfalls eine Reihe von Konten. Dabei können zwei Knotentypen unterschieden werden. Der erste Kontentyp (*Component*) kann beliebig häufig vorkommen, da er jeweils eine Komponente beschreibt, die zur Ausführung benötigt wird. Die Anforderung an eine Komponente setzt sich dabei wiederum aus Anforderungen an die Funktionalität (*Interface*) und an Benachrichtigungsfunktionen (*Event*) zusammen. Vom zweiten Knotentyp (*Platform*) existiert maximal eine Instanz pro Komponentenbeschreibung. Sie enthält die Anforderungen bezüglich der Plattform.

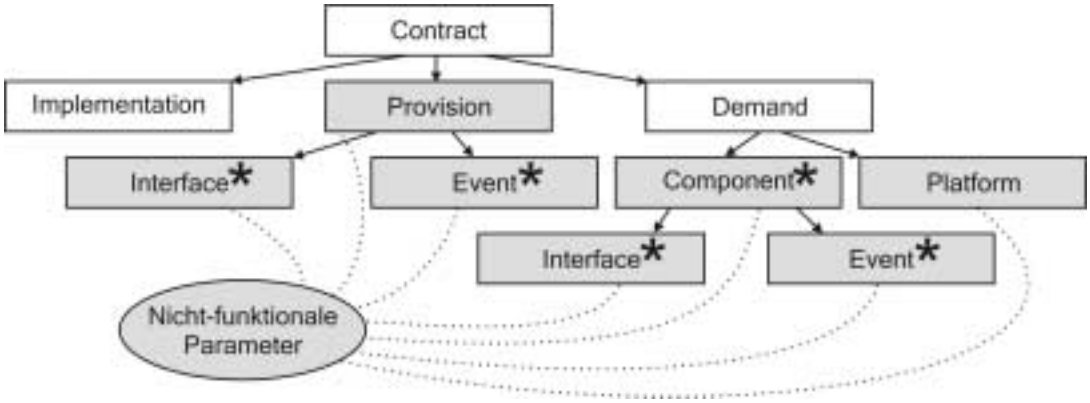

Abbildung 13 – Struktur von Beschreibungen

Diese Struktur ermöglicht die Definition aller syntaktischen Parameter, die sich aus den Programmeinheiten ergeben. Die Interfaces und Benachrichtigungen, sowohl auf Angebots- als auch auf Anforderungsseite, werden dazu in den Interface- und Eventknoten definiert. Weiterhin werden Implementierungsdetails der Komponente, wie zum Beispiel der Name der Klasse, die die angebotenen Methoden implementiert, spezifiziert werden. Die Einbettung nicht-funktionaler Parameter zur Beschreibung von Dienstgüte oder kontextueller Abhängigkeiten des Angebots und der Anforderungen, kann in den Interface-, Event-, Component- und Provisionknoten erfolgen. Nicht-funktionale Parameter, die sich auf die geforderten Eigenschaften der Plattform beziehen, können innerhalb des Plattformknotens definiert werden. Da es nicht Ziel dieser Arbeit ist, alle nicht-funktionalen Parametertypen zu klassifizieren, muss die statische Dokumentenstruktur erweiterbar sein. Dazu muss die Verteilungsinfrastruktur ermöglichen, dass Komponenten neue Knotentypen definieren und deren Verhalten festlegen.

Die Beschreibung einer Komponente besteht damit aus einem XML Dokument, dass eine vorgegebene Struktur zur vertraglichen Beschreibung der syntaktischen Merkmale der Komponente besitzt. Zusätzlich enthält dieses

Dokument Bereiche, die komponentenabhängige Erweiterungen ermöglichen. Insgesamt lassen sich Inhalt und Struktur des Dokuments aus der Struktur der Programmeinheiten ableiten, da die Beschreibung letztlich Angebot und Anforderungen einer Programmeinheit widerspiegelt.

5.2.4 Verhandlung

Die statische Struktur der vertraglichen Beschreibung syntaktischer Merkmale von Komponenten ermöglicht der Verteilungsinfrastruktur die automatische Zuordnung der Anforderungen einer Komponente zu passenden Angeboten anderer Komponenten. Durch die Vereinfachung, dass Interfaces im Bezug auf die Semantik der möglichen Implementierungen eindeutig sind, ergibt sich die Möglichkeit, auf Basis der syntaktischen Eigenschaften von Komponenten angeforderte und angebotene Funktionalität automatisch zu vergleichen, um passende Komponenten zu finden.

Der alleinige Vergleich syntaktischer Merkmale und der damit verbundene Abgleich von angebotener und benötigter Funktionalität ist jedoch im Hinblick auf die Dynamik nicht-funktionaler Parameter unzureichend. Im Gegensatz zur Syntax der Programmeinheiten, die in der Regel nach der Übersetzung des Quellcodes statisch ist[5], können nicht-funktionale Parameter im Allgemeinen nur dynamisch bestimmt werden.

Als Beispiel hierfür kann die Antwortzeit einer Komponente genannt werden. Die Antwortzeit hängt von einer Vielzahl von Parametern ab. Dazu gehört beispielsweise die verfügbare Rechenleistung des ausführenden Geräts zum Zeitpunkt des Aufrufs, und im Falle der entfernten Nutzung auch die Eigenschaften der Verbindung.

Dynamische Parameter müssen demnach zur Laufzeit bestimmt werden. Da in dem beschriebenen Anwendungsmodell nicht-funktionale Eigenschaften zu den Anforderungen von Komponenten gehören können, müssen zum automatischen Abgleich von Angebot und Anforderung die Werte der nicht-funktionalen Parameter bestimmt werden. Die Bestimmung dieser Werte hängt zwar von den Eigenschaften der Umgebung ab, ist letztlich jedoch anwendungsspezifisch.

In Verbindung mit der Umsetzung der Programmeinheiten ergibt sich dadurch ein technisches Problem. Folgt man dem Ansatz, dass Komponenten zur Nutzung instanziert werden müssen, führt das dazu, dass Ressourcen unter Umständen unnötig belegt werden. Dies begründet sich darin, dass erst

[5] Die Generierung von Code zur Laufzeit würde es ermöglichen, syntaktische Schnittstellen zur Laufzeit zu verändern. Die Adaption syntaktischer Schnittstellen wird in dieser Arbeit jedoch nicht weiter betrachtet.

nachdem eine Komponenteninstanz die dynamischen Werte der nicht-funktionalen Parameter bestimmt hat, ein Abgleich dieser Parameter erfolgen kann. Daher kann es vorkommen, dass direkt nach der Bildung einer Komponenteninstanz festgestellt wird, dass die Komponenteninstanz nicht die geforderten Eigenschaften liefert. In solchen Fällen müsste sie wieder entladen werden. Für den Zeitraum des Abgleichs würde die Komponenteninstanz daher unnötigerweise Ressourcen belegen.

Da die Komponentenbeschreibung im Bezug auf die nicht-funktionalen Parameter erweiterbar ist, ergibt sich neben der ungünstigen Ressourcennutzung ein weiteres Problem. Während die Charakteristika der syntaktischen Eigenschaften von Komponenten, im Hinblick auf den Abgleich von Funktionalität, von der Programmiersprache vorgegeben sind, können neue nicht-funktionale Parameter, neue Mechanismen zum Abgleich erfordern.

Konkret bedeutet das, dass sich die Eigenschaften für den Abgleich von syntaktischen Schnittstellen durch das Typsystem der Programmiersprache ergeben. Der Einsatz von Java Interfaces zur Beschreibung von Komponentenschnittstellen führt dazu, dass ein angefordertes Interfaces ausschließlich durch Komponenten geboten werden kann, die dieses Interface oder eines, das das geforderte Interface erweitert, bereitstellen.

Da die nicht-funktionalen Parameter erweiterbar sind, können solche Aussagen nicht für die Gesamtheit aller möglichen, nicht-funktionalen Parameter gemacht werden. Beispielsweise sind im Bezug auf den Parameter „Antwortzeit“ niedrigere Werte wünschenswert, für Parameter wie „Bandbreite“ oder „verfügbarer Speicher“ sind jedoch höhere Werte besser. Daher ist es erforderlich, dass Komponenten die Möglichkeit gegeben wird, den Abgleich von Funktionalität über geeignete Schnittstellen zu erweitern.

Die Notwendigkeit einer komponentenabhängigen Bestimmung von Parametern und der Wunsch nach einer möglichst guten Nutzung von Ressourcen führen zum Konzept der Stellvertreterobjekte für Komponenten. Diese Stellvertreterobjekte werden im weiteren Verlauf der Arbeit Factories genannt. Die Factory einer Komponente wird von der Komponente in Form einer Klasse mitgeliefert, die neben der Aushandlung und Bestimmung von nicht-funktionalen Parametern der Komponenteninstanzen für ihre Erzeugung zuständig ist. Da Factories ansonsten keine weiteren Funktionen erfüllen, kann davon ausgegangen werden, dass Factories relativ wenige Ressourcen benötigen. Aus diesem Grund müssen Factories nicht bedarfsgesteuert instanziert werden. Statt dessen kann beim Start der Infrastruktur für jede Komponente eine Instanz ihrer Factory erzeugt werden, womit sich nun die Frage stellt, wie die Funktionen der Factory für die Bestimmung von Parametern gestaltet sein müssen.

Grundsätzlich können die Werte nicht-funktionaler Parameter des Angebots einer Komponente von allen spezifizierten Abhängigkeiten abhängen. Diese Abhängigkeiten setzten sich aus Anforderungen bezüglich der Plattform, d.h. der Verteilungsinfrastruktur, und Anforderungen bezüglich benötigter Komponenten zusammen. Um eine genaue Bestimmung der Parameter zu ermöglichen, müssten die Abhängigkeiten der Komponente aufgelöst werden, wobei alle benötigten Komponenten bereitgestellt werden müssten.

Abbildung 14 – Ineffiziente Bestimmung von Parametern

Mit Blick auf das Anwendungsmodell bedeutet das, dass um die Parameter einer Komponente zu bestimmen, ein Wald von Komponenten rekursiv instanziert werden muss. Da die Instanzen Ressourcen benötigen, ist dieses Vorgehen äußerst ineffizient, denn nach der Bestimmung der Parameter ist es nicht auszuschließen, dass die Komponente nicht instanziert wird. Damit wäre eine potentiell große Zahl überflüssiger Komponenteninstanzen gebildet worden.

Um diese Ineffizienz zu vermeiden, kann die Bestimmung nicht-funktionaler Parameter von der Bildung von Komponenteninstanzen getrennt werden. Die Factory muss dabei die Möglichkeit bieten, Parameter allein auf Basis verfügbarer Komponentenbeschreibungen zu bestimmen. Da diese Bestimmung der Parameter möglicherweise ungenau ist und sich in der Zeit zwischen Bestimmung und Instanzierung die Verfügbarkeit der Komponenten geändert haben kann, muss sie bei der tatsächlichen Instanzierung wiederholt werden.

Die automatische Auflösung von Abhängigkeiten kann demnach in zwei Phasen unterteilt werden. Soll eine Komponenteninstanz gebildet werden, werden in der ersten Phase zunächst die nicht-funktionalen Parameter der Komponenten bestimmt, die über die benötigten Funktionalitäten verfügen. In dieser Phase werden keine Komponenteninstanzen gebildet und die Bestimmung der Parameter geschieht einzig auf Basis der Factory. Nachdem die Parameter der möglichen Kandidaten feststehen, wird ein Kandidat von der Verteilungsinfrastruktur ausgewählt und es wird eine Instanz gebildet. Dabei werden die Parameter nochmals bestimmt. Kann der ausgewählte Kandidat

aufgrund geänderter Verfügbarkeiten nicht instanziert werden, wird von der Verteilungsinfrastruktur ein anderer Kandidat ausgewählt. Falls keiner der Kandidaten verfügbar ist, wird die Instanzierung abgebrochen, da die Abhängigkeiten der Komponenteninstanz nicht aufgelöst werden können und damit die korrekte Ausführung der Komponenteninstanz nicht gewährleistet werden kann.

Die Bestimmung und der Abgleich von Anforderung und Angebot ist ein komplexer Vorgang. Durch die gewünschte Erweiterbarkeit von Parametern muss die Verteilungsinfrastruktur Mechanismen vorsehen, um den Abgleich und die Bestimmung erweiterbar zu halten. Da die Bestimmung von Parametern einer Komponente von allen benötigten Komponenten und der Plattform abhängen kann, ist es aus Gründen der Performance äußerst sinnvoll, ein Stellvertreterobjekt einzuführen, das die Bestimmung von Parametern ohne die Instanzierung von Komponenten erlaubt.

5.2.5 Adaption

Aufgrund der Mobilität und Dynamik ubiquitärer Rechnersysteme ist die einmalige Bestimmung nicht-funktionaler Parameter nicht ausreichend. Während der Ausführung einer Anwendung können sich die Eigenschaften von Verbindungen zu entfernten Rechnern ändern und durch neue Technik können sich selbst die Eigenschaften eines einzelnen Geräts ändern[6].

Ändert sich die Umgebung einer ausgeführten Komponenteninstanz, kann das Einfluss auf ihr Angebot haben. Damit der Komponenteninstanz, die vom Angebot abhängt, die Möglichkeit gegeben wird, sich auf das neue Angebot einzustellen, muss sie zunächst über die neue Situation informiert werden. Die Verteilungsinfrastruktur muss daher Mechanismen bieten, um Angebotsänderungen einer Komponenteninstanz zu propagieren.

Damit die Verteilungsinfrastruktur in die Lage versetzt wird, solche Änderungen zu erkennen, muss jede Komponenteninstanz ihr Angebot der Verteilungsinfrastruktur zugänglich machen. Auf dieser Basis können Änderungen dann automatisch an die nutzende Komponenteninstanz weitergeleitet werden. Wie bei allen Kommunikationen von der genutzten zur nutzenden Komponenteninstanz, darf auch hier die genutzte Komponenteninstanz nicht auf den Antworten der nutzenden beruhen. Es könnte schließlich auch Komponenteinstanzen geben, die generell nicht auf Angebotsänderungen reagieren.

[6] Beispielsweise erlaubt die Intel SpeedStep Technik die Reduktion des Energieverbrauchs zu Lasten der Taktfrequenz eines Prozessors.

Um Entwickler einfacher Komponenteninstanzen nicht zu zwingen, komplexe Benachrichtigungslogik zu implementieren, empfiehlt sich ein eventbasierter Mechanismus, mit dem komplexere Komponenteninstanzen ihr Interesse an Angebotsänderungen signalisieren können und den einfachere ignorieren können. Auf Seite der nutzenden Komponente muss die Verteilungsinfrastruktur damit Mechanismen anbieten, mit denen sich die Komponente für Nachrichten über Angebotsänderungen registrieren kann.

Da Änderungen relativ häufig auftreten können und mit der Dynamik des Systems zunehmen, sollten sie relativ klein gehalten werden. Dazu könnte die Verteilungsinfrastruktur entweder das Delta der Änderung versenden oder lediglich eine Nachricht über das Auftreten einer Änderung. Die zweite Variante erfordert allerdings, dass nutzende Komponenteninstanzen zusätzlich die Möglichkeit erhalten, das aktuelle Angebot genutzter Komponenteninstanzen abzurufen. Welche der beiden Möglichkeiten effizienter ist, hängt letztlich von den Komponenten ab. Ist die Zahl der Komponenten, die Angebotsänderungen verfolgen, groß, ist die Deltaberechnung effizienter. Ist sie hingegen klein, werden viele unnötigen Deltas bestimmt. In dieser Arbeit werden zunächst keine Deltas berechnet. Sobald ein genaueres Bild der Anwendungen und damit der Komponenten für ubiquitäre Rechnersysteme feststeht, muss diese Entscheidung erneut betrachtet werden.

Durch den Mechanismus zur Benachrichtigung von Angebotsänderungen bietet die Verteilungsinfrastruktur eine optimale Schnittstelle, um kontextuelle Informationen zu integrieren. Mit Hilfe der erweiterbaren Struktur der Komponentenbeschreibungen und der Möglichkeit, die Beschreibung auf Ebene von Komponenteninstanzen anzupassen und zu propagieren, können beliebige kontextuelle Informationen integriert werden. Diese können dann jeweils mit den nutzenden Komponenteninstanzen geteilt werden.

Die Kommunikation kontextueller Information von den Blättern eines Anwendungsbaums hin zur Wurzel entspricht genau der Definition von Softwarekomponenten, da eine Komponente nur von kontextuellen Informationen der benötigten Softwarekomponenten abhängt. Damit spezifiziert jede Komponente über ihre vertragliche Beschreibung auch alle kontextuellen Abhängigkeiten explizit.

Ein Benachrichtigungsmechanismus für Angebotsänderungen ist zwar notwendig, aber nicht hinreichend, um auf die Dynamik ubiquitärer Rechnersysteme einzugehen. Zusätzlich muss es Komponenteninstanzen ermöglicht werden, aufgrund neuer Situationen Einfluss auf die Anwendung zu nehmen. Dabei sollten Komponenteninstanzen in zweierlei Hinsicht eingreifen können. Zum einen müssen Komponenteninstanzen in der Lage sein, der Verteilungsinfrastruktur mitzuteilen, dass eine genutzte Komponenteninstanz, beispielsweise aufgrund von Angebotsänderungen, nicht mehr den geforderten Eigen-

schaften entspricht. Darüber hinaus sollte eine Komponenteninstanz entscheiden können, wann sie ihr Angebot nicht länger erbringen kann und demnach gestoppt werden sollte.

Die Verteilungsinfrastruktur muss daher Schnittstellen bieten, mit denen Komponenteninstanzen gewünschte Änderungen in ihrem eigenen Lebenszyklus bekannt machen können. Minimal muss ein Zustandsübergang von „Instanz gestartet" nach „Instanz gestoppt" ausgelöst werden können. Damit kann eine Komponenteninstanz auf Wunsch ihre Ausführung einstellen. Nutzende Komponenten benötigen dementsprechend einen Mechanismus, mit dem sie die genutzte Komponente austauschen können. Minimal sollte die Verteilungsinfrastruktur Funktionen zum erneuten Auflösen der Abhängigkeiten anbieten. Über diesen Mechanismus kann eine Komponenteninstanz ihre genutzten Komponenteninstanzen nach Bedarf austauschen. In dieser Arbeit werden Komponenteninstanzen zusätzlich mit der Möglichkeit ausgestattet, die Bindung zu einer genutzten Komponenteninstanz zu einem beliebigen Zeitpunkt aufzulösen. Durch die frühe Aufgabe von Bindungen kann der Ressourcenverbrauch reduziert werden, da die Verteilungsinfrastruktur weniger Bindungen aufrechterhalten muss.

Für Adaption von Anwendungen zur Laufzeit sind demnach zwei Mechanismen notwendig. Zum einen müssen Softwarekomponenten über Zustandsänderungen genutzter Komponenten auf dem Laufenden gehalten werden und zum anderen müssen sie auf solche Änderungen reagieren können. Direkte Einflussnahme auf Anwendung ist dabei nur an der eigenen Komponente und in Richtung genutzter Komponenten wünschenswert.

5.3 Erweiterungen

Wie bereits erwähnt, soll diese Arbeit als Grundlage für spätere Erweiterungen dienen. Ein Fokus der Erweiterungen stellt die Anbindung architekturbasierter Adaptionsstrategien dar. Betrachtet man Systeme wie Aura, stellt man fest, dass dazu zwei wichtige Mechanismen von einer zugrunde liegenden Infrastruktur bereitgestellt werden müssen.

Zum einen muss eine komponentenbasierte Verteilungsinfrastruktur in der Lage sein, Auskünfte über den aktuellen Systemzustand zu liefern und zum anderen muss sie vielseitige Möglichkeiten bieten diesen in gewünschter Weise, abhängig vom architektonischen Modell, zu ändern.

Der vermutlich wichtigste Teil des Systemzustands, den die Verteilungsinfrastruktur selbst verwaltet, sind die Bindungen zwischen Komponenteninstanzen. Damit für die spätere architekturbasierte Erweiterung, der Zugriff auf Bindungen möglichst einfach wird, sollten diese an einer zentralen Stelle der

lokalen Verteilungsinfrastruktur gespeichert werden. Daraus ergibt sich die Notwendigkeit eines lokalen Repositories für aktuelle Bindungen.

Ein idealer Punkt für zur Einflussnahme auf Bindungen ist der Algorithmus zur Komponentenselektion, der ausgeführt wird, wenn die Abhängigkeiten einer Komponenteninstanz aufgelöst werden müssen. An dieser Stelle könnte die Auswahl von Komponenten zur Instanzierung auf Basis architektonischer Modelle optimiert werden. Damit der Algorithmus zu einem späteren Zeitpunkt einfach erweitert werden kann, sollte er ebenfalls zentral verwendet werden. Damit wird es notwendig, die Selektion von Komponenteninstanzen von der restlichen Funktionalität der Verteilungsinfrastruktur zu trennen.

Die technische Umsetzung der Verteilungsinfrastruktur wird diesen beiden Punkten sorge tragen, obwohl sie derzeit nicht zwingend erforderlich sind. Das Repository zur Verwaltung von Bindungen und der erweiterbare Selektionsalgorithmus liefern damit weitere Anforderungen an die Umsetzung, die sich nicht unmittelbar aus dem Komponentenmodell ergeben.

6 Implementierung

Nachdem im vorhergehenden Kapitel die wesentlichen Konzepte einer komponentenbasierten Verteilungsinfrastruktur für ubiquitäre Rechnersysteme vorgestellt wurden, wird in diesem Kapitel auf eine mögliche Implementierung eingegangen. Diese wird im Folgenden abkürzend PCom (Pervasive Components) genannt.

Um nicht den Rahmen dieser Arbeit zu sprengen, werden die wesentlichen Teile nur kurz umrissen. Dazu gehört zunächst die Einbettung in die bestehende Infrastruktur und das daraus resultierende Schichtenmodell mit seinen Schnittstellen. Im Anschluss daran wird auf die statische Struktur und die Aufteilung der Funktionalitäten eingegangen. Abschließend wird das dynamische Zusammenspiel der statischen Elemente zur Laufzeit vorgestellt, wobei es sich ebenfalls nur um einen kleinen Ausschnitt handelt.

6.1 Schichtenmodell

Bei der Erweiterung von bestehenden Infrastrukturen gibt es verschiedene Vorgehensweisen. Dies gilt vor allem für solche, bei denen, wie im Falle von BASE, der Quellcode verfügbar ist. Zum einen kann die Erweiterung ohne Eingriffe in die bestehende Infrastruktur vorgenommen werden. Zum anderen können notwendige Teile aus der Infrastruktur extrahiert und in die Erweiterung eingebettet werden.

Die Vor- und Nachteile liegen dabei auf der Hand. Während die Extraktion notwendiger Funktionalität das Potential hat, bessere Performance bei kleinerem Umfang zu liefern, führt sie gleichermaßen dazu, dass die Erweiterung nicht automatisch von zukünftigen Verbesserungen der Infrastruktur profitieren kann.

Um die Vorteile beider Vorgehensweisen möglichst weitgehend zu kombinieren, wurde bei der Integration in BASE der Mittelweg gewählt. Zunächst diente eine vollständige und unveränderte Version von BASE als Grundlage. Im Zuge der Entwicklung wurde BASE an zwei Stellen erweitert. Aus Performancegründen wurde diese Erweiterung über Vererbung (Whitebox Wiederverwendung) und nicht mittels Komposition (Blackbox Wiederverwendung) realisiert. Dadurch ist PCom nicht vollständig unabhängig von den Implementierungsdetails von BASE. Allerdings beschränken sich die Eingriffe auf einen relativ kleinen, überschaubaren Teil der Verteilungsinfrastruktur, nämlich Proxies, Skeletons und Registry. Folglich ist sichergestellt, dass zukünftige Verbesserungen an BASE nur geringfügige Anpassungen in PCom nach sich ziehen.

Aus den vorangegangenen Entscheidungen ergibt sich folgendes Schichtenmodell des Gesamtsystems:

Die Plattform liefert geräteabhängige Abstraktionen. Diese werden durch die Laufzeitumgebung der Java 2 Micro Edition und den BASE Plug-in Mechanismus in plattformunabhängige Java Programmierschnittstellen transformiert.

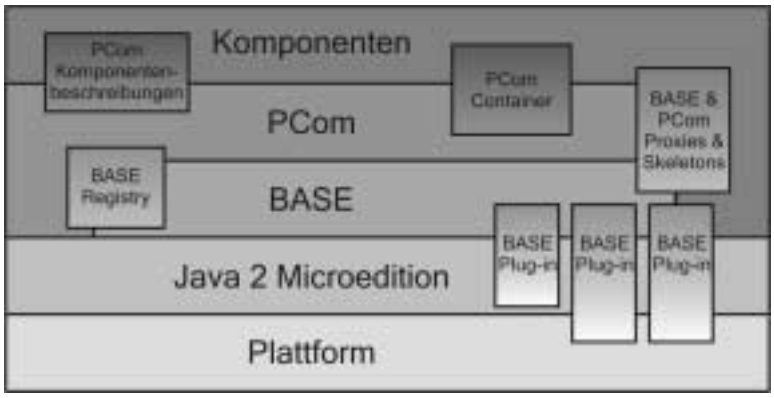

Abbildung 15 – Schichtenmodell

Die Komponenten der Anwendungen werden mit Hilfe ihrer Beschreibungen integriert. Die gesamte Verwaltung übernimmt dabei der so genannte PCom Container. Er bedient sich einer erweiterten BASE Registry und sorgt dafür, dass Komponenteninstanzen über die modifizierten BASE Proxies und Skeletons kommunizieren können.

6.2 Statisches Modell

Das vorgestellte Schichtenmodell liefert eine statische Darstellung des Gesamtsystems, die im Folgenden vor allem mit Blick auf PCom weiter detailliert wird. Da die Implementierung von PCom (ohne Entwicklungswerkzeuge) rund 60 Klassen und Interfaces umfasst, ist eine übersichtliche, detaillierte Darstellung aller Funktionen schwierig. Daher werden lediglich Ausschnitte vorgestellt.

Zunächst wird geklärt, wie Komponenten und ihre Instanzen umgesetzt werden. Dabei wird vor allem auf die Struktur der Programmeinheiten eingegangen, die im vorhergehenden Kapitel erarbeitet wurde. Daraufhin wird die Umsetzung von Factories, den Stellvertreterobjekten zur Instanzierung von Komponenten, eingegangen, wobei die Anforderungen an Entwickler in den Vordergrund gestellt werden. Im Anschluss daran wird die Aufteilung von Funktionen des Containers vorgestellt. Zuletzt wird beschrieben, wie die XML-basierten Komponentenbeschreibungen abgebildet werden.

6.2.1 Instance

Aus der Struktur von Programmeinheiten ergeben sich, wie bereits erwähnt, vier Gruppen von Methoden, die eine Komponente bereitstellen muss. Zunächst muss die Klasse, die die Schnittstelle einer Komponente bereitstellt alle Methoden ihres Angebots implementieren. Weiterhin muss sie über Methoden verfügen, mit denen PCom die Komponente mit Proxies für benötigte Komponenteninstanzen versorgen kann. Falls die Komponente auch Upcalls unterstützt, muss sie mit einem Proxy für die nutzende Komponente versorgt werden. Dieser wird im folgenden EventProcessor genannt. Darüber hinaus muss eine Komponente weitere Methoden zur Verwaltung implementieren.

Da die ersten drei Gruppen von Methoden allein vom Angebot und den Anforderungen einer Komponente abhängen, steht die genaue Zusammensetzung dieser Methoden erst zu ihrer Entwicklungszeit fest. Um die Typsicherheit von Java nicht aufzubrechen, wurde im Zuge von PCom eine Reihe von Codegeneratoren erzeugt, die dafür sorgen, dass Komponenten die geforderten Methoden liefern. Diese Generatoren müssen zwar nicht zwingend genutzt werden, doch sie vereinfachen die Entwicklung von Komponenten, da sie von einer Reihe von Implementierungsdetails abstrahieren. Grundlage für die Generatoren sind die syntaktischen Beschreibungen der Komponente, die unabhängig von der Nutzung der Generatoren vorliegen müssen.

Die einzige Gruppe von Methoden, die statisch bekannt ist, sind die Methoden zur Verwaltung von Komponenteninstanzen. Sie wird in einem Interface namens *Instance* definiert und setzt sich einerseits aus den Methoden zur Verwaltung des Lebenszyklus zusammen. Andererseits enthält sie eine Methode, die den Zugriff auf einen so genannten *InstanceController* ermöglicht. Jede Komponente muss eine Klasse bereitstellen, die das Interface *Instance* unterstützt.

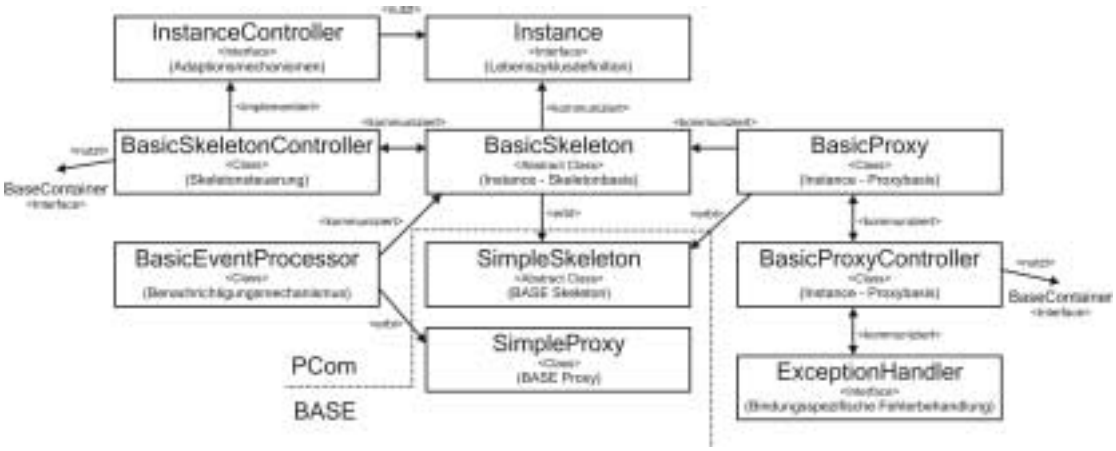

Abbildung 16 – Umsetzung von Komponenteninstanzen

Der *InstanceController* erlaubt einer Komponenteninstanz, die an sie gerichteten Anforderungen abzurufen und gleichzeitig ihr Angebot anzupassen. Die Anpassung von Anforderungen und Angebot werden jeweils auf der Seite der genutzten Komponente bzw. der nutzenden Komponente signalisiert. Darüber hinaus kann mittels des *InstanceControllers* die ausgeführte Komponenteninstanz kontrolliert heruntergefahren werden.

Für jede Komponenteninstanz, die zur Ausführung einer anderen benötigt wird, verfügt die nutzende über eine Methode, mit der ihr ein Proxy für die benötigte Komponenteninstanz zugewiesen wird. Da diese Proxies Stellvertreter für anwendungsspezifische Komponentenschnittstellen sind, müssen auch sie während der Entwicklung von Komponenten mit Hilfe eines Generators erstellt werden. Die Proxies erben dabei immer von der Klasse *BasicProxy*, die wiederum von der in BASE definierten Klasse *SimpleProxy* erbt. Anders als der *SimpleProxy* verfügt ein *BasicProxy* über einen so genannten *ProxyController*.

Über den *ProxyController* kann die nutzende Komponenteninstanz die Eigenschaften der Bindung ändern. Sie kann eine neue Bindung anfordern oder die Bindung aufheben. Weiterhin kann sie die Anforderungen an die Bindung, d.h. an die genutzte Komponenteninstanz, ändern und Änderungen des Angebots der genutzten Komponenteninstanz abrufen. Der *ProxyController* verfügt zusätzlich über einen *ExceptionHandler*, der es ermöglicht, Kommunikationsfehler bindungsspezifisch zu behandeln.

Komponenteninstanzen, die Aufrufe in Richtung nutzende Komponenteninstanz durchführen, verfügen zusätzlich über einen *EventProcessor*. Beim *EventProcessor* handelt es sich ebenfalls um eine Erweiterung der Klasse *SimpleProxy*, die von BASE bereitgestellt wird. Die Aufrufe beider Arten von Proxies werden an die jeweiligen Skeletons der genutzten bzw. der nutzenden Komponenteninstanzen weitergeleitet.

In PCom besitzen alle Komponenten eigene Skeletons. Wie Proxies sind Skeletons komponentenspezifisch und werden daher aus der Komponentenbeschreibung generiert. Alle generierten Skeletons erben von der abstrakten Klasse *BasicSkeleton*, das auf dem *SimpleSkeleton* aufbaut. Die wichtigste Aufgabe der Skeletons ist die typsichere Weiterleitung von Aufrufen. Darüber hinaus verfügen alle *BasicSkeletons* über einen *SkeletonController*.

Der *SkeletonController* liefert die Funktionalität des InstanceControllers und ist in der Lage die Ausführung einer verbundenen Instance bei Bedarf zu beenden. Darüber hinaus ist der *SkeletonController* Teil des Signalisierungsmechanismus für Anforderung- und Angebotsänderungen.

Zusammenfassend kann festgehalten werden, dass PCom zur Anbindung von anwendungsspezifischen Schnittstellen, auf Codegenerierung zurückgreift. Wie in BASE liefern die Basisklassen der Proxies und Skeletons die gemeinsame genutzte Funktionalität. Der wesentliche Unterschied zwischen BASE und PCom liegt in den Controllern, mit denen PCom Komponenteninstanzen ihre Eigenschaften und die Eigenschaften einer Bindung zur Laufzeit abrufen und ändern können.

6.2.2 Factory

Während bei Komponenteninstanzen die Generierung von Code unumgänglich ist, da die Schnittstellen der Anwendung nicht a priori bekannt sein können, ist sie im Bezug auf Factories nicht notwendig. Die Aufgaben einer Factory beschränken sich auf die Aushandlung von nicht-funktionalen Parametern und die Instanzierung von Komponenten. Die Schnittstelle von Factories ist daher im Interface *Factory* fest definiert.

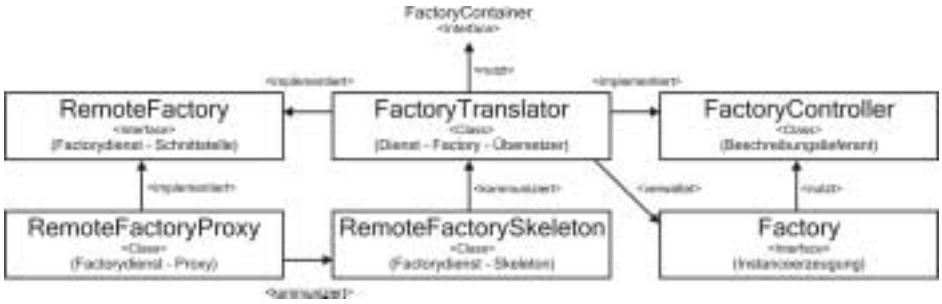

Abbildung 17 – Umsetzung von Factories

Da die Funktionalität der Factories auch von entfernten Geräten genutzt werden muss, sind sie als BASE Dienste umgesetzt. Damit die Komplexität der Entwicklung von Factories nicht unnötig durch rechnerübergreifende Kommunikation erhöht wird, setzt PCom einen so genannten *FactoryTranslator* ein. Über eine Schnittstelle des PCom Containers sorgt der *FactoryTranslator* dafür, dass alle notwendigen Operationen veranlasst werden, sobald Parameter ausgehandelt oder Instanzen erzeugt werden.

Der *FactoryTranslator* liefert Factories die Komponentenbeschreibung über das Interface *FactoryController*. Durch den *FactoryController* können Factories und Komponenten plattformunabhängig auf ihre Beschreibungen zugreifen. Plattformabhängige Lademechanismen wandern also von der Anwendung in das Komponentensystem.

Damit der *FactoryTranslator* und damit die Factory auch entfernt genutzt werden kann, muss er eine Schnittstelle implementieren und einen Proxy und ein

Skeleton bereitstellen. Das Interface *RemoteFactory* und die Klassen *RemoteFactoryProxy* und *RemoteFactorySkeleton* dienen diesem Zweck.

Durch das Zusammenspiel der vorgestellten Klassen und Interfaces vereinfacht PCom die Entwicklung von Factories erheblich. Anstatt aufwendige Schnittstellen zu bedienen und entfernte Kommunikation unterstützen zu müssen, implementieren Factories letztlich nur ein Interface, dass lokale Objekte erzeugt und Beschreibungen abgleicht, die zur Laufzeit geliefert werden.

6.2.3 Container

Der Container ist der zentrale Teil von PCom, da er die gesamte Verwaltung von Komponenten übernimmt. Da der Container als zentrales Element von nahezu allen Teilen des Systems genutzt wird, sind die unterschiedlichen Rollen des Containers durch Interfaces getrennt. Darüber hinaus wurde eine Reihe von Funktionen des Containers in untergeordnete Klassen zusammengezogen.

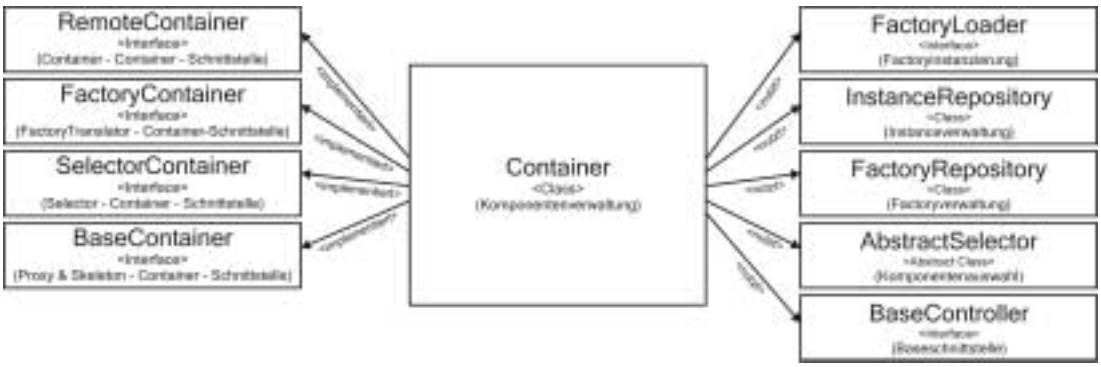

Abbildung 18 – Umsetzung des Containers

Wie Factories sind auch Container gewöhnliche BASE Dienste. Im Bezug auf andere, entfernte Dienste oder Container bieten sie ihre Funktionalität durch das Interface *RemoteContainer* an. Über diese Schnittstelle können Factories abgerufen werden, deren Komponenteninstanzen syntaktisch kompatibel mit bestimmten Anforderungen sind. Weiterhin können die verfügbaren Factories und deren Komponentenbeschreibungen abgerufen werden.

Neben der Schnittstelle zum entfernten Zugriff verfügen Container über eine Schnittstelle zum Abrufen aller entfernten Container. Dieses so genannte *SelectorContainer* Interface wird während der Selektion von Komponenten benötigt.

Das *FactoryContainer* Interface liefert Factories Methoden zur Registrierung neuer Instanzen und zur Unterstützung bei der Aushandlung von Parametern.

Methoden die von den Proxies und Skeletons der Komponenten benötigt werden, werden schließlich über das *BaseContainer* Interface angeboten. Über dieses Interface können Proxies neu gebunden und Komponenteninstanzen gestoppt werden.

Factories und Komponentenbeschreibungen werden nicht im *Container* selbst geladen. Diese plattformabhängige Aufgabe wird von einem *FactoryLoader* übernommen. Derzeit verfügt PCom über die Möglichkeit, Komponentenbeschreibungen aus XML Dokumenten und aus Java Klassen zu laden. Die Codegeneratoren von PCom bieten die Möglichkeit, XML Dokumente in solche Klassen zu überführen. Daher ist ein XML Parser zur Laufzeit nicht zwingend notwendig.

Die eigentliche Verwaltung von Instanzen und Factories findet jeweils in dem dafür vorgesehenen Repository statt. Während das *FactoryRepository* lediglich eine einfache Datenstruktur ist, erledigt das *InstanceRepository* darüber hinaus noch weitere Aufgaben. Gemeinsam mit den Proxies und Skeletons der Komponenteninstanzen liefert es einen Lease-Mechanismus, der es erlaubt, Ressourcen auf Seite der genutzten Komponenteninstanz nach Verbindungsabbrüchen freizugeben.

Mit welchen Komponenten Abhängigkeiten anderer Komponenten erfüllt werden, wird im *AbstractSelector* entschieden. Durch Externalisierung des Selektionsalgorithmus kann dieser, ohne Eingriffe in den Code des Containers, abhängig von der jeweiligen Strategie, verändert werden. Damit der Selektionsalgorithmus alle notwendigen Informationen abrufen kann, greift er auf Teile der Funktionalität des Containers mit Hilfe des *SelectorContainer* Interface zu.

Auf alle Funktionen, die vom Kommunikationsmechanismus bereitgestellt werden müssen, wird über das Interface *BaseConnector* zugegriffen. Vorrangig wird der *BaseConnector* dazu verwendet, um Skeletons lokaler Komponenteninstanzen zu verwalten und um entfernte Container zu entdecken. Die Implementierung dieser Schnittstelle übernimmt daher eine erweiterte Version der BASE Registry.

6.2.4 Contract

Neben dem Aufbau der aktiven Teile von PCom ist auch die Umsetzung der Komponentenbeschreibungen von zentraler Bedeutung. Zur Laufzeit werden die XML basierten Beschreibungen nicht als XML Document Object Model repräsentiert, sondern das XML Dokument wird beim Start der Verteilungsinfrastruktur in einen Objektgraph mitgelieferter PCom Klassen überführt.

Durch diese Transformation der Komponentenbeschreibung in ein vordefiniertes Objektmodell kann vollständig von der Form der Beschreibung abs-

trahiert werden. Das hat den Vorteil, dass PCom grundsätzlich nicht mit XML Dokumenten operieren muss. Damit PCom selbst auf kleinsten Geräten lauffähig ist, wurde im Zuge dieser Arbeit ein Codegenerator entwickelt, der in der Lage ist, XML basierte Komponentenbeschreibungen in Klassen zu überführen, mit denen entsprechende Objektgraphen erzeugt werden können.

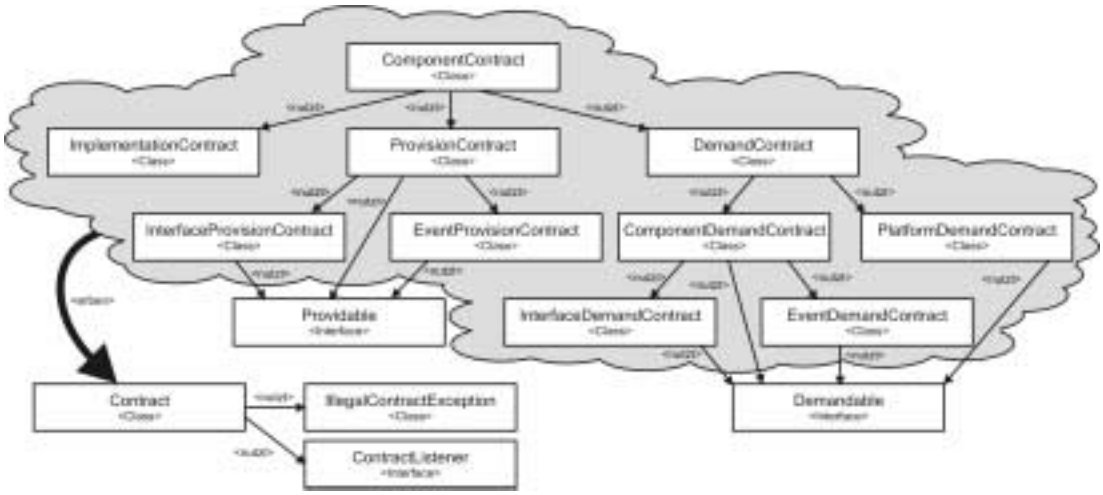

Abbildung 19 – Umsetzung von Beschreibungen

Als Basisklasse für alle Teile einer Komponentenbeschreibung dient die Klasse *Contract*. Fehler innerhalb der Komponentenbeschreibungen können mit Hilfe der *IllegalContractException* signalisiert werden. Änderungen werden mit dem *ContractListener* propagiert.

Contract definiert das generelle Verhalten von Beschreibungsteilen. Verhalten, das nur für einen bestimmten Teil des Dokuments wichtig ist, wird in der jeweiligen Subklasse hinzugefügt. Die Struktur der Subklassen entspricht der Struktur des XML Dokuments. Beispielsweise wird der *Contract*-Knoten einer Komponentenbeschreibung in ein Objekt vom Typ *ComponentContract* überführt und *Implementation*-Knoten werden durch Objekte der Klasse *ImplementationContract* abgebildet.

Nicht-funktionale Parameter werden durch das *Providable* und das *Demandable* Interface abgebildet. Durch diese Trennung von Schnittstelle und Implementierung über Java Interfaces können nicht-funktionale Parameter beliebig erweitert werden. Klassen, die das *Providable* Interface implementieren, müssen lediglich dafür sorgen, dass sie ein bestimmtes Angebot beschreiben können, während Klassen, die das Interface *Demandable* implementieren, dafür Sorge tragen müssen, dass sie ihre Anforderungen mit dem Angebot eines Objekts vom Typ *Providable* vergleichen können. Für PCom selbst ist letztlich einzig und allein wichtig, ob eine Anforderung von einem Angebot erfüllt wird oder nicht.

6.3 Dynamisches Modell

Nachdem die statische Struktur von PCom vorgestellt wurde, muss nun geklärt werden, wie diese ihre Funktionalität zur Laufzeit liefert. Wie die Vorstellung der statischen Struktur beschränkt sich auch dieser Teil der Arbeit lediglich auf einen kurzen Überblick über die wichtigsten Bereiche.

Zunächst wird daher erläutert, wie PCom und BASE interagieren, um Aufrufe von einer Komponenteninstanz zur anderen weiterzuleiten. Im Anschluss daran werden die Abläufe bei Verhandlung von nicht-funktionalen Parametern und bei der Instanzierung von Komponenten näher beleuchtet. Zum Abschluss wird noch auf die Möglichkeiten zur bindungsspezifischen Behandlung von Fehlern eingegangen.

6.3.1 Aufruf

Die Kommunikation zwischen PCom Komponenteninstanzen ähnelt in vielerlei Hinsicht der Kommunikation von BASE Diensten. Vom Objekt der Anwendung, also dem Dienst (BASE) oder der *Instance* (PCom), wird eine Methode eines *Proxies* aufgerufen. Dieser erzeugt eine Nachricht, die von *InvocationBrokern* mit Hilfe der *Registry* über *Plug-ins* an das entsprechende *Skeleton* weitergeleitet wird. Das Skeleton ruft die Methode des Anwendungsobjekts auf und liefert das Ergebnis über *InvocationBroker* und *Plug-ins* zurück an den *Proxy* und damit an das aufrufende Anwendungsobjekt.

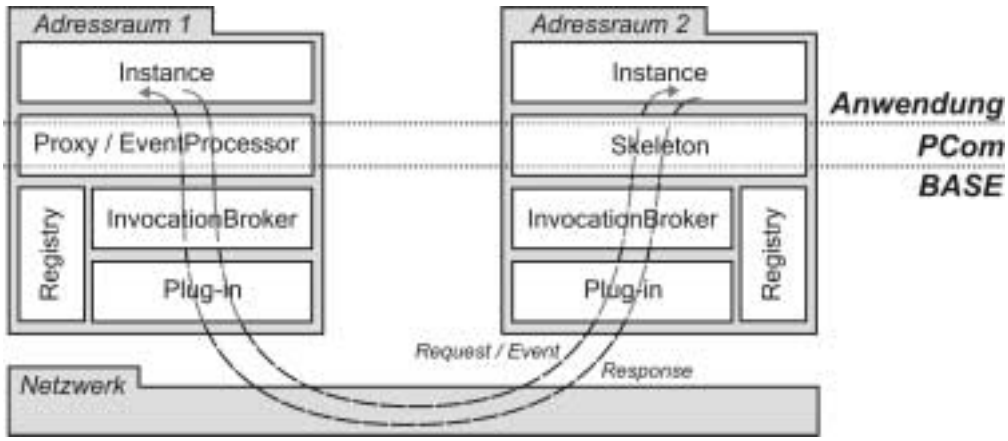

Abbildung 20 – Aufruf mit Adressraumwechsel

Während BASE Dienste jedoch ausschließlich Aufrufe an den genutzten Dienst unterstützen, ermöglicht PCom durch den Benachrichtigungsmechanismus auch Aufrufe, die von der genutzten Komponenteninstanz ausgelöst werden und an die nutzende Komponenteninstanz gerichtet sind. Diese Aufrufe werden über den *EventProcessor* der genutzten Komponenteninstanz an

den *InvocationBroker* weitergereicht. Der Rest des Aufrufs erfolgt analog zu Aufrufen, die von der nutzenden Komponenteninstanz ausgelöst wurden.

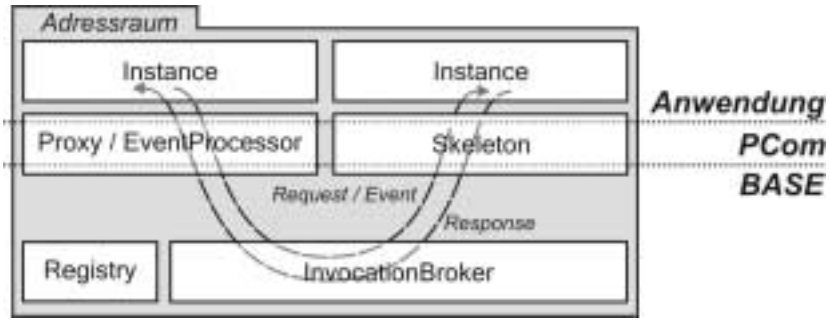

Abbildung 21 – Aufruf ohne Adressraumwechsel

Wie BASE optimiert auch PCom Aufrufe, die den lokalen Adressraum nicht verlassen. Im Gegensatz zu entfernten Aufrufen werden lokale Aufrufe direkt vom *InvocationBroker* zugestellt. Da die Aufrufe kein *Plug-in* durchlaufen, werden die Parameter lokaler Aufrufe nicht kopiert[7], was sich positiv auf die Performance auswirkt. Allerdings müssen Entwickler von Komponenten wegen dieser Optimierung sicherstellen, dass Aufrufe zwischen Komponenteninstanzen frei von Seiteneffekten bezüglich der übergebenen Parameter und Rückgabewerte sind, da diese sonst, im Vergleich zum entfernten Fall, zu unterschiedlicher Semantik führen würden.

6.3.2 Verhandlung

Bei der Verhandlung und Bestimmung nicht-funktionaler Parameter einer Komponenteninstanz müssen die drei folgenden Fragen geklärt werden:

- Können die Anforderungen an die Komponenteninstanz von ihr erfüllt werden? D.h., kann eine angeforderte Komponenteninstanz die Anforderungen im Bezug auf syntaktische und nicht-funktionale Parameter erfüllen, falls ihre Abhängigkeiten erfüllt werden können und welche Anforderungen an die Umgebung ergeben sich daraus?
- Können die Anforderungen an das lokale System erfüllt werden? D.h., sind genügend Ressourcen vorhanden, um die Komponenteninstanz unter den gewünschten lokalen Bedingungen auszuführen?
- Können die Anforderungen an benötigte Komponenteninstanzen erfüllt werden? D.h., können alle notwendigen Komponenteninstanzen

[7] Dies gilt nicht für Parameter und Rückgabewerte der Java Basistypen (z.B. int, long, float, usw.) und die Objektreferenz des Parameters, da diese als Wertparameter (call-by-value) übergeben werden und daher bei jedem Methodenaufruf von der Java Virtual Machine kopiert werden.

gefunden werden und können diese die an sie gerichteten Anforderungen erfüllen.

Das folgende Sequenzdiagramm gibt Aufschluss darüber, wie die Funktionseinheiten von PCom zusammenspielen, um genau diese Fragen zu beantworten.

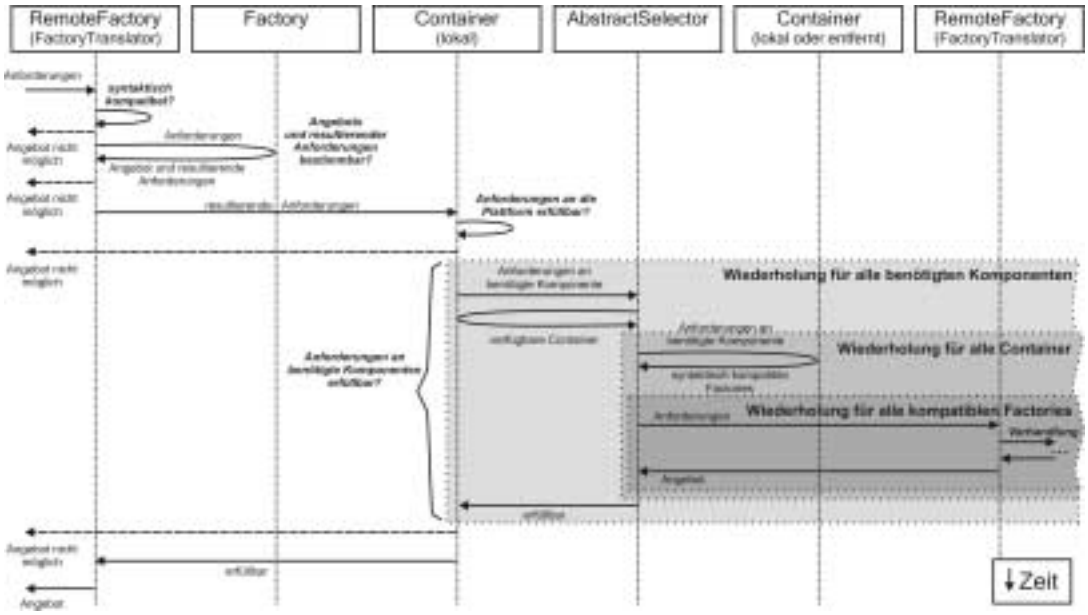

Abbildung 22 – Verhandlungen

Wird eine *RemoteFactory* dazu aufgefordert, das Angebot für bestimmte Anforderungen zu bestimmen, prüft sie zunächst, ob das Angebot syntaktisch erfüllt werden kann. Falls die Komponenteninstanz syntaktisch nicht mit den Anforderungen kompatibel ist, kann sie die Bestimmung der Parameter abbrechen. Falls ein Angebot syntaktisch möglich ist, werden die Konditionen von der *Factory* der Komponente bestimmt.

Danach wird vom *Container* geprüft, ob er die lokalen Anforderungen erfüllen könnte. Falls die Anforderungen nicht erfüllt werden können, kann die weitere Prüfung der Anforderungen sofort abgebrochen werden.

Zuletzt wird vom *AbstractSelector* geklärt, ob die Anforderungen an Komponenteninstanzen erfüllbar sind. Falls alle Anforderungen erfüllt werden können, ist die Verhandlung beendet und das Angebot kann zurückgeliefert werden. Ansonsten muss die Verhandlung abgebrochen werden, da das berechnete Angebot nicht gültig ist.

Exemplarisch zeigt das Sequenzdiagramm eine simple Selektionsstrategie, die alle verfügbaren Factories der Reihe nach prüft. Komplexere Algorithmen

könnten beispielsweise Caching verwenden, um die Zahl der entfernten Aufrufe zu minimieren.

6.3.3 Instanzierung

Die Instanzierung von Komponenten folgt aus dem Schema der Verhandlung von Parametern. Wenn Komponenten instanziert werden, muss ebenfalls geklärt werden, ob diese die an sie gerichteten Anforderungen erfüllen können und ob Plattform und notwendige Komponenteninstanzen die daraus resultierenden Anforderungen ebenfalls erfüllen können. Zusätzlich müssen eine Reihe von Registrierungen durchgeführt werden, die für den Betrieb einer Instanz notwendig sind.

Das folgende Sequenzdiagramm zeigt den Ablauf der Instanzierung von Komponenten.

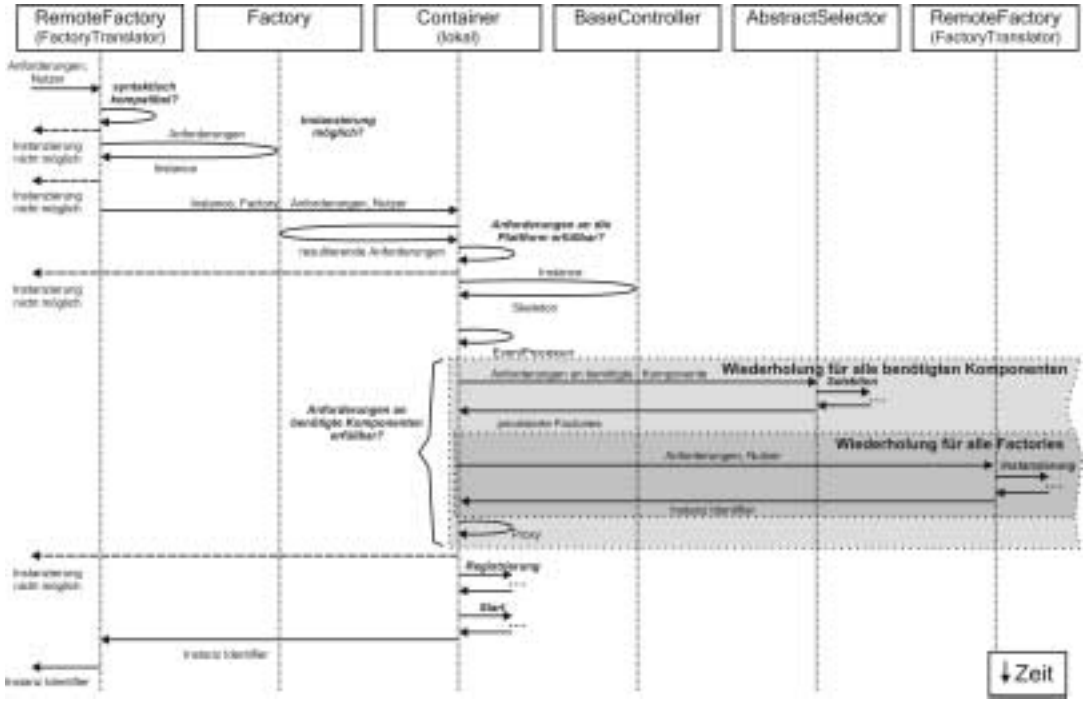

Abbildung 23 – Instanzierung von Komponenten

Wie die Verhandlung von Parametern beginnt auch die Instanzierung von Komponenten mit einer Anfrage an die *RemoteFactory*, die die Anforderungen enthält. Zusätzlich muss noch die Adresse des Nutzers mitgeliefert werden. Diese Adresse wird später für die Erzeugung eines *EventProcessors* benötigt. Nach einer syntaktischen Prüfung wird eine *Instance* durch die *Factory* erzeugt.

Im Anschluss daran werden die resultierenden Anforderungen bestimmt und im Bezug auf die Plattform geprüft. An dieser Stelle müssen die Ressourcen der Plattform bereits reserviert werden. Falls die Anforderungen an andere Komponenteninstanzen nicht erfüllt werden können, müssen die Ressourcen der Plattform wieder freigegeben werden. Nach erfolgreicher Reservierung werden ein *Skeleton* und ein *EventProcessor* erzeugt. Das *Skeleton* liefert die Adresse der Komponenteninstanz, die für rekursive Instanzierungen als Nutzeradresse dient.

Daraufhin bestimmt der *AbstractSelector* eine priorisierte Liste aller *Factories*, die genutzt werden können, um eine Abhängigkeit zu erfüllen. Diese Liste wird vom *Container* dazu verwendet, um weitere Komponenteninstanzen rekursiv zu erzeugen. Falls alle Abhängigkeiten erfüllt werden konnten, wird die Instanz gestartet und die Adresse der neu gebildeten Komponenteninstanz zurückgegeben. Damit kann der Nutzer der Komponenteninstanz einen Proxy für diese erzeugen. Für den Fall, dass es Abhängigkeiten gibt, die nicht erfüllt werden können, müssen alle Instanzen gestoppt werden, die bis dahin erzeugt wurden.

6.3.4 Fehlerbehandlung

PCom erweitert den BASE Mechanismus zur Behandlung von Fehlern beim entfernten Aufruf um die Möglichkeit zur Spezifikation von bindungsspezifischem Verhalten. Während BASE für jeden Kommunikationsfehler eine *InvocationException* auslöst, kann PCom Fehler flexibler abfangen. Dazu verfügt jeder *Proxy* über einen sogenannten *ExceptionHandler*.

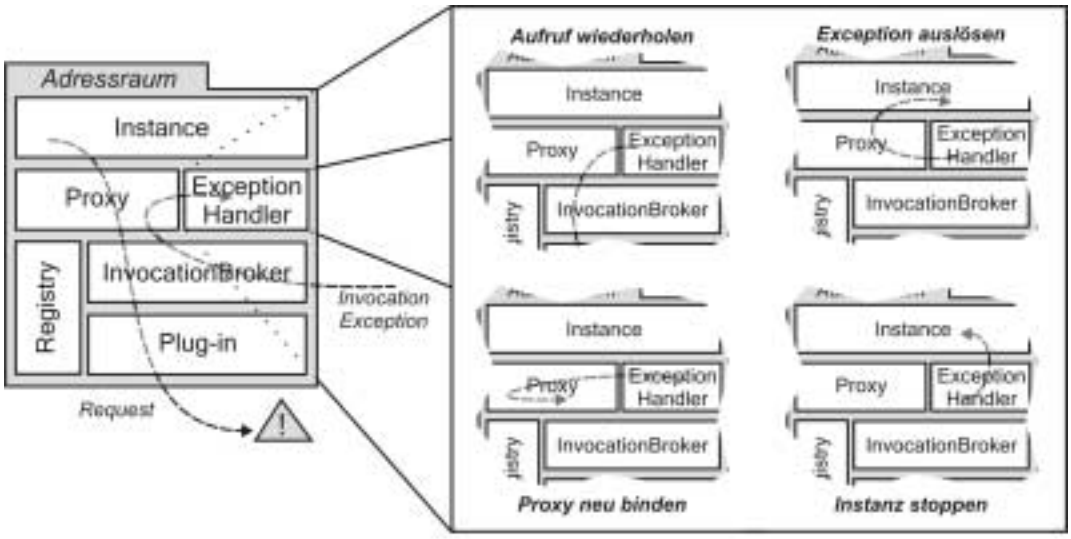

Abbildung 24 – Bindungsspezifische Fehlerbehandlung

Im Fehlerfall werden Exceptions, die von BASE erzeugt wurden, nicht direkt an die aufrufende Komponenteninstanz weitergeleitet, sondern zunächst wird der *ExceptionHandler* über den missglückten Kommunikationsversuch informiert. Der ExceptionHandler hat nun verschiedene Möglichkeiten, auf den Kommunikationsfehler zu reagieren. Die obige Abbildung zeigt einige Beispiele hierfür.

Ein wichtiger Unterschied zwischen PCom und BASE zeigt sich auch in der Semantik der Exceptions, die vom *ExceptionHandler* ausgelöst werden können. Während alle Methoden einer BASE Dienstschnittstelle *InvocationExceptions* deklarieren, erfordern Schnittstellen in PCom keine derartige Erweiterung, da mit dem *ExceptionHandler* die Möglichkeit besteht, Kommunikationsfehler sinnvoll zu behandeln, ohne die Schnittstelle der Komponente zu ändern. BASE *InvocationExceptions* beschreiben demnach jede Art von Kommunikationsfehler, wohingegen Exceptions, die von einem *ExceptionHandler* ausgelöst werden, lediglich Kommunikationsfehler beschreiben, auf die nicht mehr sinnvoll reagiert werden kann. In der Regel ist daher das Auslösen einer *Exception* im *ExceptionHandler* gleichbedeutend mit der Tatsache, dass die Komponenteninstanz ihre Leistungen in der bisherigen Form nicht weiter erbringen kann.

7 Evaluation

Nach der Vorstellung der Konzepte und der Entwicklung einer Implementierung kann nun mit Hilfe der Implementierung die Frage geklärt werden, welche Kosten eine komponentenbasierte Verteilungsinfrastruktur verursacht. Da PCom relativ lose mit BASE verbunden ist und der Kommunikationsmechanismus daher einfach ausgetauscht werden kann, stehen die zusätzlichen Kosten, die durch den Einsatz von PCom entstehen, im Vordergrund.

Zunächst stellt die Implementierung einer komponentenbasierten Verteilungsinfrastruktur wie PCom einen zusätzlichen Entwicklungsaufwand dar. Um diesen Aufwand abzuschätzen, werden einige Kennzahlen erhoben und mit den entsprechenden Kennzahlen der dienstbasierten Verteilungsinfrastruktur BASE verglichen.

Die Kosten für den tatsächlichen Einsatz von PCom setzten sich aus zwei Teilen zusammen. Zum einen kann die Erweiterung eines Programms um zusätzlichen Code zu höherem Speicherprofil führen. Im Folgenden wird deshalb weiterhin der notwendige Speicherbedarf beleuchtet.

Zum anderen kann die Geschwindigkeit des Programms unter den Erweiterungen leiden. Diese setzen sich aus einer Reihe verschiedener Faktoren zusammen: Kesselman und Wilson [Kesselman00] folgend, spielen beispielsweise die Geschwindigkeit von Berechnungen, die Startdauer und die Reaktionszeiten eine wichtige Rolle. Für Verteilungsinfrastrukturen ist jedoch vor allem die Geschwindigkeit der Übermittlung von Aufrufen wichtig. Daher wird im letzten Teil des Kapitels die Dauer der Übermittlung von Aufrufen in PCom und BASE verglichen.

7.1 Implementierung

Die Messung der Komplexität und des Umfangs eines Programms ist ein vielschichtiges Problem. Für objekt-orientierte Programme existieren unzählige Einzelmetriken und sogar ganze Metriksätze [Kidd96, Pressman00]. Aus den möglichen Alternativen wurden einige bestimmt, um die Komplexität und den Umfang von PCom zu verdeutlichen. Wie bereits erwähnt, sollen keine absoluten Aussagen getroffen werden. Der Fokus liegt klar auf dem Vergleich von BASE und PCom. Dadurch wird der Auswahlprozess der Metriken vereinfacht, denn die absoluten Werte der Metriken sind praktisch bedeutungslos.

7.1.1 Messung

Für die Auswahl von Kennzahlen der Implementierung diente die Messung von BASE, die in [Gubbels02] durchgeführt wurde, als Grundlage. Zur Messung des Umfangs von PCom und BASE wurden daher die drei Metriken *Zahl der Codezeilen* (innerhalb von Methoden und ohne Kommentar- und Leerzeilen), *Zahl der Klassen* und *Zahl der Interfaces* erhoben. Um die Komplexität des Codes zu veranschaulichen, wurden die Metriken *zyklomatische Komplexität nach McCabe* (Durchschnitt pro Methode) und *Verschachtelungstiefe* (Durchschnitt pro Methode) gewählt.

Alle Metriken wurden für die folgenden Programmteile erhoben:

- *BASE* (BASE mit Multicast Discovery und TCP Plug-in)
- *BASE Generator* (Proxy- und Skeletongenerator)
- *BASE Werkzeuge* (textbasierte Konsole)
- *PCom* (PCom mit klassenbasiertem Beschreibungslader ohne BASE)
- *PCom Generator* (Komponenten- und Beschreibungsgenerator)
- *PCom Werkzeuge* (graphische Konsole und Komponentengenerator)

Zur Erhebung aller Metriken wurde das Eclipse Metrics Plugin in der Version 1.3.2 verwendet[8]. Die folgende Tabelle zeigt die Ergebnisse der Messung:

	Zahl der Codezeilen	Zahl der Klassen	Zahl der Interfaces	Zyklomatische Komplexität	Verschachtelungs-tiefe
BASE	2236	49	13	2,718	1,741
BASE Generator	370	1	0	22,500	4,750
BASE Werkzeuge	111	3	0	4,667	2,167
PCom	1528	40	19	1,957	1,615
PCom Generator	615	1	0	17,000	4,714
PCom Werkzeuge	877	22	1	1,782	1,509

Tabelle 1 – Umfang und Komplexität

7.1.2 Auswertung

Betrachtet man die Ergebnisse der Metriken, die sich auf den Umfang der Implementierung beziehen, fällt auf, dass der Umfang des Kernsystems von

[8] Die aktuelle Version des Eclipse Metrics Plugins kann unter http://metrics.sourceforge.net kostenlos heruntergeladen werden. Um das Plugin zu nutzen wird zusätzlich die Eclipse Runtime benötigt, die unter http://www.eclipse.org heruntergeladen werden kann.

PCom ca. 75% dem des Kernsystems von BASE entspricht. Damit ist PCom, im Bezug auf den Umfang, eine nicht unerhebliche Erweiterung.

Beim Vergleich der Generatoren stellt man fest, dass der Generator für BASE deutlich kleiner ist als der für PCom. Dies lässt sich auf die Tatsache zurückführen, dass der Generator von PCom mehr Funktionen liefert. Zusätzlich zur komplexeren Proxy- und Skeletongenerierung übernimmt der PCom Generator die Transformation von XML-basierten Komponentenbeschreibungen in Java Klassen.

Die PCom Werkzeuge sind vor allem dadurch, dass diese graphisch sind, umfangreicher als die Werkzeuge, die von BASE geliefert werden. Bei der Betrachtung der Komplexität der Werkzeuge fällt gleichzeitig auf, dass der PCom Code eine vergleichsweise niedrige Komplexität hat. Auch dies lässt darauf schließen, dass die Unterschiede durch die graphische Darstellung entstehen.

Im Blick auf die Kernsysteme ist die zyklomatische Komplexität von BASE ein wenig höher als die von PCom. Dies liegt vor allem daran, dass der BASE Mikrokern durch die optimierte Implementierung die Komplexität des Gesamtsystems deutlich anhebt. Für die restlichen Teile von BASE ergibt sich eine Komplexität, die dem des PCom Kernsystems entspricht.

Betrachtet man die Komplexität der Generatoren, fällt zunächst auf, dass diese deutlich höher liegt als bei allen anderen Programmteilen. Dies lässt sich jedoch damit begründen, dass ihre Architektur recht einfach ist. Der Code jeder Klasse, die sie generieren, wird in einer einzigen Methode erzeugt. Da diese Methoden jedoch ausführlich kommentiert sind, ergibt sich daraus kein Nachteil für die spätere Wartung.

Insgesamt lässt sich damit festhalten, dass die Implementierung einer komponentenbasierten Verteilungsinfrastruktur auf Basis einer dienstbasierten ungefähr denselben zusätzlichen Aufwand verursacht wie die Implementierung einer dienstbasierten Verteilungsinfrastruktur.

7.2 Speicherbedarf

Aus der Definition von Ubiquitous Computing folgt, dass Speicherbedarf eine besondere Rolle spielt. Ubiquitous Computing zielt auf die Kooperation unterschiedlichster Geräte ab. Dies schließt auch kleinste Geräte mit knappen Ressourcen ein. Für solche Geräte ist hoher Speicherbedarf ein Ausschlusskriterium. Aus diesem Grund ist es wichtig, den Speicherbedarf, der durch den Einsatz von PCom entsteht, zu veranschaulichen.

7.2.1 Messung

Da PCom auf BASE aufbaut, macht es keinen Sinn, PCom getrennt zu betrachten. Stattdessen wird ein funktionsfähiges BASE Kernsystem (mit Multicast Discovery und TCP Plug-in) mit einer funktionsfähigen Version von PCom (PCom mit klassenbasiertem Beschreibungslader und BASE mit Multicast Discovery und TCP Plug-in) verglichen.

Der Speicherbedarf von Programmen kann in zwei Klassen unterteilt werden. Zum einen belegen die übersetzten Binärdateien eines Programms eine gewisse Menge an Speicher, zum anderen benötigt ein Programm, das ausgeführt wird, zusätzlichen Speicher zur Verwaltung von Code, Variablen, Aufrufstapeln und Ähnlichem.

Um die erste Klasse abzudecken, wurde die Größe eines Java Archivs bestimmt, das alle benötigten Java Klassen in Form von Java Bytecode enthält. Bei der Erstellung des Java Archivs wurde die Standardkomprimierung nicht deaktiviert.

Die Messung der zweiten Klasse gestaltete sich wesentlich schwieriger, da der Speicherbedarf zur Laufzeit vom jeweiligen Zustand des gesamten Systems abhängt. Aus diesem Grund wurden drei verschiedene Messungen durchgeführt. Zunächst wurde jeweils der Speicherbedarf eines ruhenden BASE und PCom Systems ohne Dienste bzw. Komponenten gemessen. In der darauf folgenden Messung wurde dem ruhenden System jeweils ein Dienst bzw. eine Komponente hinzugefügt. Bei der letzten Messung wurde jeweils eine Instanz des Diensts bzw. der Komponente mit maximaler Geschwindigkeit synchron aufgerufen.

Von [Shirazi00] inspiriert wurde der Implementierung von PCom und BASE der folgende Programmcode hinzugefügt, der ungefähr einmal pro Sekunde eine Messung vornimmt, um den Speicherbedarf zu bestimmen:

```
Thread t = new Thread() {
  public void run() {
    Runtime runtime = Runtime.getRuntime();
    while (true) {
      try {
        runtime.gc();                        // Hinweis für Garbage Collector
        long total = runtime.totalMemory(); // reservierter Speicher
        long free  = runtime.freeMemory();  // freier Speicher
        long used  = total - free;          // genutzer Speicher
        System.out.println("Total Memory: " + total);
        System.out.println("Free Memory : " + free);
        System.out.println("Used Memory : " + used);
        Thread.sleep(1000);                 // ~1 Messung/Sekunde
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
};
t.start();
```

Der Speicherbedarf dieses zusätzlichen Threads kann in Anbetracht des Umfangs von BASE und PCom vernachlässigt werden. Es muss jedoch beachtet werden, dass der Thread nicht den Speicherbedarf der gesamten Java Virual Machine bestimmt, sondern lediglich den genutzten Speicher innerhalb der Java Umgebung abschätzt. D.h. der Speicherbedarf des Gesamtsystems kann daher deutlich höher sein.

Während der Messung stellte sich heraus, dass der tatsächliche Speicherbedarf ständig schwankte. Aus diesem Grund beschreiben die unten aufgeführten Werte ein Intervall mit einer Abweichung von ungefähr -2 KB bis +2 KB.

Gemessen wurde mit einer IBM J9 Version 2.0 (CDC/Foundation Profile) Java Virtual Machine auf einem Intel x86 unter Windows XP Professional mit Service Pack 1. Dabei ergaben sich die folgenden Werte:

	Java Archiv (JAR)	Ruhend – ohne Dienst/Komponente	Ruhend – mit Dienst/Komponente	Ausgeführt – mit Dienst/Komponente
BASE	82 KB	91 KB	93 KB	114 KB
PCom (mit BASE)	153 KB	106 KB	121 KB	147 KB

Tabelle 2 – Speicherbedarf

7.2.2 Auswertung

Die statische Größe beider Programme in Java Archiven bringt deutlich zum Ausdruck, was der Umfang der Implementierung bereits vermuten lässt: PCom erhöht den Speicherbedarf um nahezu Faktor zwei.

Im Hinblick darauf scheint es zunächst überraschend, dass PCom nur zu einer um ca. 15% höheren Speicherbelegung, im ruhenden Zustand ohne Komponenten, führt. Bei einer genaueren Betrachtung stellt sich jedoch heraus, dass ohne Komponenten nahezu alle Klassen zur Beschreibung von Komponenten nicht geladen werden. Das drückt sich auch darin aus, dass durch Laden einer einzelnen Komponente im ruhenden Zustand der Speicherbedarf um 15 KB erhöht wird. Fügt man eine zweite Komponente hinzu, steigt der Speicherbedarf nur noch um ungefähr 1 KB.

Werden BASE und PCom unter Last betrieben, steigt der Speicherbedarf bei beiden Systemen um rund 20-30 KB. Der etwas höhere Speicherbedarf von PCom lässt sich dabei auf den höheren Verwaltungsaufwand des Containers zurückführen.

Insgesamt kann damit geschlossen werden, dass der zusätzliche Einsatz von PCom zu einem Speicherbedarf führt, der bis zu 100% höher liegt als der von BASE. Bei der Betrachtung des Speicherbedarfs darf jedoch nicht vergessen

werden, dass die Messungen nicht den gesamten Speicherbedarf bestimmen. Der hohe zusätzliche Speicherbedarf wird durch die Größe der Java Virtual Machine (ca. 1-2 MB) relativiert.

7.3 Geschwindigkeit

Bei der Messung der Geschwindigkeit einer Verteilungsinfrastruktur ist vor allem die Dauer der Zustellung von Aufrufen wichtig. Da PCom eine Erweiterung von BASE ist, stellt sich die Frage, inwiefern sich die Nutzung von PCom auf die Geschwindigkeit der Kommunikation auswirkt. Mit den folgenden Messungen soll deshalb die Dauer der Zustellung von Aufrufen in BASE der von PCom gegenüber gestellt werden.

7.3.1 Messung

Für die Messung der Geschwindigkeit wurden jeweils zwei BASE Dienste und zwei PCom Komponenten implementiert. Jeweils ein Dienst bzw. eine Komponente übernahm dabei die Rolle des Nutzers, während der andere Dienst bzw. die andere Komponente ein Java Interface anbot. Das Interface verfügte dabei über eine Methode, die eine beliebig lange Zeichenkette als Parameter entgegennahm und keinen Rückgabewert zurücklieferte. Die Implementierung dieser Methode war sowohl im BASE Dienst, als auch in der PCom Komponente ohne Funktion, d.h. die Methode kehrte nach dem Aufruf sofort zurück.

Gemessen wurde die Zeit, die vom nutzenden Dienst bzw. der nutzenden Komponente benötigt wurde, um 100 entfernte und 1000 lokale, synchrone Aufrufe durchzuführen. Die Länge der Zeichenkette betrug beim ersten Durchlauf 0 und beim zweiten 1000 Zeichen. Insgesamt wurde jede Messung 50 Mal wiederholt. Die einzelnen Werte wurden dabei mit folgendem Codefragment bestimmt:

```
String message = …;                          // Zeichenkette initialisieren
long start = System.currentTimeMillis();     // Beginn in ms festhalten
for (int i = 100; i > 0; i--) {
  serviceOrComponent.call(message);        // entfernter Aufruf druchführen
}
long stop = System.currentTimeMillis();      // Ende in ms festhalten
long duration = stop – start;                // Dauer berechnen
System.out.println("Duration: " + duration);
```

Wie für die Messung des Speicherbedarfs wurde auch für die Messung der Geschwindigkeit die IBM J9 Version 2.0 (CDC/Foundation Profile) Java Virtual Machine genutzt. Weiterhin wurden zwei PCs verwendet, die über ein 100MBit Ethernet verbunden waren. Durch einen vorgeschalteten Switch waren beide Rechner vom restlichen Netzwerk getrennt. Der aufrufende Dienst bzw. die aufrufende Komponente wurde auf einem PC mit Intel Pentium 4M

Prozessor mit einer Taktfrequenz von 1700 MHz und 768MB RAM unter Windows XP Professional mit Service Pack 1 ausgeführt. Der aufgerufene Dienst bzw. die aufgerufene Komponente wurde auf einem PC mit Intel Pentium 2 Prozessor mit einer Taktfrequenz von 200 MHz und 256 MB RAM unter Windows 2000 Professional mit Service Pack 3 ausgeführt. Für die lokalen Messungen wurde ausschließlich der leistungsstärkere Rechner genutzt.

Die nachfolgende Tabelle zeigt den Durchschnitt sowie den minimalen und den maximalen Wert dieser Messungen:

	100 entfernte Aufrufe 0 Zeichen/Aufruf	100 entfernte Aufrufe 1000 Zeichen/Aufruf	1000 lokale Aufrufe 0 Zeichen/Aufruf	1000 lokale Aufrufe 1000 Zeichen/Aufruf
BASE	1273 (1152/1532) ms	1662 (1572/2063) ms	819 (801/922) ms	819 (801/931) ms
PCom (mit BASE)	1281 (1201/1502) ms	1650 (1582/2094) ms	819 (801/941) ms	818 (801/942) ms

Tabelle 3 – Geschwindigkeit

7.3.2 Auswertung

Betrachtet man die Durchschnittswerte, fällt zunächst auf, dass PCom in der zweiten Messung schneller ist als BASE. Die Tatsache, dass es sich dabei um eine Differenz von zwölf Millisekunden handelt, deutet auf eine ungenaue Messung hin. Die Abweichung der ersten Messung ist kleiner und kann damit auch auf ungenaue Messungen zurückgeführt werden. Für lokale Aufrufe können praktisch keine Abweichungen festgestellt werden.

Zusammenfassen kann demnach festgestellt werden, dass die Entscheidung, die Implementierung von PCom Proxies und Skeletons durch Vererbung auf den Klassen von BASE aufzubauen, dazu führt, dass die Nutzung von PCom keinen negativen Effekt auf die Dauer der Übermittlung von Aufrufen hat.

8 Zusammenfassung

Im Folgenden werden die Ergebnisse dieser Arbeit kompakt dargestellt. Dabei werden sie nochmals eindeutig mit den Anforderungen in Verbindung gebracht. Im anschließenden Ausblick sind zukünftige Fragestellungen motiviert und Vorschläge für deren Lösung umrissen. Zum Abschluss der Arbeit werden die Schlüsse vorgestellt, die aus ihr gezogen werden können.

8.1 Ergebnisse

Anwendungen für ubiquitäre Rechnersysteme begegnen hohen Anforderungen, die eine Obermenge der Anforderungen herkömmlicher verteilter Systeme darstellen.

In dieser Arbeit wurden aus den drei besonderen Eigenschaften ubiquitärer Rechnersysteme neun Anforderungen abgeleitet, die von bestehenden Systemen zumeist nur teilweise und in unterschiedlichem Maße abgedeckt werden. Der Vergleich verschiedener Ansätze macht deutlich, dass derzeit keines der bestehenden Systeme die Summe aller Anforderungen umfassend und damit zufrieden stellend löst.

Grundlage für den Lösungsansatz war die Annahme, dass es sinnvoll ist, Abstraktionen, die sich bereits beim Einsatz in verteilten Systemen bewährt haben, an die neuen Anforderungen anzupassen. Konkret sollten die Vorteile moderner Komponentensysteme, deren kommerzieller Erfolg in herkömmlichen verteilten Systemen unbestritten ist, für ubiquitäre Rechnersysteme nutzbar gemacht werden.

Dazu wurde ein einfaches Anwendungsmodell vorgeschlagen, anhand dessen die notwendigen Konzepte der Komponenten erarbeitet wurden. Dabei stand die zentrale Aufgabe komponentenbasierter Verteilungsinfrastrukturen, die automatisierte Verwaltung von Abhängigkeiten, im Mittelpunkt.

Auf Basis der Konzepte wurde anschließend das funktionsfähige Komponentensystem PCom entwickelt. Die Implementierung machte sich die Plattformenabhängigkeit der Java 2 Micro Edition und die Flexibilität der dienstbasierten Verteilungsinfrastruktur BASE zunutze. Während der Entwicklung wurde vor allem auf hohe Geschwindigkeit, niedrigen Speicherbedarf und gute Wartbarkeit Wert gelegt.

Die anschließende Evaluation der Implementierung zeigte, dass das Komponentensystem eine vergleichsweise hohe Geschwindigkeit lieferte. Das Ziel, den Speicherbedarf zu minimieren, konnte jedoch nicht vollständig zufrieden stellend erreicht werden. Grund hierfür waren Abwägungen zwischen Wart-

barkeit, Speicherbedarf und Geschwindigkeit, die in der Regel zugunsten besserer Wartbarkeit oder höherer Geschwindigkeit getroffen wurden.

Die Anforderungen ubiquitärer Rechnersysteme zogen sich durch die gesamte Arbeit und wirkten in folgender Weise auf unterschiedlichste Teile:

Die aus der Heterogenität resultierende Forderung nach Verständlichkeit drückte sich durch den Einsatz von Java und die Entwicklung diverser Codegeneratoren aus. Darüber hinaus wurde die Steuerung von Abhängigkeiten zwischen Komponenteninstanzen mit Controllern einheitlich umgesetzt. Weiterhin kam für die Beschreibung von Komponenten XML zum Einsatz, womit für die Anpassung von Beschreibungen keine Javakenntnisse notwendig sind.

Auch die Forderung nach Minimalität kam in der Wahl der Plattform zum Ausdruck. Mit Java 2 Micro Edition und BASE wurden Plattformen genutzt, die speziell für ressourcenarme Rechnersysteme entwickelt wurden. Weiterhin sorgte die Entwicklung eines Codegenerators zur Transformation von XML Beschreibungen in Java Klassen dafür, dass die Anforderungen an Rechnersysteme verringert wurden.

Interoperabiltät wird durch die Nutzung der zugrunde liegenden Verteilungsinfrastruktur BASE gewährleistet. Im Bezug auf die Implementierung des Komponentensystems musste daher lediglich sichergestellt werden, dass jedwede Kommunikation zwischen verschiedenen Java Virtual Machines mit Hilfe von BASE Mechanismen durchgeführt wird. Dies führte letztlich dazu, dass Container und Factories als BASE Dienste modelliert wurden und dass Komponenten über erweiterte BASE Skeletons und Proxies kommunizieren.

Anforderungen, die sich aus der Mobilität und spontanen Vernetzung ubiquitärer Rechnersystemen ergeben, wurden zum größten Teil durch den Einsatz von BASE abgedeckt. Discovery und föderativer Namensdienst wurden für die Integration mobiler Rechner genutzt.

Während die Integration mobiler Geräte von der Infrastruktur automatisiert werden kann, ist für die Adaption ein reibungsloses Zusammenspiel mit der Anwendung notwendig. Um dem gerecht zu werden, wurde in dieser Arbeit eine Reihe von Mechanismen entwickelt. Über Komponentenbeschreibungen wurde das Komponentensystem mit der Möglichkeit ausgestattet, selbständig Komponenten zu selektieren. Damit Anwendungen auf diese Selektion Einfluss nehmen können, wurden dynamische Änderungen der Beschreibung ermöglicht. Weiterhin wurden ExceptionHandler und Controller entwickelt, um Anwendungen weitergehende Eingriffe zu erlauben.

Auf die Anforderungen im Bezug auf die Administration wurde nur oberflächlich eingegangen. Dies lag vor allem daran, dass Funktionen, wie beispielsweise die Migration von Komponenten, für die Konzeption eines mini-

malen Basissystems nicht notwendig erschienen. Die grundlegenden Anforderungen diesbezüglich wurden jedoch mit Hilfe von Komponentenbeschreibungen realisiert, mit denen sichergestellt wird, dass Abhängigkeiten nur mit kompatiblen Komponenten aufgelöst werden.

Kontextuelle Informationen wurden ebenfalls über Komponentenbeschreibungen modelliert. Erweiterbarkeit und die Möglichkeit, Beschreibungen zur Laufzeit zu ändern, liefern die Grundlage für die Spezifikation beliebiger kontextueller Informationen.

Damit Änderungen des Kontexts einer Komponente kommuniziert werden können, wurde ein Benachrichtigungsmechanismus entwickelt. Dieser Mechanismus machte sich die Möglichkeit zu Eigen, dass Nachrichten auch an die nutzende Komponenteninstanz weitergeleitet werden können.

Mit der vorgestellten komponentenbasierten Verteilungsinfrastruktur wurde auch die Verarbeitung kontextueller Informationen ermöglicht. Zur Demonstration der Konzepte wurde ein Lease Mechanismus implementiert. Komponenteninstanzen können dabei die Dauer eines Lease verhandeln, während die Infrastruktur für die konkrete Umsetzung sorgt.

8.2 Ausblick

Die komponentenbasierte Verteilungsinfrastruktur, die in dieser Arbeit vorgestellt wurde, soll als Grundlage für folgende Arbeiten dienen. Während der Entwicklung und Konzeption wurden eine Reihe zukünftiger Erweiterungsmöglichkeiten und Ansätze deutlich.

Zunächst scheint es sinnvoll, in zukünftigen Arbeiten die Schnittstellen zwischen PCom und BASE genauer zu evaluieren. Während der Implementierung von PCom wurde BASE als zugrunde liegende Plattform unverändert übernommen. Rückblickend scheint die Entscheidung nicht in jeder Hinsicht stichhaltig zu sein. Eine engere Integration von PCom und BASE würde zwar die Wartbarkeit negativ beeinflussen, sie könnte sich jedoch sowohl auf die Geschwindigkeit als auch auf die Größe des Gesamtsystems positiv auswirken. Darüber hinaus würde eine weitergehende Einbettung von PCom in BASE dazu führen, dass nicht-funktionale Parameter der Kommunikationskanäle von PCom genutzt werden könnten. So könnte beispielsweise die Verfügbarkeit von Plug-ins auf lokalen und entfernten Geräten während der Selektion von Komponenten genutzt werden, um die Qualität der Entscheidungen des Selektionsalgorithmus zu verbessern. Zusätzlich könnte PCom installierte Komponenten als BASE Dienste exportieren. Damit würden BASE Dienste die Möglichkeit erhalten, Komponenteninstanzen als Dienste zu nutzen. Im selben Zug sollte auch über die Integration von BASE Diensten in PCom nach-

gedacht werden. Hierfür müssten die derzeitigen BASE Dienste jedoch um kompatible Angebotsbeschreibungen erweitert werden.

Weiterhin gibt es eine Reihe interessanter Erweiterungen, mit denen der Aufwand für die Entwicklung von Komponenten verringert werden kann oder die dazu beitragen, die Flexibilität von PCom zu steigern. Beispielsweise bietet PCom derzeit keine Mechanismen, die den Umgang mit zustandsbehafteten Komponenteninstanzen erleichtern. Um diese besser integrieren zu können, bieten kommerzielle Komponentensysteme in der Regel Transaktionsunterstützung. Mit ähnlichen Mechanismen könnte PCom so erweitert werden, dass Komponenteninstanzen einer Anwendung unabhängig von ihrem Zustand ausgetauscht werden können. Zusätzlich könnte PCom um Mechanismen erweitert werden, mit denen die Migration von Komponenten ermöglicht wird, wodurch Verfügbarkeit von Komponenten erhöht werden könnte. Gleichzeitig könnte mit automatischen Aktualisierungsfunktionen auf Basis der Migration, die Administration von ubiquitären Rechnersystemen vereinfacht werden. Darüber hinaus wäre es wünschenswert, wenn die Sicherheit von PCom verbessert werden würde. Gerade in spontan vernetzten Rechnersystemen spielt Sicherheit eine wichtige Rolle, da nicht alle Rechner im Netz als vertrauenswürdig angesehen werden können. Da dieses Problem erwartungsgemäß eine große Zahl von Anwendungen betreffen wird, wäre es sinnvoll, wenn PCom standardisierte Mechanismen anbieten würde, um die Sicherheit von Anwendungen und deren Daten gleichermaßen zu gewährleisten.

Darüber hinaus könnten architektonische Spezifikationen bei der Komposition von Anwendungen genutzt werden. Derzeit kann PCom lediglich Anwendungen ausführen, wenn alle spezifizierten Komponenten verfügbar sind. Anwendungen könnten jedoch durchaus in der Lage sein, ihre Anforderungen wahlweise mit unterschiedlichen Komponenten zu erfüllen. Um solche Adaptionsmöglichkeiten zu spezifizieren, könnten Anwendungen über mehrere Architekturen verfügen, die je nach Zustand des Gesamtsystems eingesetzt werden könnten. Systeme wie Aura machen sich diese Idee bereits zu nutze. Mit einem verteilten Selektionsalgorithmus auf Basis der Beschreibungen möglicher Anwendungsarchitekturen könnte ein extrem hoher Grad an Flexibilität erreicht werden. Zum einen könnte die Stabilität von komponentenbasierten Anwendungen in ubiquitären Rechnersystemen deutlich erhöht werden, zum anderen könnte auf unterschiedlichste Optimierungskriterien eingegangen werden. Beispielsweise könnte, je nach Systemzustand, entweder die energieoptimierende oder die hoch verfügbare Architektur einer Anwendung zum Einsatz kommen.

Neben den vorgestellten Erweiterungen wären sicherlich die Evaluation und der Vergleich der dienst- und komponentenbasierten Abstraktionen von BA-

SE und PCom, besonders im Hinblick auf den Entwicklungsaufwand von Anwendungen, interessant. Hierfür könnten beispielsweise der Aufwand zweier Entwicklergruppen mit gleicher Qualifikation gemessen werden, die dieselbe Anwendung entwickeln, wobei eine Gruppe das dienstorientierte Paradigma von BASE anwendet, während die andere dem komponentenorientierten von PCom folgt. Da solche Experimente in der Regel zu viele unbekannte Variablen enthalten, könnte die Gültigkeit der grundlegenden Annahmen dieser Arbeit zwar nicht vollständig bestätigt oder widerlegt werden, allerdings wäre es sicherlich möglich, eine Tendenz abzuleiten. Weiterhin könnte die bloße Entwicklung von komponentenbasierten Anwendungen zur weiteren Standardisierung von Teilen des Komponentensystems führen. Beispielsweise könnten ExceptionHandler identifiziert werden, die in vielen Anwendungen zum Einsatz kommen. Diese könnten dann bereits von der Verteilungsinfrastruktur bereitgestellt werden.

Durch die Zahl und die Unterschiede der vorgestellten Arbeitsmöglichkeiten wird deutlich, dass durch die Konzeption und Entwicklung einer komponentenbasierten Verteilungsinfrastruktur für ubiquitäre Rechnersysteme zahlreiche Fragestellungen aufgeworfen werden, deren zukünftige Bearbeitung durchaus das Potential hat, interessante Ergebnisse zu liefern.

8.3 Fazit

Die Entwicklung von Anwendungen auf Basis von Softwarekomponenten bietet eine Reihe von Vorteilen, die vor allem auf die explizite Spezifikation von Abhängigkeiten zurückzuführen sind. Heutige Komponentensysteme machen sich die explizite Spezifikation zunutze, um Aufgaben zu automatisieren, die orthogonal zur Anwendungslogik liegen.

In ubiquitären Rechnersystemen gibt es eine Vielzahl solcher orthogonaler Aspekte, die von allen Anwendungen gleichermaßen berücksichtigt werden müssen. Die ständige Entwicklung von Programmteilen, die diese behandelt, verursacht überflüssige Kosten.

Würde man versuchen, die Anforderungen, die sich aus der Mobilität ubiquitärer Rechnersysteme ergeben, mit dienstbasierten Verteilungsinfrastrukturen abzudecken, müssten Entwickler viel Zeit und Mühe in Programmteile investieren, die sich ausschließlich mit der Verwaltung der unbeständigen Eigenschaften der Umgebung beschäftigen. Zu diesen Eigenschaften gehören vor allem die Auswahl und Nutzung von Funktionalität, die auf entfernten Geräten bereitgestellt wird und die Behandlung von Fehlern, die durch diese Nutzung entstehen.

In dieser Arbeit wurde eine komponentenorientierte Verteilungsinfrastruktur konzipiert, implementiert und evaluiert, mit der auf diese Problematik einge-

gangen wird. PCom extrahiert die Programmteile zur Verwaltung von Abhängigkeiten und zur Behandlung von Fehlern, die für die Nutzung entfernt verfügbarer Funktionalität notwendig sind, aus dem verbleibenden Teil der Anwendung. Damit können die extrahierten Programmteile einfacher wieder verwendet werden. Im Ausblick wurden einige Vorschläge gemacht, mit denen der Grad an Automation erhöht wird, wodurch der Nutzen für Anwendungen steigen könnte.

Zusätzlich ermöglicht PCom durch die erweiterbaren Beschreibungen von Komponenten die Einbettung höherer Abstraktionen, mit denen Anwendungen noch weiter automatisiert und optimiert werden können. Die Nutzung architekturbasierter Ansätze zur Flexibilisierung der Komposition von Anwendungen ist ein Beispiel hierfür.

Bei allen Vorteilen, die komponentenbasierte Verteilungsinfrastrukturen bieten, darf jedoch nicht verschwiegen werden, dass diese auch negative Effekte mit sich bringen. Als Beispiel hierfür kann der höhere Bedarf an Ressourcen aufgeführt werden. Der Vergleich mit BASE zeigt, dass die Nutzung von PCom vor allem den Bedarf an Speicher erhöht. Auch wenn dieser zusätzliche Bedarf auf vielen Rechnersystemen sicherlich vernachlässigt werden kann, so können dennoch keine allgemeingültigen Aussagen über die Praxistauglichkeit für die Hardware beliebiger, heterogener Rechnersysteme gemacht werden. Um dieses Problem abzumildern wurde im Ausblick vorgeschlagen, die Integration von BASE Diensten und PCom Komponenten voranzutreiben.

Ob Komponentensysteme kommerzielle Infrastruktur für ubiquitäre Rechnersysteme in Zukunft dominieren werden bleibt abzuwarten. Allerdings bieten sie zahlreiche Möglichkeiten zur Steigerung der Produktivität von Entwicklern und die Geschichte der Softwareentwicklung zeigt, dass sich oftmals die Abstraktionen durchsetzen, die hohe Produktivität und damit niedrige Kosten versprechen.

9 Literaturverzeichnis

[Abowd00] Gregory Abowd, Elizabeth Mynatt, **Charting Past, Present and Future Research in Ubiquitous Computing**, *ACM Transactions on Computer-Human Interaction*, 7(1), 29-58, März 2000

[Becker03a] Christian Becker, Gregor Schiele, Holger Gubbels, **BASE – A Micro-broker-based Middleware For Pervasive Computing**, *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communication*, 443-451, März 2003

[Becker03b] Christian Becker, Gregor Schiele, **Middleware and Application Adaptation Requirements and their Support in Pervasive Computing**, *Proceedings of the Third International Workshop on Distributed Auto-adaptive and Reconfigurable Systems at ICDCS*, Mai 2003

[Belaramani03] Nalini Moti Belaramani, Yuk Chow, Vivien Wai-Man Kwan, Cho-Li Wang, Francis C.M. Lau, **A Component-based Software Architecture for Pervasive Computing**, *to appear in Intelligent Virtual World: Technologies and Applications in Distributed Virtual Environments, World Scientific Publishing Co.*, 2003

[Beugnard99] Antoine Beugnard, Jean-Marc Jézéquel, Noel Plouzeau, Damien Watkins, **Making Components Contract Aware**, *IEEE Computer*, 32(7), 38-45, Juli 1999

[Chappell02] David Chappell, **.NET verstehen Einführung und Analyse**, *Addison-Wesley*, 2002

[Cheng02] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Joao Sousa, Bridget Spitznagel, Peter Steenkiste, Ningning Hu, **Software Architecture-based Adaptation for Pervasive Systems**, *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing (ARCS'02)*, 67-82, April 2002

[Date00] C. J. Date, **An Introduction to Database Systems – Seventh International Edition**, *Addison Wesley*, 2000

[DeLara01] Eyal de Lara, Dan Wallach, Willy Zwaenepoel, **Puppeteer: Component-based Adaption for Mobile Computing**, *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, 159-170, März 2001

[ECMA02] ECMA International, **C# Language Specification 2nd Edition**, *Standard ECMA-334, http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf*, Dezember 2002

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, **Design Patterns Elements of Reusable Object-Oriented Software**, *Addison-Wesley*, 1995

[Garlan02] David Garlan, Dan Siewiorek, Asim Smailagic, Peter Steenkiste, **Project Aura: Towards Distraction-Free Pervasive Computing**, *IEEE Pervasive Computing*, 1(2), 22-31, April-Juni 2002

[Grimm00] Robert Grimm, Tom Anderson, Brian Bershad, David Wetherall, **A system architecture for pervasive computing**, *Proceedings of the 9th ACM SIGOPS European Workshop*, 177-182, September 2000.

[Grimm01] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, David Wetherall, **System-level programming abstractions for ubiquitous computing**, *UbiTools'01 Workshop on Application Models and Programming Tools for Ubiquitous Computing*, September 2001

[Grimm02a] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, David Wetherall, **Programming for Pervasive Computing Environments**, *Technical Report UW-CSE-01-06-01, University of Washington*, Juni 2001

[Grimm02b] Robert Grimm, **System support for pervasive applications**, *PhD Thesis, University of Washington*, Dezember 2002

[Gubbels02] Holger Gubbels, **Basisdienste für Anwendungen in ubiquitären Rechnersystemen**, *Diplomarbeit 2016, Universität Stuttgart*, Dezember 2002

[Henricksen01] Karen Hendricksen, Jadwiga Indulska, Andry Rakotonirainy, **Infrastructure for Pervasive Computing: Challenges**, *Proceedings of the Workshop on Pervasive Computing and Information Logistics at Informatik 2001*, 25-28, September 2001

[Joy00] Bill Joy, Guy Steele, James Gosling, Gilad Bracha, **Java Language Specification**, *http://java.sun.com/docs/books/jls/, Addison-Wesley*, 2000

[Kesselman00] Jeff Kesselman, Steve Wilson, **Java Platform Performance: Strategies and Tactics**, *Addison-Wesley,* 2000

[Kidd96] Jeff Kidd, Mark Lorenz, **Object-Oriented Metrics: Measures of Complexity**, *Prentice Hall*, 1996

[Kon00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, Roy Campbell, Monitoring, **Security, and Dynamic Configuration with the DynamicTAO Reflective ORB**, *Proceed-*

ings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, 121-143, April 2000

[Microsoft03a] Microsoft Cooperation, **.NET Framework Home Page**, *http://msdn.microsoft.com/library/en-us/netstart/html/sdkstart.asp*, 2003

[Microsoft03b] Microsoft Cooperation, **.NET Compact Framework Overview**, *http://msdn.microsoft.com/vstudio/device/compactfx.aspx*, 2003

[Omg98] Object Management Group, **The Common Object Request Broker: Architecture and Specification (Revision 2.2)**, *http://www.omg.org-/docs/formal/98-07-01.pdf,* Februar 1998

[Omg02] Object Management Group, **CORBA Components (Version 3.0)**, *http://www.omg.org/docs/formal/02-06-65.pdf*, Juni 2002

[Pressman00] Roger Pressman, **Software Engineering A Practitioner's Approach (Fifth Edition, European Adaption)**, *McGraw-Hill*, 2000

[Román99] Manuel Román, Ashish Singhai, Dulcineia Carvalho, Christopher Hess, Roy Campbell, **Integrating PDAs into Distributed Systems: 2K and PalmORB**, *Lecture Notes in Computer Science*, 1707, 137-149, 1999

[Román00] Manuel Román, Roy Campbell, **Gaia: Enabling Active Spaces**, *9th ACM SIGOPS European Workshop*, September 2000

[Román01a] Manuel Román, Roy Campbell, Fabio Kon, **Reflective Middleware: From Your Desk to Your Hand**, *IEEE Distributed Systems Online*, 2(5), 2001

[Román01b] Manuel Román, Roy Campbell, **Unified Object Bus: Providing Support for Dynamic Management of Heterogeneous Components**, *Technical Report UIUCDCS-R-2001-2222, University of Illinois at Urbana-Champaign*, Mai 2001

[Shirazi00] Jack Shirazi, **Java Performance Tuning**, *O'Reilly & Associates*, 2000

[Sun01a] Sun Microsystems, **Java Remote Method Invocation (RMI)**, *http://java.sun.com/j2se/1.4/docs/guide/rmi/index.html*, 2001

[Sun01b] Sun Microsystems, **Jini Architecture Specification (Version 1.2)**, *http://wwws.sun.com/software/jini/specs/jini1.2html/jini-title.html*, Dezember 2001

[Sun02a] Sun Microsystems, **Java 2 Platform Micro Edition Overview**, *http://java.sun.com/j2me/j2me-ds.pdf*, 2002

[Sun02b] Sun Microsystems, **Enterprise JavaBeans Specification (Version 2.1 Proposed Final Draft)**, *http://java.sun.com/products/ejb/docs.html*, August 2002

[Sun03] Sun Microsystems, **Java 2 Platform Standard Edition (Version 1.4.2) API Specification**, *http://java.sun.com/j2se/1.4.2/docs/api/index.html*, 2003

[Szyperski98] Clemens Szyperski, **Component Software Beyond Object-Oriented Programming**, *Addison-Wesley*, 1998

[W3C00] World Wide Web Consortium, **Extensible Markup Language (XML) 1.0 (Second Edition)**, *http://www.w3.org/TR/REC-xml*, Oktober 2000

[Weiser91] Mark Weiser, **The Computer for the Twenty-First Century**, *Scientific American*, 265(3), 94-104, September 1991

[Weiser93a] Mark Weiser, **Some Computer Science Issues in Ubiquitous Computing**, *Communications of the ACM*, 36(7), 74-84, Juli 1993

[Weiser93b] Mark Weiser, **Ubiquitous Computing**, *IEEE Computer Hot Topics*, 71-72, Oktober 1993

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

Marcus Handte