Universität Stuttgart

Fakultät Informatik,
Elektrotechnik und
Informationstechnik

Diplomarbeit Nr. 2291

Entwicklung und Evaluation eines Algorithmus zur Anpassung von Anwendungen in PCOM

Andreas Störzbach

Studiengang: Informatik

Prüfer: Prof. Dr. Kurt Rothermel
Betreuer: Dipl.-Inf. Marcus Handte

begonnen am: 01.01.2005
beendet am: 04.07.2005

CR-Klassifikation: C.2.4, D.4.7, I.2.8

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

Zusammenfassung

Das 3PC-Projekt hat sich die Entwicklung minimaler, aber flexibler Systemunterstützung für adaptive, ubiquitäre Anwendungen zum Ziel gesetzt. Ubiquitäre Anwendungen bieten für die Benutzer neue Möglichkeiten und verbesserte Unterstützung bei der täglichen Arbeit. In dem Komponentensystem PCOM lassen sich Anwendungen als Komposition von Komponenten zusammensetzen. Da sich die Art und Anzahl der verfügbaren Ressourcen in der Umgebung verändern kann, ist eine Anpassung der Anwendungen auf die veränderte Umgebung notwendig.

Im Rahmen dieser Diplomarbeit wird die Anpassung von Anwendungen untersucht. Die Anpassung von Anwendungen verursacht in der Regel Kosten. Damit der zu entwickelnde Algorithmus die Kosten mit in die Suche nach einer Konfiguration einbeziehen kann, werden die Kostenfaktoren für die Anpassung von Anwendungen genauer untersucht und ein Kostenmodell entwickelt.

Die Anpassung von Anwendungen in PCOM kann durch einen vollständigen Optimierungsalgorithmus oder eine geeignete Heuristik erfolgen. Im Rahmen dieser Diplomarbeit wird eine Greedy Heuristik entwickelt, da der Aufwand zur Berechnung der optimalen Lösung im Allgemeinen zu hoch ist. Die Greedy Heuristik ist nicht immer optimal, aber in der Evaluation wird sich zeigen, dass die Heuristik sehr häufig eine Konfiguration in der Nähe des Optimum ermittelt und besser abschneidet als die Rekonfiguration ohne Beachtung von Kosteninformationen.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Ubiquitous Computing 1
1.2 Motivation 2
1.3 Aufgabenstellung 2
1.4 Aufbau der Arbeit 3
1.5 Einordnung der Diplomarbeit in 3PC 3

2 Grundlagen **5**
2.1 Rekonfiguration 5
2.1.1 Ursachen für die Rekonfiguration 5
2.1.2 Ort der Rekonfiguration 7
2.1.3 Art der Rekonfiguration 8
2.1.4 Zeitpunkt der Rekonfiguration 9
2.2 PCOM Grundlagen 9
2.2.1 PCOM Anwendungsmodell 9
2.2.2 PCOM Architektur 10
2.2.3 Distributed Constraint Satisfaction Problem 10
2.2.4 Asynchroner Backtracking Algorithmus 12
2.2.5 Finden einer initialen Konfiguration in PCOM 13
2.3 Verwandte Arbeiten 16
2.3.1 Klassische komponentenbasierte Verteilungsinfrastrukturen 16
2.3.2 Ubiquitäre Middleware 16
2.3.2.1 Gaia 17
2.3.2.2 Aura 20
2.3.3 Dienstgüteorientierte Verteilungsstrukturen 23
2.3.3.1 SpiderNet 23
2.3.3.2 $2K^{Q+}$ 25

3 Analyse **29**
3.1 Problemanalyse 29
3.1.1 Rekonfigurationsalternativen 30
3.1.2 Rekonfiguration unter Beachtung von Kostenfaktoren . 32
3.1.3 Lösungsstrategien 32
3.1.4 Anforderungen an den Algorithmus 33
3.1.5 Bewertung der Lösungsstrategien 33
3.1.5.1 Vollständiger Optimierungsalgorithmus 33
3.1.5.2 Heuristik 34
3.1.6 Einschränkungen 35

4 Lösungsansatz **37**
4.1 Kostenmodell 37
4.1.1 Kostenfaktoren für die Rekonfiguration 37
4.1.2 Verfahren zur Wiederherstellung von Dienstzuständen . 40
4.1.3 Entwurf des Kostenmodells 41
4.1.3.1 Anforderungen an das Kostenmodell 41
4.1.3.2 Einschränkungen des Kostenmodells 41
4.1.3.3 Kostenmodellierung 42
4.2 Rekonfigurationsheuristiken 45
4.2.1 Rekonfiguration ohne Beachtung von Kosteninformationen 46
4.2.2 Greedy Heuristik 47
4.2.3 Mehrfache Optimierung 48

5 Implementierung **51**
5.1 Erweiterungen am PCOM-Simulator 51
5.1.1 Ablauf einer Konfiguration/Rekonfiguration im PCOM-Simulator 51
5.1.2 Integration des Kostenmodells 55
5.2 Integration der Rekonfigurationsstrategien 56
5.2.1 Rekonfiguration ohne Beachtung von Kosteninformationen 56
5.2.2 Greedy Heuristik 56

6 Evaluation **63**
6.1 Vorüberlegungen 63
6.2 Bewertungskriterien 65
6.2.1 Verursachte Kosten 65
6.3 Szenarien 65
6.3.1 Parameter 65
6.3.2 1. Szenario 67
6.3.3 2. Szenario 70
6.4 Messergebnisse 72
6.4.1 Messergebnisse 1. Szenario 72

7 Zusammenfassung und Ausblick **79**
7.1 Zusammenfassung 79
7.2 Ausblick 80

Literaturverzeichnis **81**

Abbildungsverzeichnis

1 3PC Architektur 4
2 Reichweite der Rekonfigurationsstrategien 7
3 PCOM Architektur 11
4 Komponentenkomposition einer Anwendung 14
5 Backtracking-Richtung 16
6 Modell eines Active Space 17
7 Anwendungsframework Infrastruktur 18
8 Architektur von Aura 20
9 Aura Überblick 21
10 SpiderNet Dienstzusammensetzung Systemarchitektur 24
11 $2K^{Q+}$ Middlewarearchitektur 26
12 Rekonfiguration einer Anwendungskonfiguration 31
13 Komposition einer Anwendung 37
14 Verschicken der *berechneKosten*-Nachricht 43
15 Verschicken der *übermittleKosten*-Nachricht 43
16 Beispiel für die Kostenverteilung einer Anwendung 44
17 Rekonfiguration ohne Beachtung von Kosteninformationen . . 47
18 Greedy Rekonfigurationsstrategie 49
19 Simulationsablauf im PCOM-Simulator 52
20 Anpassung der Instanzenpositionen 54
21 Verbindungsaufbau 57
22 Unmodifizierte Greedy Heuristik 58
23 Modifizierte Greedy Heuristik 59
24 Suche nach einer Belegung 60
25 Greedy Heuristik Szenario 64
26 1. Szenario 67
27 Messergebnisse 2. Szenario 71

Tabellenverzeichnis

1 Messergebnisse 1. Szenario 69
2 Messergebnisse Szenario 1-1 72
3 Messergebnisse Szenario 1-2 73
4 Messergebnisse Szenario 1-3 73
5 Messergebnisse Szenario 1-4 74
6 Messergebnisse Szenario 1-5 74
7 Messergebnisse Szenario 1-6 75
8 Messergebnisse Szenario 1-7 75
9 Messergebnisse Szenario 1-8 76
10 Messergebnisse Szenario 1-9 76
11 Messergebnisse Szenario 1-10 77

1 Einleitung

1.1 Ubiquitous Computing

Die Vision von Ubiquitous Computing geht auf die Veröffentlichungen von Marc Weiser [Weiser], [Wei91], [Wei93] zurück. In seinen Veröffentlichungen beschreibt der Autor welche neuen Möglichkeiten sich für die Nutzer im Alltag einer ubiquitären Computerumgebung bieten. Seine Vision von Ubiquitous Computing beschreibt er mit den Worten:

> „Ubiquitous computing is the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user.“ [Wei93]

Laut Weiser wird sich die Art und Weise wie Nutzer mit Computern in Zukunft interagieren stark verändern. Ubiquitous Computing steht für die Kombination mehrerer Technologietrends:

- **Verschmelzung von Computern mit Alltagsgegenständen:**
  Ubiquitous Computing steht für die Allgegenwärtigkeit der Computer- und Informationstechnologie. Computer halten mehr und mehr Einzug in Alltagsgegenstände wie beispielsweise Kleidung oder Unterhaltungselektronik. Mit der Verschmelzung muss sich die Art und Weise der Interaktion zwischen dem Benutzer und den Geräten verändern, damit das System benutzbar bleibt.

- **Drahtlose Vernetzung der heterogenen Geräte untereinander:**
  Mit dem Einzug von Computern in Alltagsgegenständen ist der Benutzer in ubiquitären Umgebungen von einer Vielzahl an verschiedenen heterogenen Geräten umgeben. Die verschiedenen Geräte vernetzen sich spontan miteinander, um den Benutzer bei der Erledigung seiner Aufgaben zu unterstützen. Durch die Kooperation der Geräte innerhalb der Umgebung können möglicherweise neue Anwendungen angeboten werden und ungenutzte Ressourcen zur Verfügung gestellt werden.

- **Verwendung von Sensoren für neue Bedienkonzepte und die Erfassung von Kontextinformationen:**
  Mit Hilfe von Sensoren können die Geräte den Kontext des Benutzers (wie beispielsweise seine aktuelle Position oder Aktivitäten) erfassen, Informationen sammeln und die Bedienfreundlichkeit für den Benutzer weiter erhöhen.

- **Mobilität der Geräte:**
  Durch die mögliche Mobilität der Geräte entsteht eine relativ hohe Dynamik in ubiquitären Umgebungen. Benutzer bewegen sich mit Geräten zwischen verschiedenen Umgebungen und der Kontext, in welchem sich der Benutzer und die Geräte befindet, ändert sich stets.

1.2 Motivation

Ubiquitäre Umgebungen versprechen eine verbesserte Unterstützung der Benutzer bei ihrer täglichen Arbeit. Wie in Kapitel 1.1 bereits eingeführt besitzen ubiquitäre Umgebungen gewisse Merkmale wie beispielsweise drahtlose Vernetzung und Kooperation mehrerer Geräte. Für einen Anwendungsprogrammierer bieten ubiquitäre Umgebungen neue Möglichkeiten in der Anwendungsentwicklung. Beispielsweise können rechenintensive Teile einer Anwendung auf leistungsstärkere Computer verlagert werden oder die Ausgabe einer Anwendung dynamisch auf die in der Umgebung verfügbaren Monitore umgeleitet werden. Für einen Benutzer soll die Umgebung transparent erscheinen, d.h. er soll sich keine Gedanken über die Vernetzung oder Fähigkeiten der Geräte in der Umgebung machen müssen.

Die mögliche Dynamik in ubiquitären Umgebungen stellt dabei eine große Herausforderung dar. Die Geräte sind in der Regel drahtlos vernetzt und mobil, d.h. in ubiquitären Umgebungen herrscht eine Dynamik an verfügbaren Ressourcen. Es drängt sich die Frage auf, was mit verteilten Anwendungen geschieht, wenn benötigte Ressourcen in der Umgebung auf Grund von Dynamik nicht mehr verfügbar sind. Die Anpassung an sich verändernde Ressourcenverfügbarkeiten ist eine der zentralen Aufgaben im Ubiquitous Computing und die Möglichkeiten zur Anpassung einer Anwendung in einer Ubiquitous Computing Umgebung werden in dieser Diplomarbeit näher untersucht.

1.3 Aufgabenstellung

PCOM ist ein Komponentensystem für Ubiquitous Computing [BHSR04], welches in der Abteilung Verteilte Systeme an der Universität Stuttgart entwickelt wird. PCOM stellt ein Anwendungsmodell zur Verfügung, welches die Spezifikation von ubiquitären Anwendungen ermöglicht.

Im Rahmen dieser Diplomarbeit wird ein verteilter Algorithmus für PCOM weiterentwickelt, welcher es ermöglichen wird, bestehende Anwendungskon-

figurationen an sich veränderte Verfügbarkeiten von Ressourcen und Komponenten anzupassen. Neben dem Algorithmus sollen die Kosten für die Anpassung von Anwendungskonfigurationen näher untersucht werden und in die Anpassung der Anwendung mit einbezogen werden, so dass die Kosten für die Anpassung möglichst gering sind.

Eine detaillierte Analyse der Aufgabenstellung wird in Kapitel 3.1 vorgenommen.

1.4 Aufbau der Arbeit

Die weitere Struktur der Diplomarbeit gliedert sich folgendermaßen: In Kapitel 2 wird das Themengebiet genauer analysiert. Es wird auf benötigte Grundlagen von PCOM eingegangen, welche zum weiteren Verständnis der Diplomarbeit nötig sind. Außerdem werden verwandte Ansätze zur Rekonfiguration vorgestellt. Kapitel 3 beschäftigt sich mit der Analyse der Aufgabenstellung, Lösungsstrategien und den Anforderungen an den zu entwickelnden Algorithmus. Kapitel 4 widmet sich der Konzeption eines geeigneten Kostenmodells und des Algorithmus. In Kapitel 5 wird die Integration des Algorithmus in PCOM beschrieben. Kapitel 6 gibt einen Überblick über die Evaluation und Kapitel 7 gibt eine Zusammenfassung und einen Ausblick auf zukünftige Forschungsgebiete.

1.5 Einordnung der Diplomarbeit in 3PC

Das 3PC-Projekt [3PC] ist ein Forschungsprojekt der Universität Stuttgart, Abteilung Verteilte Systeme. Die Aufgabe des 3PC Projekt ist die Entwicklung minimaler, aber flexibler Systemunterstützung für adaptive, ubiquitäre Anwendungen. Das Projekt wird teilweise durch die Deutsche Forschungsgemeinschaft (DFG) und Daimler Chrysler Research gefördert. Abbildung 1 gibt einen Überblick über die Architektur von 3PC.

Grundlage der 3PC-Architektur bildet BASE [BSGR03], eine Middleware, welche automatische Adaption auf der Kommunikationsschicht unterstützt und dabei ressourcenschonend ist. BASE ist sehr flexibel in der Unterstützung verschiedener Geräte und Protokolle und erlaubt beispielsweise auch den Wechsel des Kommunikationsprotokolls zur Laufzeit.

PCOM [BHSR04] ist ein kleines, flexibles Komponentensystem für ubiquitäre Anwendungen. PCOM setzt auf der Middleware BASE auf. Eine Anwendung

in PCOM wird aus mehreren Komponenten zusammengesetzt, welche über verschiedenen Geräte hinweg verteilt sein können. Auf benötigte Grundlagen zu PCOM wird in Kapitel 2.2 genauer eingegangen.

Innerhalb des 3PC-Projekt liegt der Schwerpunkt der Forschung zur Zeit in den folgenden Gebieten. SANDMAN untersucht die Möglichkeiten von Service Discovery in Bezug auf Energieeffizienz. G-SPAN beschäftigt sich mit graph-basierten Sicherheitsprotokollen für sichere Kommunikation in Ad-hoc-Netzwerken. Diese Diplomarbeit ist im Gebiet CAPA eingeordnet. CAPA untersucht die automatische Konfiguration von komponentenbasierten Anwendungen in PCOM.

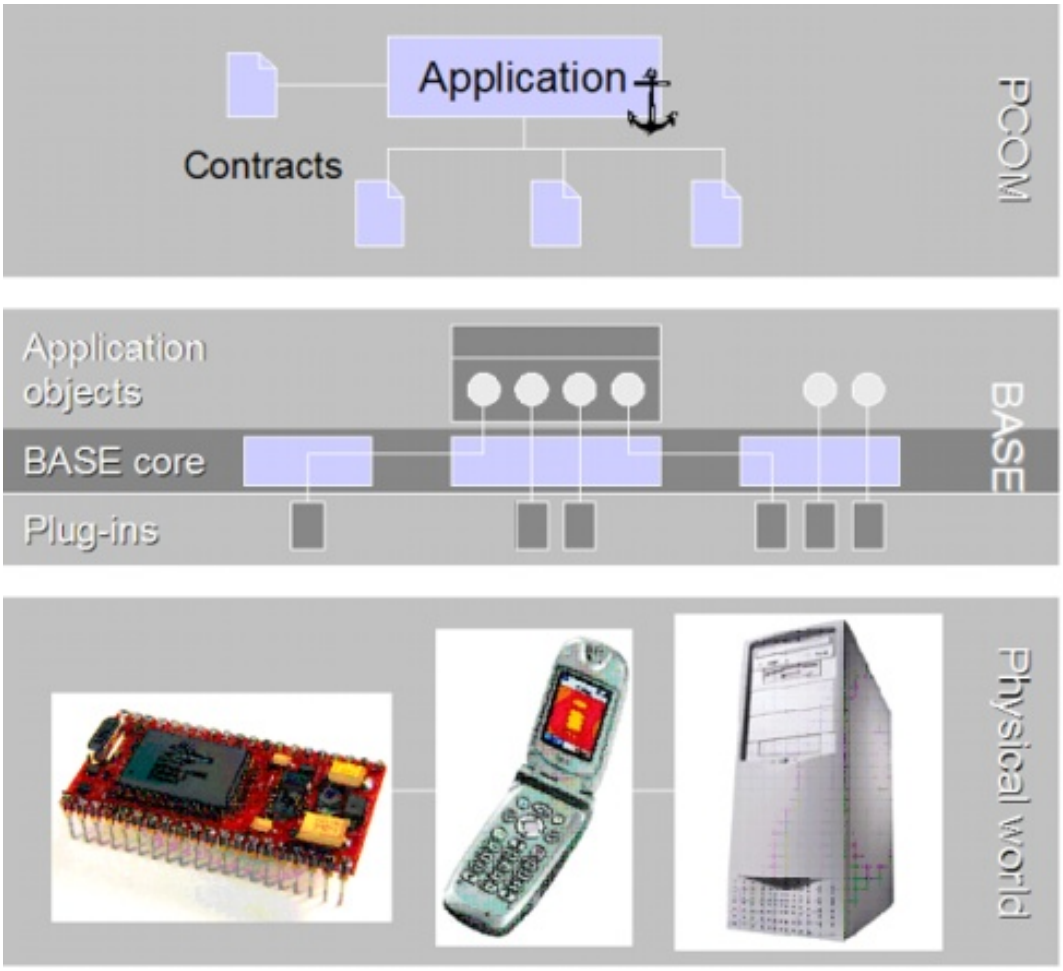


Abbildung 1: 3PC Architektur [3PC]

2 Grundlagen

2.1 Rekonfiguration

Ein wichtiges Merkmal von ubiquitären Umgebungen ist die mögliche Mobilität einzelner Geräte und die damit verbundene variierende Anzahl von Geräten in der Umgebung. Die Folge davon ist, dass die verfügbaren Ressourcen innerhalb der ubiquitären Umgebung Schwankungen ausgesetzt sind. Der Ressourcenbegriff wird innerhalb dieser Diplomarbeit sehr weit gefasst, d.h. unter einer Ressource versteht man nicht nur verfügbaren Speicherplatz oder Rechenleistung, sondern auch komplexere Dienste/Komponenten, welche von Geräten angeboten werden und auch periphere Geräte wie Ein- und Ausgabemedien, z.B. ein Monitor. Die mögliche Dynamik in ubiquitären Umgebungen erhöht die Komplexität für die Entwicklung von Anwendungen, welche auf mehreren Geräten gleichzeitig verteilt ausgeführt werden. Anwendungen in ubiquitären Umgebungen müssen in der Lage sein, sich während ihrer Ausführung an Veränderungen in der Umgebung anzupassen oder durch die ubiquitäre Umgebung angepasst zu werden.

Im Folgenden werden verschiedene Aspekte der Rekonfiguration kurz vorgestellt. Es wird zwischen Ursache, Ort, Art und Zeitpunkt für die Rekonfiguration unterschieden.

2.1.1 Ursachen für die Rekonfiguration

Eine Anpassung oder auch Rekonfiguration wird in der Regel dann nötig, wenn sich die Verfügbarkeit von Ressourcen in der Umgebung verändert. Die Ursachen für solche Veränderungen lassen sich in folgende Klassen einteilen:

- **Wegfall von Ressourcen in der Umgebung:**
  Der Wegfall von Ressourcen in der Umgebung kann verschiedene Gründe als Ursache haben, welche im Folgenden kurz vorgestellt und diskutiert werden.
  - **Mobilität und Charakteristika drahtloser Übertragung:**
    In ubiquitären Umgebungen befinden sich eine Vielzahl an verschiedenen Ressourcen, welche entweder stationär oder mobil sind. Mobile Ressourcen sind in der Regel über drahtlose Netzwerke mit anderen Ressourcen vernetzt. Auf Grund von Mobilität besteht daher die Möglichkeit, dass eine verwendete Ressource sich

nicht mehr im Empfangsbereich der anderen Anwendungsressourcen befindet, weil beispielsweise ein Benutzer mit einem Gerät die Umgebung verlassen hat. Weiterhin gilt, dass die drahtlose Übertragung durch mehrere Faktoren wie Interferenzen oder Abschattung negativ beeinflusst werden kann. In Folge von Mobilität und Übertragungsproblemen auf dem drahtlosen Kanal entstehen möglicherweise Netzwerkpartitionen und es muss eine Rekonfiguration betroffener Anwendungen durchgeführt werden.

  - **Systemfehler:**
    Ressourcen sind nicht mehr verfügbar, wenn sie auf Grund eines Systemfehlers in ihrer korrekten Ausführung unterbrochen werden.

  - **Abschaltung:**
    Ressourcen werden möglicherweise durch den Anwender oder das System abgeschaltet. Beispielsweise könnte eine Ressource auf Grund von einem niedrigen Energieniveau in der Batterie in den Stromsparmodus wechseln und für die Anwendung nicht mehr zur Verfügung stehen.

  - **Wegfall auf Grund von Ressourcenumverteilung:**
    Anwendungen konkurrieren um die in der Umgebung verfügbaren Ressourcen. Die verwendeten Ressourcen einer Anwendung werden möglicherweise einer anderen Anwendung zugeteilt, da diese eine höhere Priorität besitzt.

- **Verfügbarkeit neuer Ressourcen in der Umgebung:**
  Eine Rekonfiguration von Anwendungen kann allerdings (abhängig von der Rekonfigurationsstrategie) auch durch die Verfügbarkeit von neuen Ressourcen angestoßen werden. In diesem Fall wird eine Anwendung möglicherweise rekonfiguriert, weil die neu verfügbaren Ressourcen einen größeren Nutzen für die Anwendung und den Anwender versprechen.

  - **Mobilität und Charakteristika drahtloser Übertragung:**
    Mobilität kann neben dem Wegfall von Ressourcen auch der Grund für die Verfügbarkeit von neuen Ressourcen in der Umgebung sein, wenn z.B. ein Benutzer mit seinen Geräten eine Umgebung (den Empfangsbereich der anderen Ressourcen) betritt. Ebenso werden Geräte in der Umgebung wieder verfügbar, wenn Ereignisse wie Interferenz oder Abschattung nicht mehr auftreten.

- **Veränderung des Ressourcenbedarfs:**
  Wenn eine bestehende Anwendung in der Umgebung ihre Ausführung beendet oder ihr Ressourcenbedarf sich verkleinert, dann werden in der Umgebung die betroffenen Ressourcen freigegeben und sind für andere Anwendungen nutzbar.
- **Hochfahren:**
  Wenn Ressourcen in der Umgebung neu hochfahren/starten, werden sie in der Umgebung neu verfügbar.

2.1.2 Ort der Rekonfiguration

In [Sat96] wurde unter anderem untersucht, in welcher Reichweite sich die möglichen Rekonfigurationsstrategien bewegen können. Laut [Sat96] kann die Rekonfiguration an verschiedenen Orten erfolgen. Die Reichweite der möglichen Rekonfigurationsstrategien bewegt sich zwischen zwei Extremen, siehe Abbildung 2.

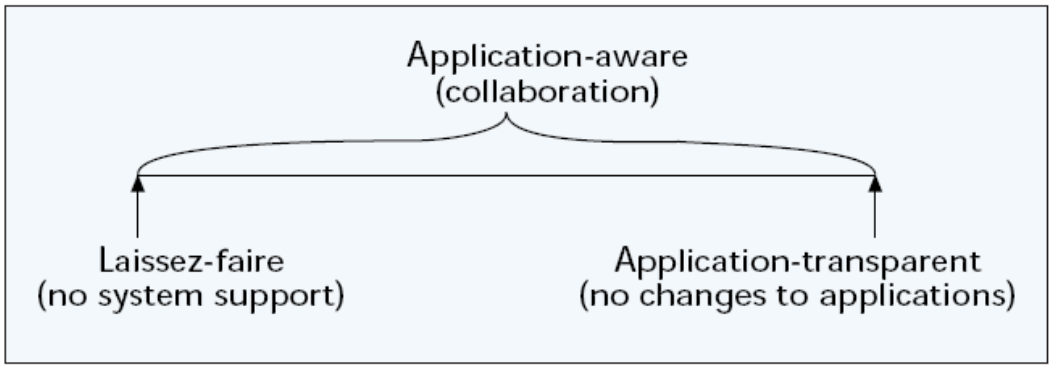


Abbildung 2: Reichweite der Rekonfigurationsstrategien [Sat96]

Auf der linken Seite der Abbildung wird die Rekonfigurationsstrategie vollständig und isoliert durch die einzelne Anwendung gesteuert (Laissez-faire). Die Anwendung kann die Entscheidung über die Rekonfiguration beispielsweise an den Benutzer weitergeben und dieser wählt dann unter einer Auswahl an Rekonfigurationsmöglichkeiten eine Konfiguration aktiv aus. Je nach Dynamik entsteht für den Benutzer ein nicht unerheblicher zusätzlicher Aufwand für die Anpassung der Anwendung. Anstatt den Endbenutzer in die Anpassung der Anwendung einzubeziehen, kann die Anpassungsentscheidung auch in der Anwendung selbst getroffen werden. Die Behandlung der Anpassung an Veränderungen in der Umgebung erhöht die Komplexität in der

Anwendungsentwicklung erheblich. Von Nachteil ist, dass für die Rekonfiguration keine Unterstützung durch das System in Anspruch genommen wird und daher die Rekonfigurationsstrategie nicht mit anderen Anwendungen in dem System abgestimmt werden kann.

Auf der anderen Seite der Abbildung findet die Rekonfiguration transparent für die Anwendungen innerhalb des System statt. Das hat den Vorteil, dass die Anwendungen vollständig isoliert sind von der Rekonfiguration. Der Anwendungsprogrammierer muss keine Unterstützung für die Rekonfiguration in die Anwendung integrieren, was die Entwicklung von Anwendungen erheblich vereinfacht. Es besteht allerdings die Möglichkeit, dass die systemweite Rekonfiguration möglicherweise für einige Anwendungen ungeeignete Rekonfigurationen vornimmt.

Zwischen diesen zwei Extremen liegt eine Vielzahl an Möglichkeiten für die systemunterstützte Anwendungsrekonfiguration.

2.1.3 Art der Rekonfiguration

Es gibt verschiedene Möglichkeiten eine Anwendung an sich verändernde Voraussetzungen/Umgebungen anzupassen.

- **Parametrische Rekonfiguration:**
  Die parametrische Rekonfiguration ändert anwendungsspezifische Parameter zur Anpassung an die Umgebung. Beispielsweise kann eine Videostreamanwendung die Datenrate bei der Übertragung dynamisch an die verfügbare Bandbreite anpassen. Die Anwendung bleibt in ihrer Zusammensetzung erhalten und der Anwendungsentwickler kann beispielsweise mehrere Anforderungsprofile in der Anwendung definieren: Minimumanforderungen, Optimalanforderungen. Die Anwendung wird erst strukturell rekonfiguriert, wenn die Minimumanforderungen nicht mehr erfüllt werden können.

- **Strukturelle Rekonfiguration:**
  Die strukturelle Rekonfiguration verändert nicht Anwendungsparameter, sondern ersetzt Anwendungsteile, welche nicht mehr verfügbar sind oder nicht die angeforderte Dienstgüte anbieten können. Die Anforderungen der Anwendung bleiben hierbei im Gegensatz zur parametrischen Rekonfiguration fest, aber die Komposition der Anwendungskomponenten verändert sich.

2.1.4 Zeitpunkt der Rekonfiguration

Es gibt verschiedene Zeitpunkte wann eine Rekonfiguration erfolgen soll. Im Folgenden wird zwischen proaktiver und reaktiver Refkonfiguration unterschieden.

- **Proaktive Rekonfiguration:**
  Die proaktive Rekonfiguration verfolgt eine Open-Loop-Strategie, d.h. es wird versucht die Rekonfiguration im Fehlerfall zu vermeiden, indem die Anwendung vorsorglich rekonfiguriert wird. Dazu werden bei der proaktiven Rekonfiguration die Ressourcen in der Umgebung permanent überwacht. Beispielsweise kann eine Komponente vorsorglich ersetzt werden, wenn die Signalstärke einen gewissen kritischen Wert unterschreitet und sich daher die Komponente mit einer relativ hohen Wahrscheinlichkeit bald nicht mehr in Reichweite befindet.

- **Reaktive Rekonfiguration:**
  Die reaktive Rekonfiguration verfolgt eine Closed-Loop-Strategie, d.h. es wird erst eine Rekonfiguration durchgeführt nachdem die Anwendung auf Grund eines Fehlers nicht mehr ausgeführt werden kann. Die Anwendung wird angehalten und es wird versucht, eine neue gültige Konfiguration für die Anwendung zu finden und den letzten konsistenten Anwendungszustand wiederherzustellen.

2.2 PCOM Grundlagen

In diesem Kapitel wird die komponentenbasierte Middleware PCOM näher vorgestellt. Zuerst wird ein Überblick über die PCOM Architektur und das Anwendungsmodell gegeben und dann näher auf die Konfiguration von Anwendungen eingegangen.

2.2.1 PCOM Anwendungsmodell

In PCOM bestehen Anwendungen aus einer Komposition von Komponenten, die miteinander interagieren. Die Komponenten sind atomar in ihrer Verteilung. Eine Komponente benötigt allerdings möglicherweise weitere Komponenten um einen Dienst anzubieten. Es ist ein breites Spektrum von einfachen bis hin zu komplexen Anwendungen denkbar.

Zwischen den Komponenten bestehen explizite Abhängigkeiten, welche durch

Verträge beschrieben werden. Jede Komponente besitzt einen Vertrag, welcher aus folgenden Bestandteilen besteht:

- **Lokale Anforderungen:**
  Im Anforderungsteil wird spezifiziert, welche Anforderungen die Komponente an die Plattform hat, auf der die Komponente gestartet wird.
- **Angebotene Dienste:**
  Hier werden die Dienste spezifiziert, welche die Komponente anbietet.
- **Abhängigkeiten:**
  Um einen Dienst anzubieten ist die Komponente möglicherweise abhängig von den Diensten anderer Komponenten. Diese Abhängigkeiten werden ebenfalls im Vertrag spezifiziert.
- **Implementierung:**
  In diesem Abschnitt werden Informationen über die Implementierung abgelegt, die vom Container, welcher die Komponente instanziiert, benutzt wird.

2.2.2 PCOM Architektur

Abbildung 3 gibt einen Überblick über die PCOM Architektur. PCOM baut auf der Middleware BASE auf. Der PCOM Container verwaltet die lokalen Komponenteninstanzen, die Abhängigkeiten der Komponenten untereinander und kommuniziert mit anderen Containern. Jeder Container gibt über das Remote Container Interface seine lokal verfügbaren Komponenten bekannt. Der Container managt außerdem die Vertragsverhandlungen zur Instanziierung und Zugang von entfernten Komponenten in anderen Containern.

Anwendungen in PCOM sind in der Regel aus mehreren Komponenten zusammengesetzt. Der Konfigurationsalgorithmus aus [Rei04] sucht eine gültige Konfiguration für die Anwendung. Das Problem der Suche nach einer gültigen Konfiguration wird dabei als Constraint Satisfaction Problem betrachtet. Im Folgenden wird das Constraint Satisfaction Problem näher vorgestellt.

2.2.3 Distributed Constraint Satisfaction Problem

Die Suche nach einer gültigen Konfiguration, welche alle Ressourcenbeschränkungen und Komponentenabhängigkeiten in Betracht zieht, ist ein NP-vollständiges Problem. In der Diplomarbeit von Michael Reinsch wurde dieses

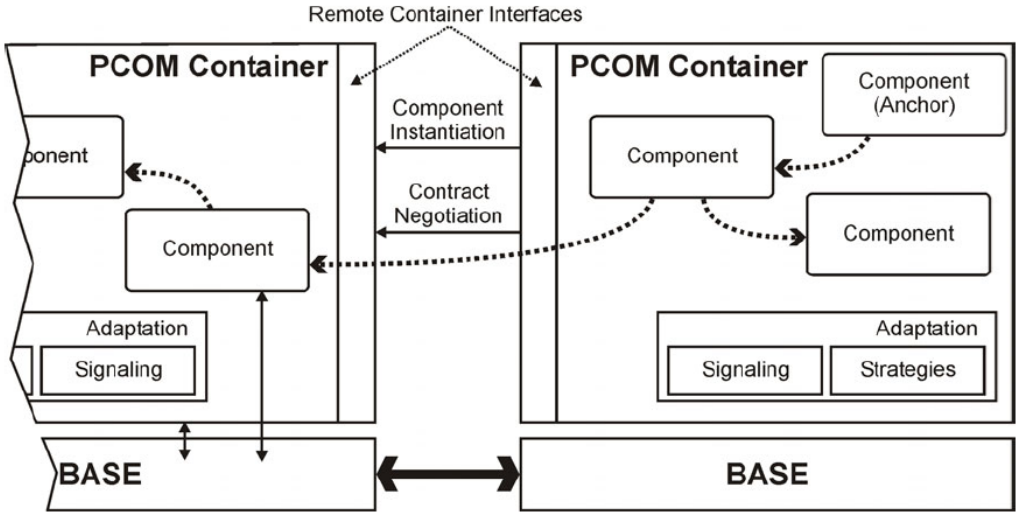


Abbildung 3: PCOM Architektur [BHSR04]

NP-vollständige Problem als Distributed Constraint Satisfaction Problem betrachtet. Das Distributed Constraint Satisfaction Problem (nachfolgend kurz DCSP) ist formal folgendermaßen definiert [YDIK98]:

Ein DCSP besteht auf n Variablen x_1, x_2 , ..., x_n, mit diskreten, endlichen Wertebereichen D_1, D_2, ..., D_n. Weiterhin existiert ein Satz an Einschränkungen, welche die möglichen Werte für die einzelnen Variablenbelegungen einschränken. Eine Einschränkung ist als Prädikat $p_k(x_{k1}, ..., x_{kj})$ über dem kartesischen Produkt der entsprechenden Wertebereiche $D_{k1} \times ... \times D_{kj}$ definiert. Die einzelnen Prädikate sind über eine beliebige Teilmenge der Variablen definierbar und werden dann zu *wahr* ausgewertet, wenn die entsprechende Variablenbelegung die Bedingung erfüllt. Die Variablen und Einschränkungen sind auf mehrere Agenten verteilt und die Agenten kommunizieren untereinander über Nachrichten, welche in endlicher Zeit ausgeliefert werden. Zur Lösung des DCSP muss eine entsprechende Variablenbelegung gefunden werden, welche alle Einschränkungen gleichzeitig erfüllt. In [YDIK98] wird ein Algorithmus zur Lösung des DCSP vorgestellt, welcher in abgewandelter Form zur initialen Konfiguration von Anwendungen in PCOM eingesetzt wird.

2.2.4 Asynchroner Backtracking Algorithmus

Der Algorithmus benutzt asynchrones Backtracking um eine Lösung für das DCSP zu suchen und läuft verteilt auf den einzelnen Agenten. O.B.d.A wird angenommen, dass jeder Agent nur eine Variable hält und möglicherweise ein oder mehrere Einschränkungen für diese Variable hat. Das DCSP kann als Graph dargestellt werden, wobei jeder Agent mit seiner Variable als Knoten dargestellt wird und jede Einschränkung als gerichtete Kante zwischen zwei Agenten dargestellt wird. Zwischen den einzelnen Agenten wird eine totale Ordnung eingeführt um unter anderem Zyklen im Graph zu unterbinden und die Backtrackingrichtung zu ermitteln. Jeder Agent hat eine einzigartige ID, welche seine Priorisierung innerhalb des Graphen repräsentiert. Die Kommunikation der Agenten untereinander findet über Nachrichten statt:

- OK?-Nachricht: Versand der gewählten Variablenbelegung an einen Nachbaragenten.
- noGood-Nachricht: Ablehnung des gewählten Variablenwertes. Mitsenden der Variablenwerte, welche im Konflikt stehen.

Zum Start des Algorithmus wählt jeder Agent parallel einen Startwert für seine Variable und versendet diese mit einer OK?-Nachricht an seine entsprechenden Nachbarn (die Nachbarn, welche über ausgehende Kanten verbunden sind).

Empfängt ein Agent eine OK?-Nachricht, dann übernimmt er die gewählte Variablenbelegung des anderen Agenten und speichert diese in seiner Sicht zwischen. Die Sicht repräsentiert die aktuellen Variablenbelegungen der anderen Agenten, welche über eingehende Kanten mit dem Agenten verbunden sind. Nach dem Empfang der OK?-Nachricht wird überprüft, ob die eigene gewählte Variablenbelegung in Konflikt mit der Variablenbelegung des anderen Agenten steht. Wenn ein Konflikt besteht, dann versucht der Agent lokal einen anderen Variablenwert zu wählen. Falls dies erfolgreich ist, dann verschickt dieser Agent seinen neu gewählten Variablenwert über OK?-Nachrichten an seine entsprechenden Nachbarn. Ist es dem Agenten nicht möglich einen neuen Variablenwert zu finden, dann muss ein Backtracking gestartet werden und der Agent sendet eine noGood-Nachricht an den Agenten mit der niedrigsten Priorität mit dessen Variablenbelegung er im Konflikt steht (wichtig für die Backtrackingrichtung). Zusammen mit der noGood-Nachricht wird eine Liste aller Agenten und deren Variablenbelegung mitgeschickt, welche aus der Sicht des Agenten Bestandteil des Konflikts mit der Einschränkung sind. Nach dem Versand der noGood-Nachricht entfernt der

Agent die Variablenbelegung des entsprechenden Agenten aus seiner Sicht.

Nach dem Empfang einer noGood-Nachricht überprüft der Agent, ob die übermittelte Liste an Variablenbelegungen mit der ihm eigenen lokalen Sicht kompatibel ist. Eine Sicht ist dann kompatibel, wenn die übermittelten Variablenwerte mit den Variablenwerten der lokalen Sicht des Agenten übereinstimmen, falls dieser die entsprechenden Agentenvariablen in seiner Sicht hat. Falls die noGood-Nachricht nicht kompatibel zu der lokalen Sicht ist, dann kann die Nachricht verworfen werden, weil sie veraltet ist. Im Fall einer kompatiblen (aktuellen) noGood-Nachricht muss der Agent versuchen den Konflikt aufzulösen indem er einen neuen Variablenwert wählt (und ihn ggf. mit OK?-Nachrichten dann wieder verteilt). Ist dies nicht möglich, dann muss der Konflikt an den nächsthöheren Agenten weitergegeben werden. Unter Umständen hat der Agent jedoch keine eingehende Kante (Einschränkung) zu dem nächsthöheren Agenten, daher muss zuerst eine Kante (Einschränkung) zwischen diesen beiden Agenten aufgebaut werden und danach wird die noGood-Nachricht mit der Konfliktliste an den Agenten weitergesendet. Die Konfliktliste enthält alle Konflikte außer dem Konflikt des sendenden Agenten.

Da zwischen den Agenten eine totale Ordnung gilt und neue Einschränkungen nur zwischen Agenten höherer ID $\rightarrow$ niedrigerer ID erzeugt werden, bleibt der Graph weiterhin zyklenfrei. Wenn der Agent mit der höchsten ID eine noGood-Nachricht empfängt und er schon alle möglichen Werte aus dem Wertebereich für seine Variable durchprobiert hat, dann terminiert der Algorithmus mit dem Ergebnis, dass es keine Lösung für das DCSP gibt. Wenn es eine Lösung für das DCSP gibt, dann werden nach endlicher Zeit keine Nachrichten mehr zwischen den Agenten verschickt. Dies kann über ein Termination Detection Protokoll festgestellt werden.

2.2.5 Finden einer initialen Konfiguration in PCOM

Der weiter oben beschriebene asynchrone Backtracking-Algorithmus wird in PCOM zur initialen Konfiguration von Anwendungen eingesetzt. Der vollständige Algorithmus ist unter [HBR05] einzusehen. Einige wichtige Details werden hier vorgestellt. Der Algorithmus unterscheidet sich in einigen Punkten von dem allgemeinen asynchronen Backtracking-Algorithmus, da für das Finden einer gültigen Konfiguration in PCOM in der Regel nur eine partielle Lösung erforderlich ist. Das bedeutet, dass möglicherweise nicht alle Komponenten Teil der Konfiguration sind, daher sind die Zustände der unbeteiligten Komponenten nicht relevant für eine gültige Konfiguration. Weiterhin startet

die Suche nach einer Konfiguration nicht in allen Komponenten gleichzeitig, sondern wird von einem Applikations-Anker gestartet und breitet sich stufig aus. Die folgende Abbildung zeigt eine Beispielanwendung, welche aus 7 Komponenten besteht, siehe Abbildung 4.

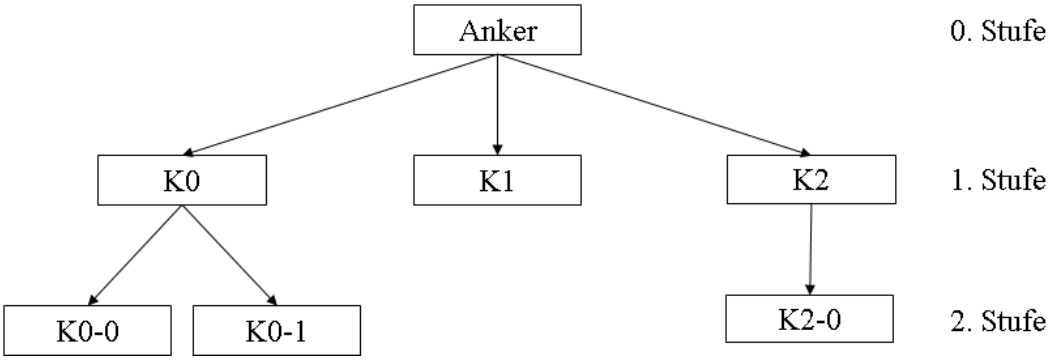


Abbildung 4: Komponentenkomposition einer Anwendung

Wenn der Algorithmus zur Suche nach einer initialen Konfiguration gestartet wird, dann wird zuerst die Ankerkomponente initialisiert und festgestellt, dass die Ankerkomponente 3 weitere Komponenten (K0, K1, K2) benötigt. Der Applikationsanker wählt in der ersten Stufe für jede Komponentenabhängigkeit einen Kandidaten und verschickt an diesen eine Update-Nachricht. Wenn die gewählten Komponenten weitere Komponenten benötigen wie z.B. die Komponente K0, dann werden in der nächsten Stufe an die entsprechenden Kandidaten wieder Update-Nachrichten verschickt (K0 schickt eine Update-Nachricht an K0-0 und K0-1, K2 schickt eine Update-Nachricht an K2-0).

Für die initiale Konfiguration gibt es zwei Klassen von Einschränkungen, welche im Vertrag der Komponente aufgeschlüsselt sind:

- **Strukturelle Einschränkungen:**
  Hier wird im Vertrag spezifiziert, ob zur Instanziierung der Komponente noch die Instanziierung von anderen Komponenten nötig ist.
- **Ressourceneinschränkungen:**
  Jede Komponente hat einen Bedarf an bestimmten Resourcen, welche im Vertrag der Komponente spezifiziert sind.

Jede Komponenteninstanz darf nur dann instanziiert werden, wenn die oben genannten Einschränkungen erfüllt werden können, d.h. es stehen genug Ressourcen zur Verfügung und es können auch alle anderen benötigten Komponenteninstanzen instanziiert werden.

Diese zwei Klassen von Einschränkungen müssen nun als Einschränkungen zwischen Variablenbelegungen modelliert werden, so dass das Finden einer initialen Konfiguration als DCSP betrachtet werden kann. Strukturelle Einschränkungen werden folgendermaßen repräsentiert: Jede Komponenteninstanz kann durch einen mehrdimensionalen Vektor repräsentiert werden. Jede Dimension in diesem Vektor repräsentiert eine Abhängigkeit zu einer anderen Komponenteninstanz. Der Wertebereich der einzelnen Dimensionen richtet sich nach der Anzahl der verfügbaren Komponenteninstanzen, welche für eine Instanziierung in Frage kommen können. Da eine Komponenteninstanz möglicherweise nicht Teil einer Konfiguration sein kann, wird zusätzlich noch ein Null-Wert $\oslash$ in den Wertebereich jeder Dimension eingefügt. Dieser Null-Wert wird gewählt, wenn die Komponenteninstanz nicht von einer anderen Instanz instanziiert wird. Für die einzelnen Werte innerhalb eines Wertebereiches der einzelnen Dimensionen des Vektors muss eine feste Abarbeitungsreihenfolge herrschen, damit asynchrones Backtracking angewendet werden kann (zuerst Kandidat 1, dann $2, ..., n$).

Ressourceneinschränkungen werden folgendermaßen repräsentiert: Jede Komponenteninstanz hat einen Ressourcevektor vom Typ Integer, welcher die benötigten Ressourcen darstellt. Jede Dimension in dem Vektor repräsentiert eine benötigte Ressource und der aktuelle Wert gibt die benötigte Menge an. Die Werte des Vektor können sich zur Laufzeit verändern, da sich die Anforderungen an die Instanz zur Laufzeit verändern können. Die Summe der Komponenteninstanzvektoren muss zu jeder Zeit kleiner (oder gleich) der auf dem Geräte verfügbaren Ressourcen sein. Für jedes Geräte wird daher ein Vektor vom Typ Integer gehalten, welcher die auf dem Gerät verfügbaren Ressourcen darstellt.

Wenn während der Suche nach einer Konfiguration ein Konflikt auftritt, dann wird dieser durch asynchrones Backtracking aufgelöst. Die Ordnung gibt dann die Backtracking-Richtung an. Es gibt mehrere Möglichkeiten eine totale Ordnung für den Algorithmus aufzubauen, siehe Abbildung 5.

Die aktuelle Implementierung des Algorithmus in PCOM verfolgt die linke Backtracking-Strategie. Die Idee dahinter ist, dass niedrigere Komponenten in der Regel weniger rekursive Abhängigkeiten haben als Komponenten weiter oben in dem Anwendungsbaum, d.h. der Konflikt wird möglicherweise durch weniger Update-Nachrichten aufgelöst. Es sind allerdings auch Szenarien denkbar, wo diese Strategie schlechter abschneidet als beispielsweise die rechte Backtracking-Strategie.

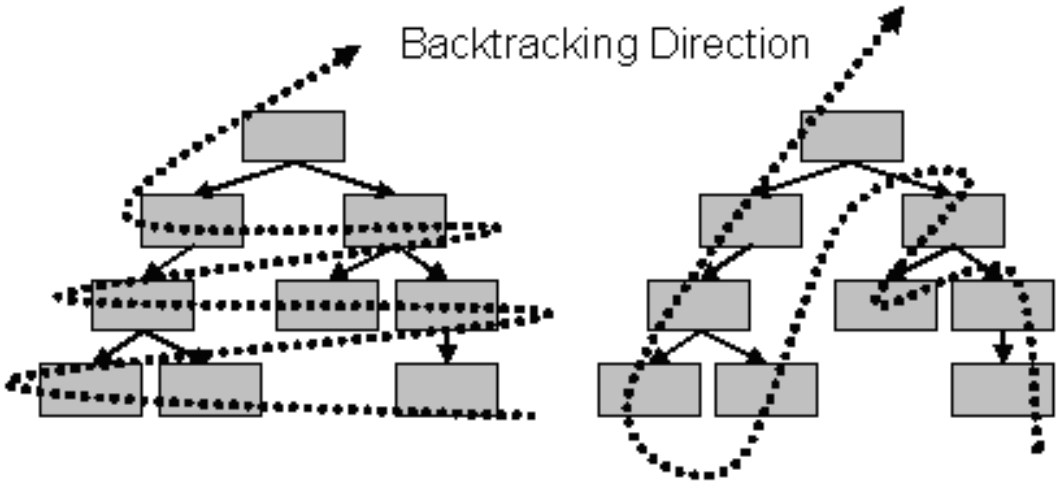


Abbildung 5: Backtracking-Richtung [HBR05]

2.3 Verwandte Arbeiten

2.3.1 Klassische komponentenbasierte Verteilungsinfrastrukturen

Klassische komponentenbasierte Verteilungsinfrastrukturen wie beispielsweise Corba Components 3.0 [Omg02] oder Enterprise JavaBeans [Sun05] modellieren Anwendungen als eine Komposition von Softwarekomponenten. Der Begriff der Softwarekomponente wird in der Literatur sehr unterschiedlich definiert. Eine mögliche Definition wird in [Szy02] gegeben. Softwarekomponenten besitzen in der Regel die Blackboxeigenschaft, d.h. sie bieten nach außen wohldefinierte Dienste über spezifizierte Schnittstellen an und haben keine impliziten Abhängigkeiten zu anderen Komponenten. Weiterhin wird die interne Repräsentation der Daten und die Implementierung nach außen verborgen. Die klassischen Verteilungsinfrastrukturen bieten keine automatische Rekonfiguration und Unterstützung für ubiquitäre Umgebungen, daher werden diese im Rahmen dieser Diplomarbeit nicht näher betrachtet.

2.3.2 Ubiquitäre Middleware

Es gibt mittlerweile eine Vielzahl an unterschiedlichen ubiquitären Verteilungsinfrastrukturen wie beispielsweise Gaia [RHC+02], Aura [GSS+02] und One.World [GDL+01]. Im Folgenden werden stellvertretend zwei ubiquitäre Verteilungsinfrastrukturen vorgestellt, welche beide Ubiquitous Computing

mit Hilfe von Smart Environments unterstützen.

2.3.2.1 Gaia Gaia ist eine Middleware-Betriebssystem für Ubiquitous Computing [RHC+02]. In Gaia werden so genannte Active Spaces für verteilte Anwendungen zur Verfügung gestellt. Ein Active Space modelliert einen physikalischen Raum mit fest definierten Grenzen und physischen Objekten. Innerhalb dieses Active Space können Benutzer mit Anwendungen interagieren, welche kontextbezogene Informationen des Active Space mit einbeziehen können. Abbildung 6 zeigt das Modell eines Active Space.

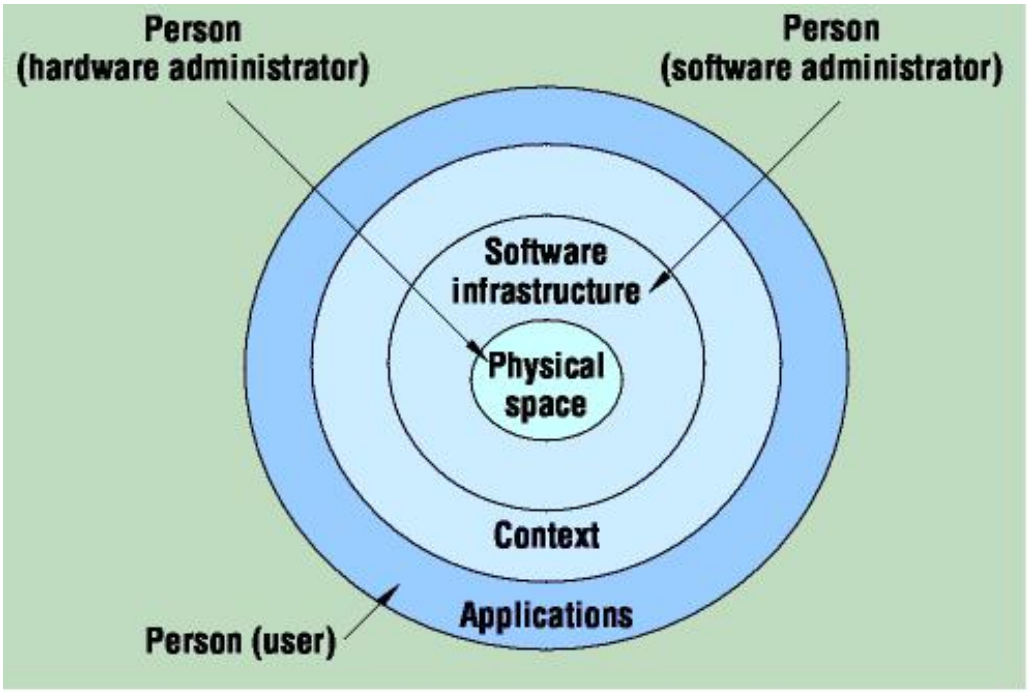


Abbildung 6: Modell eines Active Space [RHC+02]

In einem Active Space gibt es statische, leistungsfähige Geräte (PCs, Bildschirme,...) und mobile, leistungsschwächere Geräte (Laptops, PDAs,...). Die statischen, leistungsfähigen Geräte bilden eine Infrastruktur für den Active Space. Die Geräte arbeiten mit dem GaiaOS, welches verschiedene Dienste anbietet, wie beispielsweise das Space Repository, in dem alle in dem Active Space verfügbaren Komponenten aufgelistet sind. Mobile Geräte können sich

zwischen Active Spaces bewegen. Beim Betreten eines Active Space werden die Userdaten und Anwendungen durch Gaia dynamisch auf die in dem Active Space verfügbaren Geräte verteilt. In Active Space Anwendungen wird zwischen Anwendungssemantik und der Zusammensetzung der Anwendung durch einzelne Komponenten unterschieden. Es gibt nur Bindungen zwischen dem Koordinator und den Anwendungskomponenten, d.h. es ist nicht möglich, dass eine Komponente zur Erfüllung ihrer Aufgabe weitere Komponenten mit einbindet. Anwendungen werden nach dem „Modell, Präsentation, Kontroller, Input Sensor und Koordinator"-Modell aufgebaut, siehe Abbildung 7.

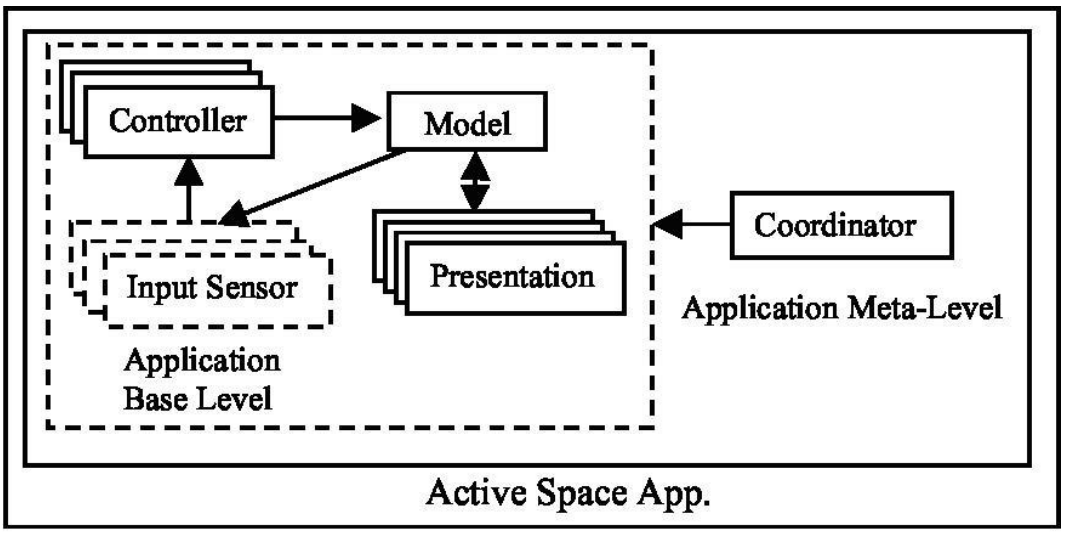


Abbildung 7: Anwendungsframework Infrastruktur [RHC02]

Das Modell ist die einzige Komponente auf dem Application Base Level, welche den Zustand der Anwendung hält. Input Sensor, Präsentation und Kontroller sind zustandslos. Eine Ersetzung dieser Komponenten ist sehr einfach, da einfach nur eine neue passende Komponente eingebunden werden muss. Der Koordinator hält Informationen über die Zusammensetzung der einzelnen Komponenten (Meta-Level). Kritische Anwendungsteile wie der Koordinator und das Modell der Anwendung werden in der Infrastruktur gehalten. Es wird angenommen, dass die Infrastruktur in dem Active Space immer vorhanden ist und ausfallsicher ist.

Um die Mobilität eines Benutzers zu gewährleisten muss eine Anwendung zur Laufzeit rekonfigurierbar sein. Es wird zwischen Intra-Space Mobility und Inter-Space Mobility unterschieden [RHC02]. Unter Intra-Space Mobil-

ity versteht man den Wechsel von interaktiven Anwendungskomponenten innerhalb eines Active Space wie z.B. E/A Komponenten. Es wird dabei angenommen, dass der Koordinator, das Modell und die Kontroller fest in dem Active Space stehen (ein Wechsel dieser Komponenten fällt dann unter Inter-Space Mobility). Der Koordinator sorgt zusammen mit dem application managment interface (Bestandteil des GaiaOS) dafür, dass die entsprechenden Komponenten für die Anwendung ausgewählt werden. Der Wechsel einer Komponente muss vom Benutzer der Anwendung unterstützt werden, d.h. der Benutzer wählt die neue Komponente aus einer Liste verfügbarer Komponenten aus.

Unter Inter-Space Mobility versteht man den Wechsel zwischen zwei Active Spaces. Wenn der Anwender einen Active Space verlässt, dann soll die Anwendung mit dem Benutzer in den anderen Active Space wechseln. Es wird in diesem Fall zwischen zwei Arten von Mobilität unterschieden: interactive und full. Im Interactive-Modus dürfen nur interaktive Komponenten der Anwendung in den anderen Active Space umziehen. Eine Beispielanwendung dafür wäre z.B. die Überwachung von Sensordaten eines Active Space von einem anderen Active Space aus. Der Full-Modus wird in der Regel für Anwendungen benutzt, welche um den Benutzer herum aufgebaut werden und den Kontext im Umfeld des Benutzers mit einbeziehen. In diesem Modus werden alle Anwendungskomponenten zusammen mit dem Benutzer vom einem Active Space in den anderen umgezogen. Die Sicherung des Modell-Zustand ist anwendungsabhängig und muss vom Anwendungsentwickler unterstützt werden (saveState-Methode). Die Daten der Anwendung werden dann in einen persistenten Benutzerspeicher geschrieben, welcher in den Active Spaces gemountet ist (Gaia Kontextfilesystem). Die aktuelle Komposition der Komponenten wird als ACD (application customized description) ebenfalls dort gespeichert und die Anwendung danach angehalten. Wenn der Benutzer den anderen Active Space betritt, wird die Anwendung wiederhergestellt. Da in dem anderen Active Space möglicherweise nicht die gleichen Komponenten zur Verfügung stehen, wird mit Hilfe der ACD und der AGD (application generic description) eine neue Komposition für die Anwendung ermittelt.

Da eine Anwendung über mehrere Geräte verteilt sein kann, ist Zuverlässigkeit ein wichtiger Faktor. Die Anwendung muss in der Lage sein festzustellen, dass Komponenten ausgefallen sind oder nicht mehr korrekt arbeiten. Eine Komponente sendet periodisch ein Signal an den Präsenzdienst von Gaia und wenn dieses Signal ausbleibt, dann wird die Komponente aus dem Space Repository entfernt und die betroffenen Anwendungen werden darüber informiert. In der aktuellen Implementation wird der Koordinator der Anwendung über den erkannten Fehler informiert und dieser löst das Problem auf (z.B.

durch Wechsel auf eine andere Komponenten, Neustart der Anwendung, ...). Kosten für den Wechsel von Komponenten werden in Gaia nicht betrachtet.

2.3.2.2 Aura Das Auraprojekt der Carnegie Mellon University [GSS+02], [SG02] hat das Ziel, den Benutzer und seine Aufgaben in den Mittelpunkt zu stellen. Die Benutzer bewegen sich ähnlich wie in Gaia zwischen verschiedenen Umgebungen, welche reich an verschiedenen Ressourcen sind. Abbildung 8 gibt einen Überblick über die Auraarchitektur.

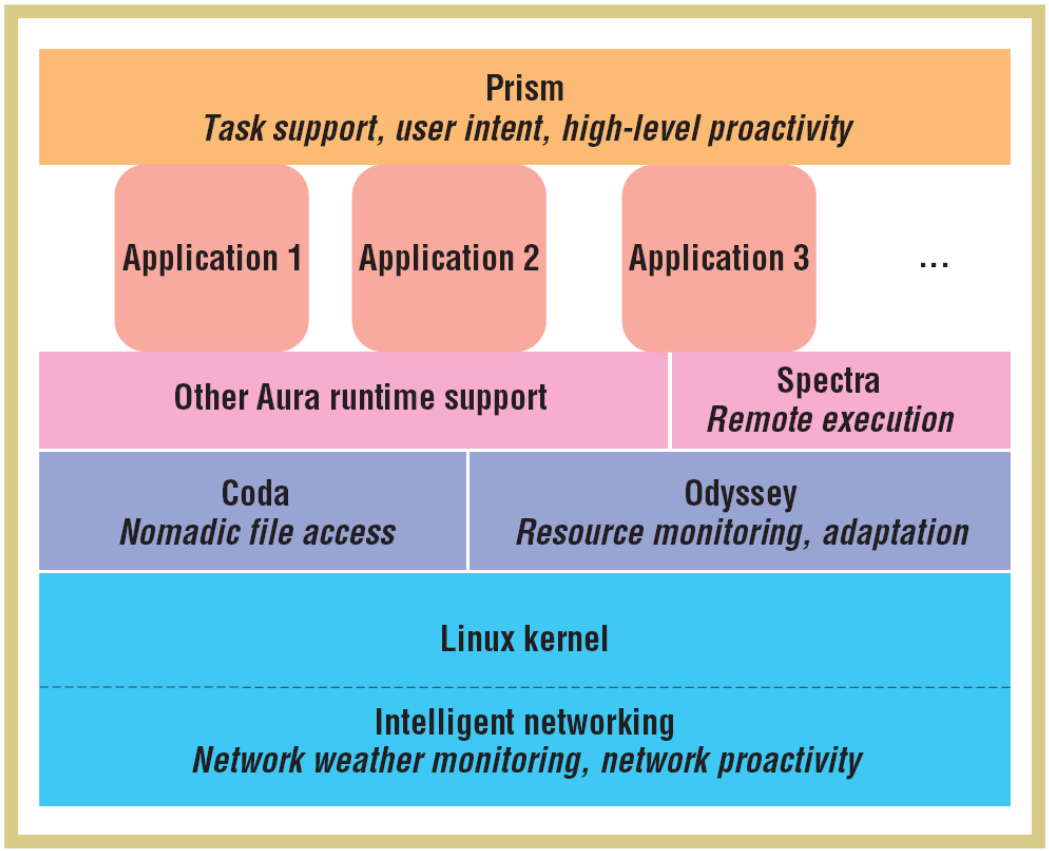


Abbildung 8: Architektur von Aura [GSS+02]

Das Auraprojekt verfolgt den Ansatz, dass jeder Benutzer von einer persönlichen Aura umgeben ist. Diese Aura interagiert als Proxy mit den verschiedenen Umgebungen und der wechselnden Verfügbarkeit von Ressourcen. Dadurch wird die Komplexität der (Re)Konfiguration der Umgebung und Ressourcen von dem Benutzer gekapselt. Damit die Aurainfrastruktur die Umgebung an die Aufgaben eines Benutzers (re)konfigurieren kann, benötigt

die Infrastruktur detaillierte Informationen über die Aufgaben des Benutzers. Abbildung 9 gibt einen groben Überblick über Aura:

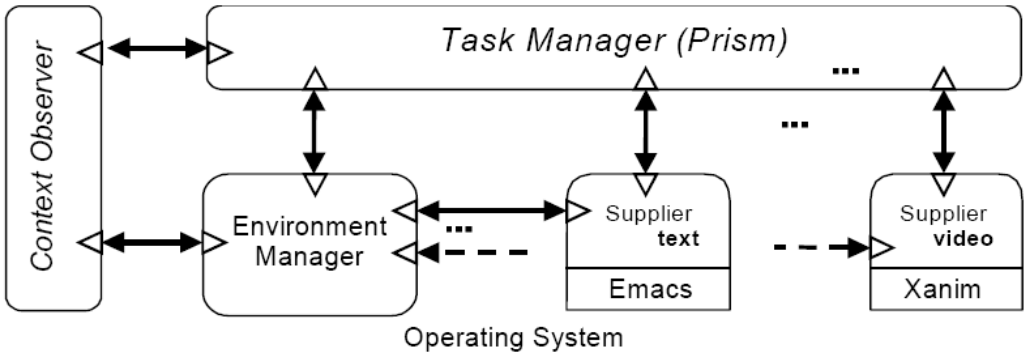


Abbildung 9: Aura Überblick [SG02]

In Aura kann man nach [SG03] zwischen folgende Softwareschichten unterscheiden:

- **Aufgabenmanagement:**
  Die Aufgabe dieser Schicht ist die Erfassung der Aufgaben des Benutzers unter Bezugnahme des Kontexts, in dem sich der Benutzer befindet und welche Absichten der Benutzer hat. Diese Informationen werden ermittelt und an die Umgebungsmanagementschicht weitergeleitet und die verfügbaren Dienste für die Aufgaben ausgehandelt. Eine Aufgabe kann in einer Umgebung möglicherweise durch verschiedene Dienste erfüllt werden und das Aufgabenmanagement versucht die für den Benutzer am besten passenden Dienste zu finden. Mit der Zeit lernt das Aufgabenmanagement die Vorlieben eines Benutzers kennen und nutzt dieses Wissen um die passenden Dienste für die Aufgaben bei der Umgebungsmanagementschicht anzufordern. Informationen über die Aufgabe, wie z.B. benötige Dienstgütegarantien und Benutzervorlieben, werden in einer Nutzenfunktion zusammengefasst, welche an die Umgebungsmanagementschicht übermittelt wird.

- **Umgebungsmanagement:**
  In der Umgebungsmanagementschicht werden abstrakte Modelle der Umgebung gehalten, welche unter anderem die Fähigkeiten der in der Umgebung verfügbaren Geräte beinhalten. Es werden verfügbare Res-

sourcen überwacht und die Dienste festgelegt, welche die Benutzeraufgaben übernehmen. In dieser Schicht findet zur Laufzeit eine ständige Anpassung der Aufgaben an die verfügbaren Dienste statt um dem Benutzer bestmöglich zu unterstützen. Die Rekonfiguration wird durch die Nutzenfunktion gesteuert.

- **Umgebung:**
  In der Umgebungsschicht werden die Dienste und Geräte verwaltet, welche durch die Umgebungsmanagementschicht den Aufgaben eines Benutzers zugewiesen werden können. Wenn die Konfiguration für die Aufgaben konfiguriert sind, interagiert der Benutzer mit den Geräten und Diensten in der Umgebung direkt. Es gibt (ähnlich wie in Gaia) nur einstufige Bindungen zwischen den Diensten und den Aufgaben. Ein Dienst kann also nicht andere Dienste für seine Aufgabe nutzen.

Die Rekonfiguration in Aura findet proaktiv statt. Es wird permanent durch die Infrastruktur überprüft, ob die Nutzenfunktion für die Aufgabe maximiert ist, d.h. die bestmöglichsten verfügbaren Dienste für die Aufgabe benutzt werden. Für eine Rekonfiguration kann es zwei unterschiedliche Arten von Gründen geben:

- Rekonfiguration auf Grund von Wegfall oder Verschlechterung eines Dienstes.
- Rekonfiguration auf Grund von neuen Diensten, welche in der Umgebung verfügbar sind.

Im Falle einer Rekonfiguration auf Grund von neuen Diensten zieht die Umgebungsmanagementschicht beim Wechsel des Dienstes aktiv einen Schnappschuss vom aktuellen Zustand des zu wechselnden Dienstes und spielt diesen gesicherten Zustand in den neuen Dienst ein. Die gleiche Vorgehensweise kann angewendet werden, wenn sich die Dienstgüte eines Dienstes verschlechtert. Um den plötzlichen Wegfall eines Dienstes zu kompensieren, können regelmäßig Schnappschüsse der Dienstzustände gezogen werden und es wird dann der zuletzt konsistente Zustand wieder eingespielt. Die Aufgabenmanagementschicht bietet auch die Möglichkeit einen Schnappschuss des Systemzustandes zu ziehen, falls der Benutzer seine Aufgaben wechselt und später wieder fortsetzen möchte oder die Umgebung wechselt/verlässt.

In Aura kann zusätzlich noch eine Kostenfunktion für den Wechsel von Diensten definiert werden. Es ist nicht immer im Interesse des Benutzers einen Dienst sofort zu wechseln wenn ein besserer Dienst verfügbar ist. Beispielsweise würde es einen Benutzer ablenken, wenn er mitten in der Anwendung das Ausgabemedium wechseln müsste, weil ein besseres Ausgabemedium frei-

geworden ist. Selbst wenn ein Wechsel sinnvoll ist, können Prioritäten gesetzt werden, dass der Wechsel des Dienstes erst dann erfolgt, wenn eine Teilaufgabe erledigt ist, so dass der Benutzer in seiner Arbeit nicht unnötig abgelenkt wird. Die Kostenfunktion wird mit in die Nutzenfunktion eingebunden und sorgt dafür, dass gewisse Komponenten nur gewechselt werden, wenn sie einen relativ großen Vorteil für die Aufgabe des Benutzers versprechen. Die Kostenfunktion kann auch an die Vorlieben des Benutzers angepasst werden.

2.3.3 Dienstgüteorientierte Verteilungsstrukturen

Neben ubiquitären Verteilungsinfrastrukturen gibt es noch eine Reihe von Verteilungsinfrastrukturen, welche die im Netzwerk verfügbare Dienstgütequalität betrachten, wie beispielsweise Spidernet [GNY04], 2KQ [WNGX01] oder CINEMA [BDH+93]. Anwendungen haben beispielsweise zusätzlich zu den benötigen Ressourcen auch gewisse Mindestanforderungen an die Netzwerkcharakteristika wie Verzögerung, Jitter oder Durchsatz. Anwendungen kommen beispielsweise aus dem Multimediabereich wie z.B. die Verteilung von Multimediadaten. Im Folgenden werden stellvertretend für dienstgüteorientierte Verteilungsstrukturen zwei Systeme näher vorgestellt.

2.3.3.1 SpiderNet SpiderNet [GNY04], [GN05] ist ein Peer-to-Peer Rahmenwerk für die Zusammensetzung von Diensten. Verschiedene Geräte bilden spontan ein Peer-to-Peer-basiertes Netzwerk und die Knoten in diesem Netzwerk bieten diverse, atomare Dienste an. Das System unterstützt die Zusammensetzung einzelner Dienste zu neuen Diensten. Einsatzgebiete sind ubiquitäre Verteilung von Inhalten, wie beispielsweise das Streaming von Multimediadaten, oder kooperierende wissenschaftliche Berechnungen. SpiderNet ist ein Forschungsprojekt von der University of Illinois at Urbana-Champaign, welches von der NASA unterstützt wird.

Eine Dienstkomponente in SpiderNet ist eine Anwendungseinheit, welche eine bestimmte Funktionalität (F) anbietet. Die Komponente hat ein oder mehrere Warteschlangen über die die Komponente Eingabedaten über das Netzwerk empfängt. Eingabedaten werden durch die Komponente nur akzeptiert, wenn sie eine gewisse minimale Dienstgüte (Q^{in}) erfüllen. Nach der Verarbeitung der Eingabedaten erzeugt die Komponente Ausgaben mit der Dienstgüte (Q^{out}). Weiterhin hat die Komponente einen gewissen Verbrauch an Ressourcen R und es wird eine gewisse Dienstgütequalität Q^p mit ihr assoziiert, welche beispielsweise Verzögerung und Datenverlustrate mit einbezieht.

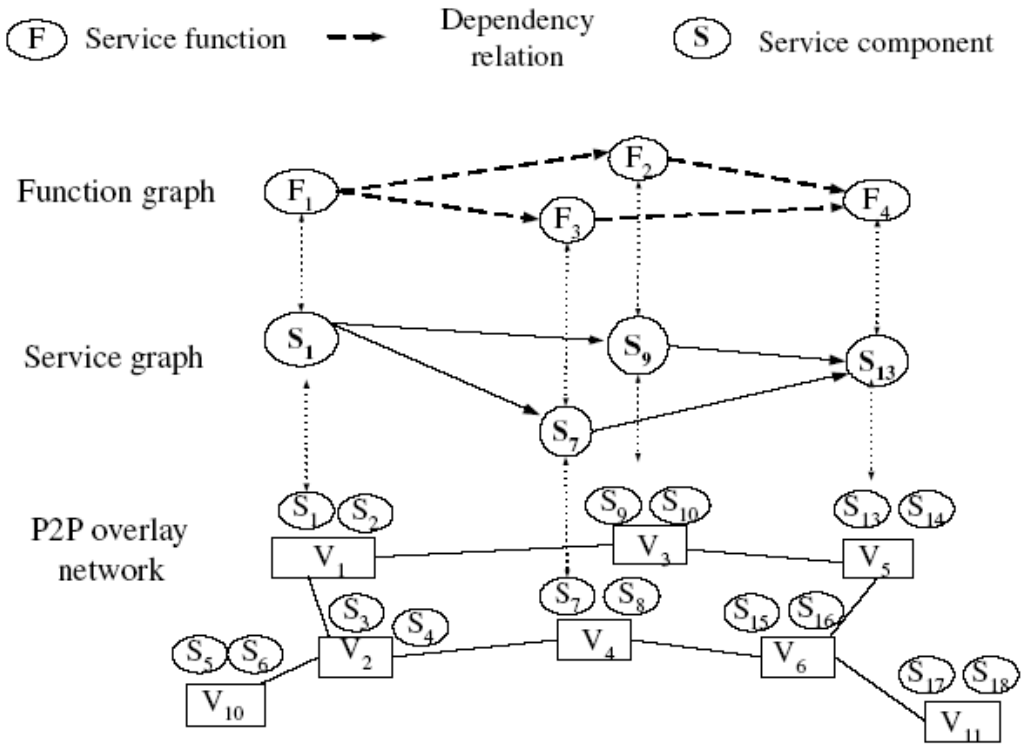


Abbildung 10: SpiderNet Dienstzusammensetzung Systemarchitektur [GNY04]

Eine Anforderung nach einem zusammengesetzten Dienst besteht aus zwei Bestandteilen: Einem Graph, der die funktionale Zusammensetzung des Dienstes darstellt, siehe Abbildung 10 und die Dienstgütegarantieanforderungen an den zusammengesetzten Dienst Q^{req}. Der Funktionsgraph verdeutlicht die Abhängigkeiten der einzelnen Dienst zueinander, beispielsweise benötigen die Dienste F_2 und F_3 aus Abbildung 10 Eingabedaten von F_1. Der Servicegraph repräsentiert die aktuelle Zusammensetzung der einzelnen Dienste.

SpiderNet verfolgt eine proaktive Wiederherstellung im Fehlerfall. Der Sender, welcher den zusammengesetzten Dienst anstößt, hält für jede aktive Sitzung eine gewisse Anzahl an Ersatzservicegraphen, damit er im Fehlerfall schnell auf einen der Ersatzgraphen umschalten kann. Der Sender muss während der Ausführung regelmäßig überprüfen, ob die Ersatzservicegraphen noch funktionsfähig sind und die geforderten Dienstgütegarantien erfüllen. Die Anzahl der Ersatzservicegraphen und der Grad der Überdeckung mit dem aktiven Servicegraphen sind je nach Einsatzszenario unterschiedlich und erfordern einen Kompromiss zwischen Ausfallsicherheit und schneller Wiederherstellung im Fehlerfall.

2.3.3.2 2K^{Q+} Das $2K^{Q+}$-Projekt ist ein komponentenbasiertes Rahmenwerk für einheitliches Dienstgütequalitätsmanagement, entwickelt an der University of Illinois at Urbana-Champaign [WNGX01]. $2K^{Q+}$ ermöglicht die Entwicklung verschiedener verteilter, komponentenbasierter Anwendungen (z.B. verteilte Multimediaanwendungen). In einer Anwendung wird ein Pfad von einem Sender zu einem Empfänger aufgebaut. Dieser Pfad ist abhängig von den im Netzwerk verfügbaren Ressourcen und Dienstgütequalitätsanforderungen. Verändert sich die Verfügbarkeit von Ressourcen oder die Dienstgüte von Komponenten im Netzwerk, dann wird zur Laufzeit die Anwendung entsprechend rekonfiguriert. Die Rekonfiguration sucht zur Laufzeit eine alternative Konfiguration, welche die benötigten Ressourcen und Dienstgütequalität bietet. Weitere Kosten für die Rekonfiguration werden hier nicht betrachtet.

Abbildung 11 gibt einen Überblick über das Rahmenwerk. Das Rahmenwerk besteht aus drei Bestandteilen. Die Grundlage bildet das komponentenbasierte, netzwerkzentrische Betriebssystem 2K [KCM+00], welches in Teilen auch in GaiaOS eingesetzt wird. Auf dem 2K-Betriebssystem ist die komponentenbasierte Middleware $2K^Q$ aufgesetzt, welche dynamisch zur Laufzeit die Rekonfiguration durchführt. Die Rekonfiguration wird durch die Dienstgütequalitätsanforderungen von der Anwendung, dynamischen Einschränkun-

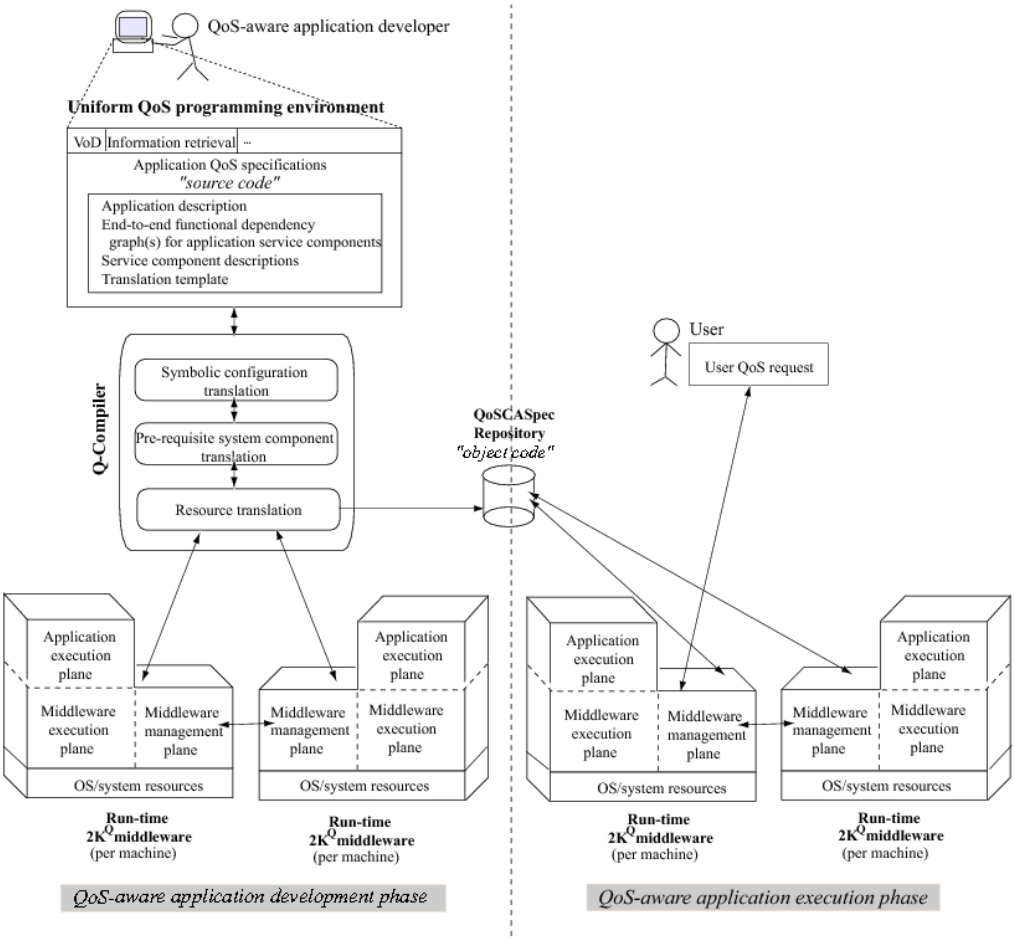


Abbildung 11: $2K^{Q+}$ Middlewarearchitektur [WNGX01]

gen, Verfügbarkeit von Ressourcen und Benutzervorgaben gesteuert. Der Q-Compiler und die einheitliche Dienstgüte-Programmierumgebung bieten eine einheitliche Schnittstelle zur Spezifikation von Dienstgütequalitätskriterien.

3 Analyse

3.1 Problemanalyse

PCOM ermöglicht die Entwicklung von verteilten und komponentenbasierten Anwendungen in ubiquitären Umgebungen. Ein grundlegendes Merkmal von ubiquitären Umgebungen ist die hohe Anzahl an mobilen und heterogenen Geräten, welche miteinander kommunizieren. Auf Grund der Mobilität und möglicher Systemausfälle muss eine Anwendung an die neue Situation innerhalb des Systems durch PCOM angepasst werden können.

Die Suche nach einer gültigen Konfiguration in PCOM, welche Ressourceneinschränkungen und Abhängigkeiten von Komponenten untereinander beachtet, ist ein NP-vollständiges Problem. Die NP-Vollständigkeit für das Problem hat Michael Reinsch in seiner Diplomarbeit [Rei04] nachgewiesen und für die Suche nach einer initialen PCOM-Konfiguration einen Algorithmus entwickelt. Wie in Kapitel 1.3 schon kurz vorgestellt, wird im Rahmen dieser Diplomarbeit ein Algorithmus entwickelt, welcher PCOM-Anwendungen an sich verändernde Komponenten- und Ressourcenverfügbarkeiten anpasst. Neben der Anpassung der Anwendung an die veränderte Umgebung muss der Algorithmus darauf achten, dass eine Konfiguration gewählt wird, welche möglichst geringe Kosten für die Anpassung verursacht.

Es lassen sich für diese Diplomarbeit daher zwei Problemfelder identifizieren, welche im Rahmen dieser Arbeit näher untersucht werden:

- (1) Die Suche nach einer neuen, gültigen Konfiguration, damit die Anwendung in der veränderten Umgebung wieder fortgesetzt werden kann.
- (2) Die gesuchte Konfiguration aus (1) soll möglichst geringe Anpassungkosten verursachen. Um die Anpassungskosten beurteilen zu können, müssen zuerst die Kosten für die Anpassung von Anwendungen identifiziert werden. Nach der Identifizierung der Kosten wird ein Kostenmodell entworfen, welches zur Anpassung der Anwendung hinzugezogen werden kann und die gefundene Lösung bewertet.

Wie in Kapitel 2.1 bereits eingeführt gibt es verschiedene Aspekte für die Rekonfiguration. Der zu entwickelnde Rekonfigurationsalgorithmus konzentriert sich auf die reaktive Rekonfiguration von Anwendungen, d.h. der Wegfall von Anwendungskomponenten zur Laufzeit der Anwendung löst eine Rekonfiguration aus. Das System versucht eine neue Konfiguration zu finden, indem eine strukturelle Rekonfiguration durchgeführt wird.

Im Folgenden werden zuerst die möglichen Rekonfigurationsalternativen vorgestellt, um eine Anwendung zu rekonfigurieren. Danach wird auf die Anpassung von Anwendungen unter Beachtung von Kosten näher eingegangen.

3.1.1 Rekonfigurationsalternativen

Das NP-vollständige Problem der Suche nach einer gültigen Konfiguration wird auf die Lösung eines DCSP umgesetzt. Für die Rekonfiguration wird das Problem der Suche nach einer neuen, gültigen Konfiguration weiterhin als DCSP betrachtet. Wenn eine Anwendungskonfiguration an die veränderte Umgebung angepasst werden muss, gibt es grundsätzlich zwei Rekonfigurationsalternativen:

- **Erneute Konfiguration:**
  Bei einer erneuten Konfiguration einer Anwendung wird der letzte, konsistente Zustand der Anwendung gesichert. Nach der Sicherung werden alle verbliebenen Komponenteninstanzen der Anwendung freigegeben und danach startet der Algorithmus zur Suche nach einer initialen Konfiguration neu und konfiguriert eine neue Komponentenkomposition für die Anwendung. Wenn der Algorithmus eine neue Konfiguration findet, wird danach der gesicherte Anwendungszustand wieder in die Anwendung übertragen und die Anwendung mit diesem Zustand wieder gestartet.

  Diese Strategie ist sehr einfach, da sie keine Modifikation an dem bestehenden Konfigurationsalgorithmus benötigt. Allerdings werden keinerlei Rekonfigurationskosten beachtet. Unter Umständen ist die Suche nach einer neuen Anwendungskonfiguration aufwendiger als das Ersetzen der abgebrochenen Anwendungsteile. Außerdem kann die Wiederherstellung eines kompletten Anwendungszustand einen sehr hohen Aufwand erfordern. Es sind allerdings Szenarien denkbar, wo eine erneute Konfiguration der Anwendung durchaus von Vorteil sein kann. Dies ist beispielsweise dann der Fall, wenn keine Anwendungszustände wiederhergestellt werden müssen.

- **Rekonfiguration der Anwendungskonfiguration:**
  Anstatt eine Anwendung im Fehlerfall komplett zu beenden und eine neue Konfiguration zu suchen, verfolgt diese Strategie das Ziel, die bestehende Anwendung möglichst zu erhalten und die betroffenen Anwendungsteile zu rekonfigurieren.

  Abbildung 12 zeigt in (a) ein Beispiel für eine Anwendungskomposition

und in (b) die Anwendungskomposition im Fehlerfall, wenn die Komponente K0 ausfällt. Die Komponenten K0-0 und K0-1 bemerken nach einem Timeout, dass sie keine Bindung mehr zu ihrer Vaterkomponente K0 haben und nach dem Timeout werden die Komponenten K0-0 und K0-1 freigegeben.

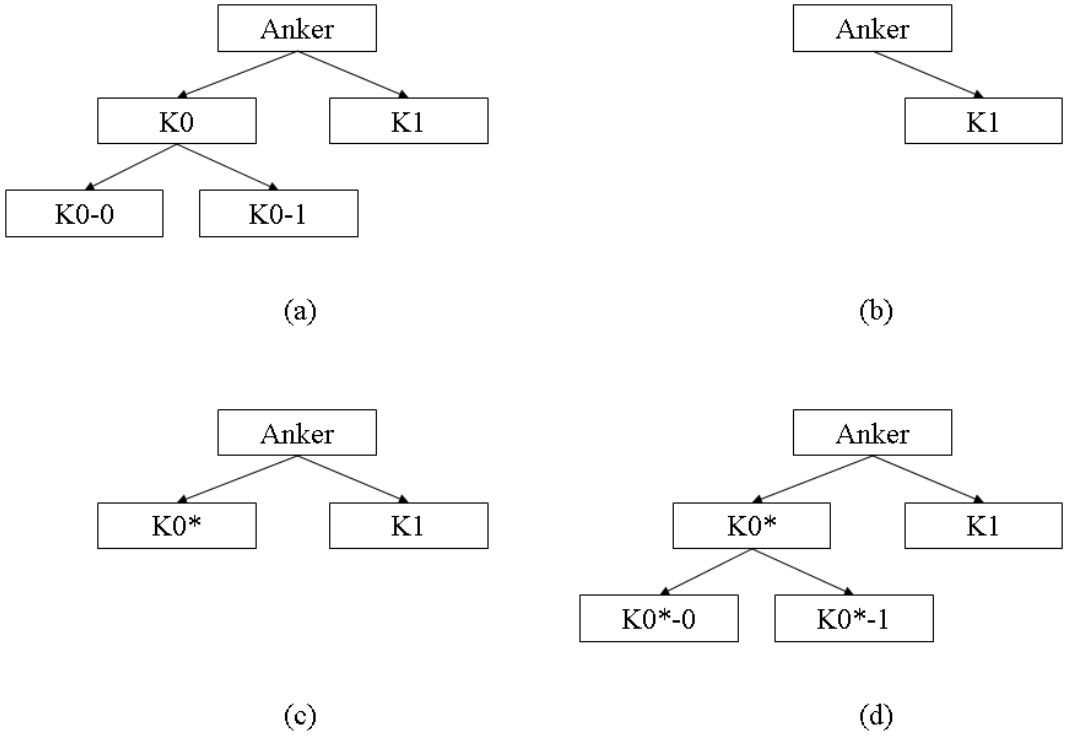


Abbildung 12: Rekonfiguration einer Anwendungskonfiguration

In diesem Beispiel versucht nun der Anwendungsanker eine neue Komponente für seine erste Abhängigkeit zu wählen und die Wahl für die zweite Abhängigkeit (K1) bleibt bestehen, d.h. es wird zuerst versucht, nur die nötigen Anwendungsteile zu rekonfigurieren. In (c) hat der Anwendungsanker eine neue Komponenten K0* gewählt, welche für sich selbst in (d) dann die Komponenten K0*-0 und K0*-1 wählt.

In diesem Fall konnte die Anwendung rekonfiguriert werden, ohne die anderen Teile der Anwendung zu rekonfigurieren. Wenn allerdings auf Grund von Ressourceneinschränkungen für den zu rekonfigurierenden Anwendungsteil keine Komposition gefunden werden kann, dann müssen auch bestehend Anwendungsteile, die prinzipiell funktionsfähig sind, rekonfiguriert werden.

3.1.2 Rekonfiguration unter Beachtung von Kostenfaktoren

Damit der Rekonfigurationsalgorithmus Kostenfaktoren mit in die Suche nach einer neuen Konfiguration einbeziehen kann, wird in Kapitel 4.1.3 ein Kostenmodell entworfen. Mit Hilfe dieses Kostenmodells kann eine gefundene gültige Konfiguration bezüglich ihrer Kosten für die Rekonfiguration bewertet werden.

Die Suche nach einer optimalen (in Bezug auf die entstandenen Kosten), gültigen Konfiguration ist ein Optimierungsproblem und ist NP-hart. Informal lässt sich das folgendermaßen veranschaulichen: Die Kosten für eine Konfiguration sind erst bekannt, wenn die Konfiguration gefunden ist. Der Grund dafür liegt darin, dass die Komponenten rekursiv weitere Komponenten benötigen können und für diese wieder neue Kosten entstehen, d.h. die Kosten können erst berechnet werden, wenn die Komponentenkomposition für die Anwendung vollständig aufgebaut ist. Die Berechnung einer gültigen Konfiguration ist NP-vollständig. Wenn man die optimale Lösung, d.h. die Konfiguration mit den geringsten Kosten sucht, dann muss man alle gültigen Konfigurationen betrachten und untereinander vergleichen, da selbst die letzte gültige Konfiguration noch möglicherweise das Optimum sein kann. Das bedeutet, dass man das NP-vollständige Problem mehrfach lösen muss, was bedeutet, dass das Problem NP-hart ist.

3.1.3 Lösungsstrategien

Neben der Anpassung der Anwendung an die sich veränderte Umgebung soll der zu entwickelnde Algorithmus eine Konfiguration finden, welche möglichst geringe Kosten für die Anpassung verursacht. Für die Anpassung mit Berücksichtigung der Kosten gibt es grundsätzlich zwei Möglichkeiten das Problem zu lösen:

- **Vollständiger Optimierungsalgorithmus:**
  Die Suche nach einer Konfiguration, welche möglichst geringe Kosten verursacht, ist ein Optimierungsproblem. Das Problem kann durch einen vollständigen Optimierungsalgorithmus gelöst werden, indem beispielsweise alle möglichen, gültigen Konfigurationen miteinander verglichen werden.

- **Heuristik:**
  Neben einem vollständigen Optimierungsalgorithmus kann die Suche nach einer optimalen Konfiguration durch geeignete Heuristi-

ken/Strategien erfolgen.

3.1.4 Anforderungen an den Algorithmus

Für den zu entwickelnden Algorithmus ergeben sich daher mehrere Anforderungen, welche er zu erfüllen hat:

- **Gültigkeit:**
  Der Algorithmus hat das Ziel eine Anwendung zu rekonfigurieren, wenn Teile der Anwendung auf Grund von veränderten Verfügbarkeiten von Komponenten und Ressourcen nicht mehr vorhanden sind. Die Anwendung befindet sich nach dem Ablauf des Algorithmus wieder in einer gültigen Konfiguration, falls eine gültige Konfiguration existiert.
- **Vollständigkeit:**
  Der Algorithmus soll in der Lage sein eine Anwendung zu rekonfigurieren und festzustellen, ob es eine alternative gültige Konfiguration existiert. Falls keine gültige Konfiguration existiert, dann wird die Anwendung abgebrochen und der Anwender darüber informiert.
- **Rekonfigurationskosten:**
  Die gewählte neue Konfiguration sollte möglichst geringe Kosten verursachen. Der Algorithmus soll daher die Rekonfigurationskosten mit in die Suche nach einer neuen Konfiguration einbeziehen. Die Rekonfigurationskosten werden in einem Kostenmodell abstrahiert.
- **Effizienz:**
  Auf Grund der Vielzahl an heterogenen Geräten und der Komplexität eine alternative gültige Konfiguration zu finden, soll der Algorithmus möglichst sparsam mit den auf den Geräten verfügbaren Ressourcen umgehen.

3.1.5 Bewertung der Lösungsstrategien

Im Folgenden werden die Vor- und Nachteile der Lösungsstrategien vorgestellt und die Auswahl einer Lösungsstrategie begründet.

3.1.5.1 Vollständiger Optimierungsalgorithmus Ein vollständiger Optimierungsalgorithmus hat den Vorteil, dass die ermittelte Konfiguration nach der Rekonfiguration immer das Optimum darstellt. Der Optimierungsalgorithmus garantiert, dass keine Konfiguration existiert, welche geringere

Kosten hat als das ermittelte Optimum. Falls mehrere günstigste Konfigurationen existieren, wählt der Algorithmus eine dieser Konfigurationen als neue Konfiguration.

Der vollständige Optimierungsalgorithmus hat neben dem Vorteil, dass immer die Konfiguration mit den geringsten Kosten ermittelt wird, einen nicht unerheblichen Nachteil. Damit im Allgemeinen garantiert werden kann, dass die ermittelte Konfiguration das Optimum bezüglich der verursachten Kosten darstellt, müssen in der Regel mehrere Konfigurationen ermittelt und miteinander verglichen werden. Es kann zwischen zwei Fällen unterschieden werden:

- **Das Optimum ist vor der Rekonfiguration bekannt:**
  Wenn das Optimum vor der Rekonfiguration bereits bekannt ist, dann kann diese Information für die Suche nach der optimalen Konfiguration hinzugezogen werden. Beispielsweise kann der Optimierungsalgorithmus Teillösungen verwerfen, welche größere Kosten verursachen als die optimale Konfiguration, da keine negativen Kosten für die Rekonfiguration einer Anwendungskonfiguration entstehen können.

- **Das Optimum ist vor der Rekonfiguration nicht bekannt:**
  Da das Optimum einer Rekonfiguration in der Regel nicht im Voraus bekannt ist, muss der Optimierungsalgorithmus im Allgemeinen mehrere Konfigurationen ermitteln und miteinander vergleichen. Dies führt zu einer erheblich längeren Laufzeit, da das NP-vollständige Problem für die Suche nach einer gültigen Konfiguration möglicherweise mehrmals gelöst werden muss. Im schlimmsten Fall müssen alle Konfigurationen durch den Optimierungsalgorithmus ermittelt und miteinander verglichen werden.

3.1.5.2 Heuristik Anstatt einem vollständigen Optimierungsalgorithmus kann für die Suche nach einer optimalen Konfiguration auch eine geeignete Heuristik verwendet werden. Der Vorteil einer Heuristik ist, dass sehr häufig eine Lösung nahe am Optimum ermittelt wird und die Laufzeit zur Berechnung der Konfiguration niedriger liegt als beim Optimierungsalgorithmus. Der Nachteil ist, dass eine Heuristik nicht in jedem Fall eine möglichst optimale Lösung ermittelt.

Auf Grund der Anforderungen an den zu entwickelnden Algorithmus wurden im Rahmen dieser Diplomarbeit Heuristiken entwickelt, welche eine Rekonfiguration von Anwendungen ermöglicht. Die zu entwickelnden Heuristiken sind Gegenstand von Kapitel 4.

3.1.6 Einschränkungen

Für den Algorithmus, welcher im Rahmen dieser Diplomarbeit entwickelt wird, gelten ähnliche Einschränkungen wie für den Algorithmus zur initialen Konfiguration.

- **Konkurrenz zwischen Anwendungen:**
  Der Rekonfigurationsalgorithmus berücksichtigt nicht die Situation, dass mehrere Anwendungen gleichzeitig rekonfiguriert werden und daher in Konkurrenz um die in der Umgebung verfügbaren Ressourcen sind.
- **Stabilität der Umgebung:**
  Während der Rekonfiguration wird angenommen, dass sich die Ressourcenverfügbarkeit in der Umgebung nicht verändert, die Umgebung also stabil ist.
- **Reaktive Rekonfiguration:**
  Der Rekonfigurationsalgorithmus beschränkt sich nur auf Rekonfigurationen, wenn Teile der Anwendung nicht mehr verfügbar/funktionsfähig sind, siehe reaktive Rekonfiguration in Kapitel 2.1.4.

4 Lösungsansatz

4.1 Kostenmodell

Damit Konfigurationen bezüglich ihrer Kosten bewertet werden können, wird im Folgenden ein Kostenmodell entworfen. Zuerst werden in Kapitel 4.1.1 die Kostenfaktoren für die Berechnung der Kosten näher untersucht und in Kapitel 4.1.3 wird das Kostenmodell näher vorgestellt.

4.1.1 Kostenfaktoren für die Rekonfiguration

Der im Rahmen dieser Diplomarbeit zu entwickelnde Algorithmus beschränkt sich auf reaktive, strukturelle Rekonfiguration, d.h. eine Anwendung wird durch den plötzlichen Wegfall von verwendeten Komponenten in ihrer Ausführung unterbrochen. Die Anpassung von Anwendungskonfigurationen an die veränderten Umgebungen verursacht Kosten. Die Kostenfaktoren für die Anpassung von Konfigurationen sind Gegenstand dieses Unterkapitels und werden im Folgenden näher betrachtet.

Eine gültige Konfiguration einer komponentenbasierten PCOM-Anwendung kann als Baum mit gerichteten Kanten dargestellt werden. Die Knoten im Baum repräsentieren die Komponenten der Anwendung. Die gerichteten Kanten repräsentieren die Anforderungen von Komponenten an andere Komponenten, siehe Abbildung 13.

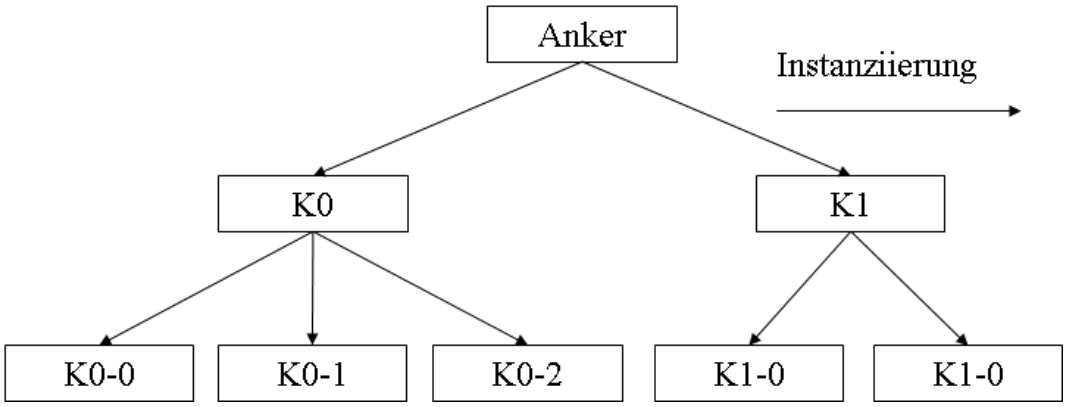


Abbildung 13: Komposition einer Anwendung

In einer PCOM-Anwendung lassen die Komponenten in zwei Klassen einteilen:

- **Komponenten mit Abhängigkeiten zu anderen Komponenten:** Diese Klasse beinhaltet Komponenten, welche zur Bereitstellung ihres Dienstes die Instanziierung weiterer, geeigneter Komponenten erfordert. Die Erfordernisse an weitere Komponenten sind im PCOM-Vertrag explizit spezifiziert. In der Abbildung 13 ist diese Klasse von Komponenten durch die Knoten im Baum repräsentiert, welche einen Ausgangsgrad größer null besitzen.

- **Komponenten ohne Abhängigkeiten zu anderen Komponenten:**
  Komponenten aus dieser Klasse benötigen keine Instanziierung weiterer Komponenten um ihren Dienst anzubieten. In der Regel bieten Komponenten aus dieser Klasse sehr elementare Dienste an, welche von Komponenten aus der anderen Klasse benötigt werden, um komplexere Dienst anzubieten. In der Abbildung 13 ist diese Klasse von Komponenten durch die Knoten im Baum repräsentiert, welche einen Ausgangsgrad gleich null besitzen.

Wenn eine Anwendung durch den Wegfall von Komponenten in ihrer Ausführung unterbrochen wurde, muss für die Anwendung eine neue, gültige Konfiguration gesucht werden. Wie in Kapitel 3.1 bereits eingeführt, bestehen grundsätzlich zwei Möglichkeiten die Anwendung anzupassen: Erneute Konfiguration oder Rekonfiguration der Anwendungskonfiguration. In beiden Fällen setzen sich die Kosten für die Anpassung aus mehreren Bestandteilen zusammen:

- **Aufwand für die Sicherung von Komponentenzuständen:**
  Grundsätzlich muss zu jeder Zeit eine Wiederherstellung von Komponentenzuständen möglich sein, da im Allgemeinen nicht vorhergesagt werden kann, wann eine Komponente innerhalb der Anwendung nicht mehr verfügbar sein wird. Es besteht allerdings die Möglichkeit, dass noch funktionsfähige Komponenten innerhalb der Anwendung gewechselt werden müssen, damit eine neue gültige Konfiguration gefunden werden kann. Der Aufwand für den gezielten Wechsel einer Komponente unterscheidet sich daher von der allgemeinen Sicherung von Komponentenzuständen.

  Wenn eine Komponente aus der Anwendungskonfiguration gewechselt wird, muss der Zustand der zu wechselnden Komponente gesichert werden, damit er nach der erfolgten Rekonfiguration wiederhergestellt werden kann. Falls die zu wechselnde Komponente keine weiteren Komponenten instanziiert hat, beschränkt sich die Sicherung des Zustand auf die zu wechselnde Komponente. Wenn allerdings eine Komponente in

der Anwendung gewechselt werden soll, welche weitere Komponenten instanziiert hat (beispielsweise die Komponente K0 aus Abbildung 13), müssen die Zustände dieser Komponenten ebenfalls gesichert werden, da der Wechsel der Vaterkomponente automatisch die Bindung zu den instanziierten Komponenten löst und diese dann in der Umgebung wieder verfügbar werden. Das bedeutet, dass der Wechsel einer Vaterkomponente auch einen Wechsel aller Kinderkomponenten anstößt.

Der konkrete Aufwand für die Sicherung eines Komponentenzustand ist von der Implementierung der Komponente, der verwendeten Sicherungsstrategie und dem aktuellen Zustand innerhalb der Komponente abhängig. Ein weiterer Faktor ist die Hardwareplattform, auf der die Komponenteninstanz ausgeführt wird, da die Leistungsfähigkeit der Plattform die Geschwindigkeit der Sicherung beeinflusst. Von Vorteil für den Entwurf des Kostenmodells ist, dass sich dieser Aufwand für jede Komponente im Voraus berechnen lässt (in Abhängigkeit des aktuellen Anwendungszustand).

- **Aufwand für die Übertragung des gesicherten Systemzustands in eine andere Komponente:**
  Ein gesicherter Zustand muss in eine andere Komponente übertragen werden, damit die zu wechselnde Komponente gewechselt werden kann. Wenn eine geeignete Ersatzkomponente gefunden ist, muss der gesicherte Zustand in diese Komponente wiedereingespielt werden. Der benötigte Aufwand für die Übertragung des gesicherten Systemzustands in eine andere Komponente ist von der Qualität der Netzwerkverbindung und der Größe der zu übertragenden Daten abhängig.

- **Aufwand für die Suche nach einer neuen gültigen Konfiguration:**
  Die Suche nach einer gültigen Konfiguration ist ein NP-vollständiges Problem. Im Allgemeinen kann der Aufwand zur Suche einer neuen, gültigen Konfiguration im Voraus nur sehr schwer abgeschätzt werden, da möglicherweise alle möglichen Konfigurationen untersucht werden müssen.

  Im Fall des Wechsel einer Komponente innerhalb der Anwendung muss eine geeignete Ersatzkomponente gefunden werden, welche die gestellten Anforderungen erfüllt. Auf Grund von Ressourcenbeschränkungen auf den Containern ist es möglich, dass diese Ersatzkomponente nur dann instanziiert werden kann, wenn andere Komponenten innerhalb der Anwendung gewechselt werden. Der Wechsel von weiteren Komponenten erhöht daher weiter den Aufwand für die Suche nach einer neuen

gültigen Konfiguration. Der konkrete Aufwand für die Suche nach einer Ersatzkomponente lässt sich erst berechnen, wenn für die weiteren Komponenten Ersatzkomponenten gefunden wurden, d.h. wenn eine neue gültige Konfiguration gefunden wurde.

- **Aufwand für die Wiederherstellung des gesicherten Zustands in den Ersatzkomponenten:**
  Der Aufwand für die Wiederherstellung eines gesicherten Zustands ist von mehreren Faktoren abhängig: (1) der verwendeten Wiederherstellungsstrategie innerhalb der Komponente und (2) der Leistungsfähigkeit der Hardwareplattform, auf der die Komponenteninstanz ausgeführt wird.

Mögliche Verfahren zur Sicherung und Wiederherstellung eines Komponentenzustands werden in Kapitel 4.1.2 diskutiert.

4.1.2 Verfahren zur Wiederherstellung von Dienstzuständen

In [Keil03] wurde die Unterstützung zustandsbehafteter Dienste im Ubiquitous Computing in Bezug auf PCOM untersucht. Es gibt verschiedene Verfahren zur Wiederherstellung eines Systemzustandes. Nachfolgend werden mögliche Verfahren kurz vorgestellt:

Replay-Verfahren:
Replay-Verfahren werden unter anderem in Datenbanksystemen wie DB2 [HR01] von IBM eingesetzt. Im Replay-Verfahren wird die Historie der Funktionsaufrufe mitprotokolliert. Wenn ein Dienst ausfällt, dann wird die Historie auf einen anderen geeigneten Dienst übertragen und dort die entsprechenden Befehle der Reihe nach ausgeführt. Da die Dienste die State-Machine-Eigenschaft erfüllen, befindet sich der Dienst nach Abarbeitung der Historie in dem identischen Zustand wie der ausgefallene Dienst. Die Historie wird auf dem Klient gespeichert, der den Dienst nutzt. Das Verfahren ist sehr einfach zu implementieren, allerdings steigt die Größe der Historie mit jedem protokollierten Funktionsaufruf und die Wiedereinspielung nimmt möglicherweise sehr viel Zeit in Anspruch.

Checkpoint-Verfahren:
Checkpoint-Verfahren sichern explizit den Zustand eines Dienstes. Es wird ein Schnappschuss eines Dienstes erstellt und an den Klient des Dienstes übertragen. Der Vorteil von diesem Verfahren ist, dass der gesicherte Zustand in einen anderen Dienst direkt übertragen werden kann und dieser nicht erst die Funktionshistorie abarbeiten muss. Um eine lückenlose Wiederher-

stellbarkeit zu gewährleisten muss allerdings bei jedem Zustandswechsel ein Schnappschuss des Dienstzustands erzeugt und auf den Klienten übertragen werden, was einen hohen Aufwand des Verfahrens bedeutet.

Kombination der Verfahren:
Eine Kombination dieser zwei Verfahren ist ebenfalls möglich. Durch die Kombination werden die Nachteile der beiden Verfahren verringert. Es wird am Anfang der Ausführung ein Schnappschuss des Dienstzustands erstellt und danach eine Historie mitprotokolliert. Wenn die Historie eine bestimmte Größe überschreitet oder beispielsweise sehr rechenintensive Berechnungen in dem Dienst stattgefunden haben, wird wieder aktiv ein Schnappschuss des Dienstzustands erzeugt und an den Klienten übermittelt. Der vorherige Schnappschuss und die bis zum Schnappschuss protokollierten Funktionsaufrufe können entfernt werden und eine neue Historie angelegt werden.

4.1.3 Entwurf des Kostenmodells

4.1.3.1 Anforderungen an das Kostenmodell Für das zu entwickelnde Kostenmodell gelten folgende Anforderungen:

- **Modellierung der Kosten einzelner Anwendungskomponenten:**
  Das Kostenmodell soll die Kosten für die Rekonfiguration der einzelnen Anwendungskomponenten modellieren, d.h. im Kostenmodell ist bekannt, welche Kosten der Wechsel der einzelnen Komponenten verursacht.
- **PCOM-Integration:**
  Das Kostenmodell soll in PCOM intgegriert werden, so dass Konfigurationen auf ihre Kosten hin bewertet werden können. Die Modellierung der Kosten in PCOM ermöglicht die Integration von Heuristiken, welche die Kosteninformationen für die Suche nach einer neuen Konfiguration berücksichtigen.

4.1.3.2 Einschränkungen des Kostenmodells Das im Rahmen dieser Diplomarbeit entworfene Kostenmodell modelliert die Sicherungskostenfaktoren. Für die Wiederherstellungskosten wird angenommen, dass diese mit hoher Wahrscheinlichkeit relativ gleich groß sind wie die Sicherungskosten. Für die Berechnung der tatsächlichen Wiederherstellungskosten müsste die Wiederherstellung einer gefundenen Konfiguration durchgeführt werden, d.h. diese Kosten sind erst bekannt, wenn eine Konfiguration bereits gefunden ist.

Weiterhin lassen sich die Kosten für die Suche nach einer Ersatzkomponente im Allgemeinen nicht im Voraus berechnen, da die Ersatzkomponenten möglicherweise weitere Komponenten instanziieren oder den Wechsel anderer Komponenten benötigen.

Die Folge davon ist, dass die Berechnung eines Kostenmodells erst dann möglich wäre, wenn eine gültige Konfiguration gefunden wurde. Dieses Kostenmodell wäre dann nur in der Lage die Kosten für die Rekonfiguration im Nachhinein für diese eine Konfiguration zu beurteilen. Das im Rahmen dieser Diplomarbeit zu entwickelnde Kostenmodell soll allerdings die Kosten für die Anpassung der Anwendung bereits zur Rekonfiguration ermitteln, damit eine möglichst optimale Rekonfigurationsstrategie gewählt werden kann, so dass die Kosten für die Anpassung möglichst gering sind.

4.1.3.3 Kostenmodellierung Eine Anwendung in PCOM ist eine Komposition von Komponenten. Für die Modellierung der Kosten gelten folgende Annahmen:

- **Modellierung von Sicherungskosten:**
  Das Kostenmodell modelliert die Sicherungskosten für die Rekonfiguration von Komponenten. Jede Komponente kann die Kosten für die Sicherung des eigenen Zustands ermitteln und lokal speichern. Die internen Kosten der einzelnen Komponenten sind größer oder gleich 1, d.h. jede Rekonfiguration einer Komponente verursacht Kosten und es kann keine negativen Kosten geben. Wie in Kapitel 4.1.1 ausgeführt, stößt der Wechsel einer Komponente möglicherweise den Wechsel weiterer Komponenten an. Damit dieser Kostenfaktor mit berücksichtigt werden kann, müssen die Sicherungskosten der instanziierten Komponenten bekannt sein. Die Kosten für den Wechsel einer Komponente setzen sich aus der Summe der Kosten für den Wechsel der instanziierten Komponenten und den internen Kosten für die Komponente zusammen.

- **Kostenverteilung:**
  Weiterhin gilt, dass eine Elternkomponente immer größere Rekonfigurationskosten besitzt als ihre instanziierten Komponenten. Die Kosten entlang eines Pfades von einer Komponente mit Ausgangsgrad = 0 zum Anker sind daher steigend monoton.

Die Kostenberechnung für die Rekonfiguration wird durch den Wegfall von Anwendungskomponenten ausgelöst. Die Anwendung wird in ihrer Ausführung unterbrochen und vom Anker ausgehend wird an die verfügbaren, in-

stanziierten Anwendungskomponenten eine Nachricht verschickt (*berechneKosten*), siehe Abbildung 14. Die Kinder-Komponenten verschicken wiederum eine Nachricht an die von ihnen instanziierten Komponenten. Hat eine Komponente keine weiteren Komponenten instanziiert oder sind diese nicht mehr vorhanden, dann werden keine Nachrichten verschickt.

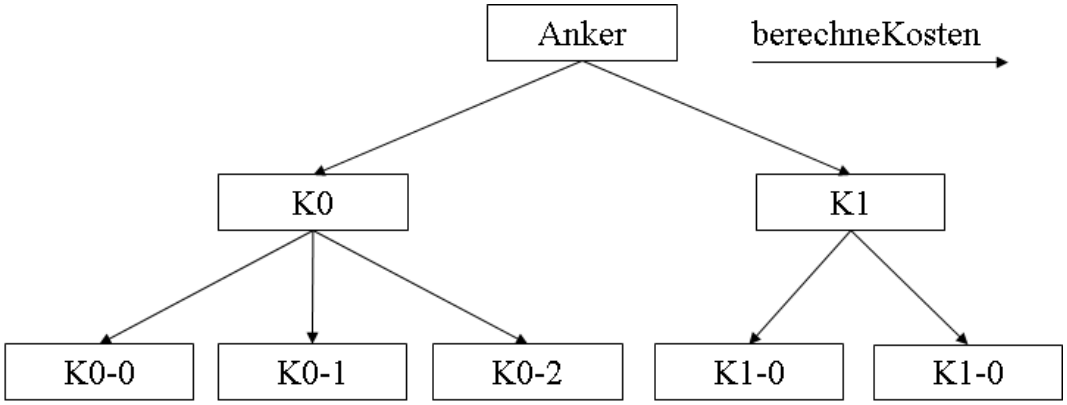


Abbildung 14: Verschicken der *berechneKosten*-Nachricht

Wenn das Versenden der Nachricht terminiert, hat jede funktionsfähige Anwendungskomponente die Nachricht empfangen und die Berechnung der internen Kosten für die Sicherung des aktuellen Zustands (IK) durchgeführt. Die Anzahl der verschickten Nachrichten entspricht der Anzahl der Anwendungskomponenten -1. In der Abbildung 15 stehen die internen Kosten neben den einzelnen Komponenten.

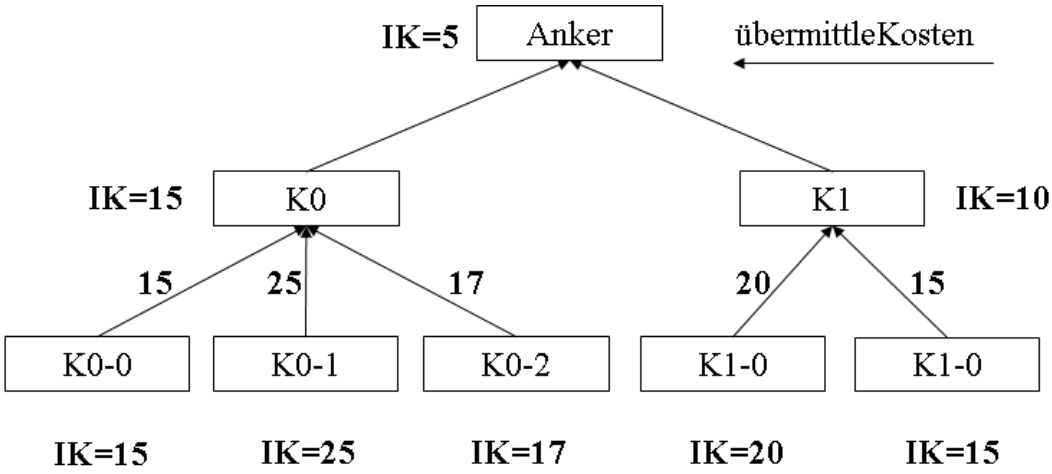


Abbildung 15: Verschicken der *übermittleKosten*-Nachricht

Nach der Berechnung der internen Kosten für jede Komponente werden die Gesamtkosten für den Wechsel der Komponenten berechnet. Die Gesamtkosten (GK) ergeben sich aus der Summe der Gesamtkosten für die instanziierten Komponenten und den internen Kosten der Komponente. Die Berechnung der Gesamtkosten für die Komponenten startet zuerst in Komponenten, welche keine weiteren Komponenten instanziiert haben. In diesem Fall gilt für die Gesamtkosten: GK = IK. Die berechneten Gesamtkosten werden über eine Nachricht (*übermittleKosten*) an die entsprechende Eltern-Komponente verschickt. Wenn eine Komponente von allen instanziierten Komponenten die *übermittleKosten*-Nachricht empfangen hat, können die Gesamtkosten für diese Komponente berechnet werden und an die entsprechende Eltern-Komponente verschickt werden. Zusätzlich zu den Gesamtkosten der Komponente werden auch die Gesamtkosten der instanziierten Komponenten mitgeschickt. Die Nachrichten werden immer in Richtung des Ankers verschickt. Der Anker berechnet nach dem Empfang aller Nachrichten die Gesamtkosten für den Wechsel der Anwendung. Der Anker besitzt weiterhin die Gesamtkosten für jede einzelne, funktionsfähige Komponente aus der Anwendungskonfiguration. Die Anzahl der verschickten *übermittleKosten*-Nachrichten = der Anzahl der *berechneKosten*-Nachrichten.

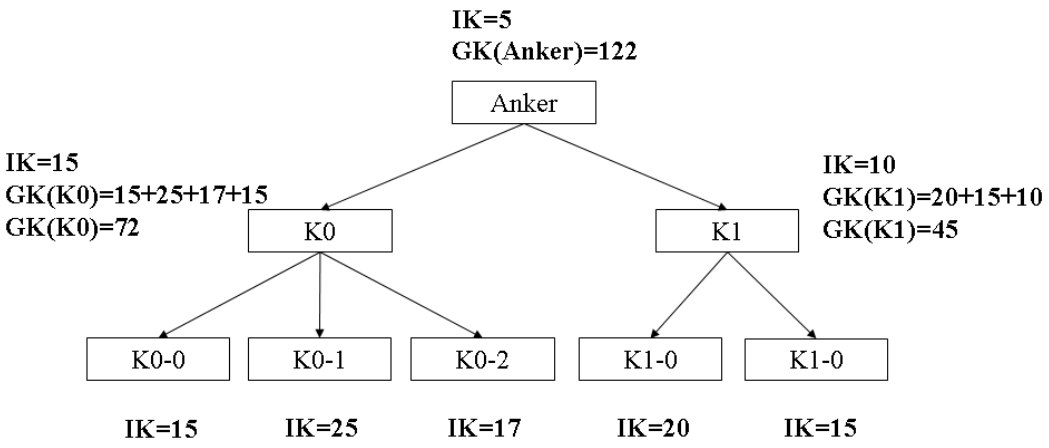


Abbildung 16: Beispiel für die Kostenverteilung einer Anwendung

Abbildung 16 ist ein Beispiel für eine Anwendung mit berechneten Kosten. Damit eine Komponente für die Rekonfiguration feststellen kann wie teuer sie ist im Vergleich zu den anderen Komponenten, wird nach der Berechnung der Kosten das Wissen über die einzelnen Komponenten vom Anker aus an die einzelnen Komponenten verteilt. Das Verschicken der (*verteileKosteninformationen*)-Nachricht verläuft analog zu den

berechneKosten-Nachrichten. Am Ende verfügt jede funktionsfähige Komponente über die Kosteninformationen der Anwendung.

4.2 Rekonfigurationsheuristiken

Die Suche nach einer PCOM-Konfiguration wird wesentlich durch die implementierte Backtracking-Strategie bestimmt. Für das Backtracking ist wichtig, dass keine mögliche Lösung übersprungen wird. In der Anwendungskomposition herrscht eine totale Ordnung zwischen den einzelnen Komponenten. Komponenten mit einer niedrigeren Ordnung werden zuerst rekonfiguriert, d.h. eine Vaterkomponente darf erst rekonfiguriert werden, wenn alle Möglichkeiten für seine Kinderkomponenten durchsucht sind.

Wenn die Backtracking-Strategie mögliche Lösungen überspringen würde, dann wäre es möglich, dass der Algorithmus gerade die einzig mögliche, gültige Konfiguration übersprungen hat und am Ende der Ausführung mit einem falschen Ergebnis (keine Konfiguration gefunden) terminiert. Das widerspricht dann der Anforderung an die Vollständigkeit des Algorithmus aus Kapitel 3.1.4. Für die im Folgenden vorgestellten Rekonfigurationsstrategien muss daher gelten, dass keine mögliche Lösung ignoriert wird.

Wie in Kapitel 3.1.3 bereits eingeführt, gibt es für die Suche nach einer Konfiguration mit möglichst geringen Kosten zwei grundsätzliche Vorgehensweisen: ein vollständiger Optimierungsalgorithmus oder geeignete Heuristiken. Ein vollständiger Optimierungsalgorithmus würde jede mögliche Konfiguration untersuchen bis die günstigste Konfiguration gefunden wurde. Da für die Suche nach der optimalen Konfiguration das NP-vollständige Problem der Suche nach einer gültigen Konfiguration im Allgemeinen mehrmals gelöst werden muss, scheidet diese Lösungsstrategie für eine Implementierung in PCOM aus. Im Rahmen dieser Diplomarbeit wurden daher verschiedene Heuristiken entworfen, welche versuchen, im Allgemeinen eine möglichst optimale Lösung zu finden, ohne dabei jede mögliche, gültige Lösung zu betrachten. Die im Folgenden vorgestellten Heuristiken versuchen die Anwendung zu rekonfigurieren, d.h. es wird nicht vom Anker aus eine neue Konfiguration gesucht, sondern versucht, für die fehlenden Anwendungsteile neue Komponenten zu instanziieren.

4.2.1 Rekonfiguration ohne Beachtung von Kosteninformationen

Die hier vorgestellte Heuristik rekonfiguriert eine Anwendung ohne Beachtung von Rekonfigurationskosten, d.h. es wird die Backtracking-Strategie aus dem ursprünglichen Konfigurationsalgorithmus verwendet. Es wird zuerst versucht für die fehlenden Anwendungsteile neue Komponenten zu instanziieren, d.h. die betroffenen Komponenteninstanzen versuchen eine gültige Belegung für die zu instanziierenden Komponenten zu finden. Schlägt die Instanziierung einer Komponente auf Grund von Ressourcenbeschränkungen auf einem Container fehl, wird im Container festgestellt, welche Komponente entfernt werden muss. Details zur Ressourcenreservierung in einem Container werden in Kapitel 5.2.1 vorgestellt. Kann eine Komponenteninstanz keine gültige Belegung für die zu instanziierenden Komponenten finden, wird ebenfalls eine Backtracking-Nachricht generiert, siehe Kapitel 5.2.1.

Zur Veranschaulichung der Konfliktauflösungs-Strategie in der Rekonfiguration ohne Beachtung von Kosteninformationen ist die totale Ordnung zwischen den Komponenten in Abbildung 17 für eine Beispielanwendung mit Pfeilen eingezeichnet. Zur Vereinfachung der Abbildung sind die Bindungen zwischen den Komponenteninstanzen nur gestrichelt eingezeichnet und anstatt der Positionsangabe sind nur die globalen Kosten für die Komponenteninstanzen eingetragen. In diesem Beispiel lassen sich die Komponenteninstanzen eindeutig durch ihre globalen Kosten identifizieren, was im Allgemeinen nicht der Fall ist. Die Anker-Komponente besitzt die höchste Priorität und die Komponente mit den Kosten 5 besitzt die niedrigste Priorität. Die Pfeile zwischen den Komponenten zeigen zu der Komponenten, welche die nächsthöhere Priorität besitzt. Wenn ein Konflikt auftritt, stehen zwei oder mehrere Komponenteninstanzen miteinander in Konflikt. In dieser Heuristik wird immer zuerst versucht die konfliktierende Komponente mit der niedrigsten Priorität zu wechseln. Da die Komponente mit der niedrigsten Priorität nicht immer die günstigste Komponente in der Anwendungskonfiguration ist, ist die Rekonfiguration ohne Beachtung von Kosteninformationen im Allgemeinen nicht immer optimal was die Kosten betrifft.

Wenn beispielsweise die Komponenten mit den Kosten 10 und 20 aus der Abbildung 17 miteinander in Konflikt stehen, würde die Komponente mit den Kosten 20 rekonfiguriert werden, da diese eine niedrigere Priorität besitzt als die Komponente mit den Kosten 10.

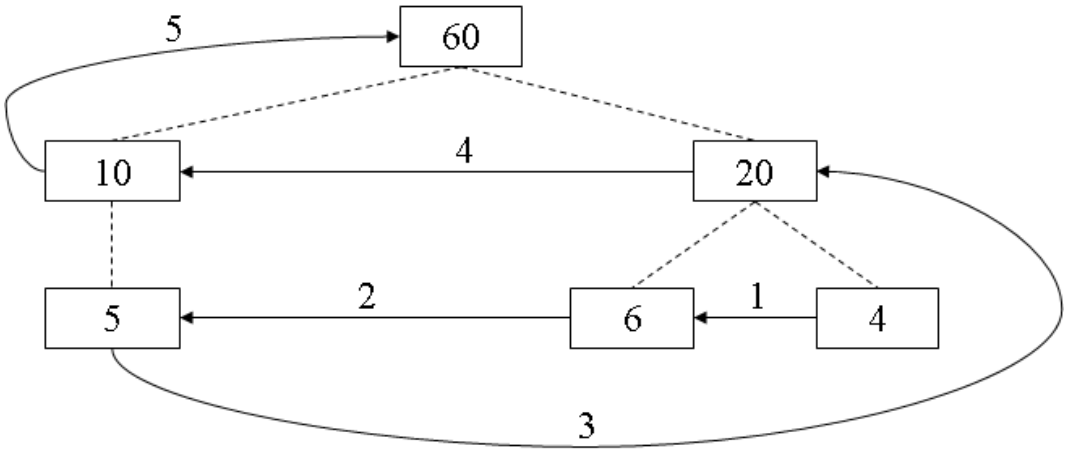


Abbildung 17: Rekonfiguration ohne Beachtung von Kosteninformationen

4.2.2 Greedy Heuristik

Die Greedy Heuristik rekonfiguriert eine Anwendung, indem bei einem Konflikt immer die günstigste Komponente zuerst rekonfiguriert wird. Um diese Strategie umzusetzen betrachtet die Greedy Heuristik neben der Position der Komponenten auch die Kosteninformationen der Komponenten. Im Unterschied zu der Rekonfiguration ohne Beachtung von Kosteninformationen wird die totale Ordnung nicht fest durch die Positionsinformation festgelegt, sondern durch die Kosteninformationen der Komponenten beeinflusst. Das bedeutet, dass die Ordnung sich dynamisch an die jeweilige Anwendung anpasst. Durch die dynamische Ordnung kann erreicht werden, dass Komponenten mit hohen Kosten später rekonfiguriert werden als Komponenten mit niedrigen Kosten.

Für die Greedy Heuristik ist es im Folgenden wichtig zu zeigen, dass die Ordnung durch den Algorithmus eingehalten wird, so dass keine Zyklen entstehen und keine Lösungsmöglichkeiten übersehen werden können.

Zuerst wird gezeigt, dass keine möglichen Lösungen durch die Greedy Heuristik übersehen werden. Wenn eine Vaterkomponente und die instanziierten Kinderkomponenten miteinander in Konflikt stehen, dann muss sichergestellt werden, dass die Vaterkomponente erst dann rekonfiguriert wird, wenn alle Rekonfigurationsmöglichkeiten für die Kinder vorher ausprobiert wurden und den Konflikt nicht auflösen konnten. Es muss also sichergestellt werden, dass eine Vaterkomponente immer eine höhere Priorität besitzt als die Kinderkomponenten. Dies wird durch das in Kapitel 4.1.3 entworfene Kostenmodell

garantiert, da jede Vaterkomponente als globale Kosten mindestens die Summe aus den globalen Kosten der Kinder und den eigenen internen Kosten (> 0) besitzt.

Die Komponenteninstanzen müssen für die Heuristik total geordnet sein und es muss sichergestellt werden, dass diese Ordnung durch die Heuristik weiterhin erhalten bleibt. Die Komponenteninstanzen werden anhand ihrer globalen Kosten geordnet, d.h. die Komponenteninstanz mit den höchsten Kosten besitzt die höchste Priorität. Komponenteninstanzen, welche während der Konfiguration neu instanziiert werden, besitzen die Kosten 0 und daher die niedrigste Priorität. Für den Fall wo Komponenteninstanzen identische Kosten besitzen, zählt die Positionsinformation als zweites Ordnungskriterium. So ist sichergestellt, dass zwischen allen Komponenteninstanzen eine totale Ordnung besteht.

Im Folgenden wird gezeigt, dass die Ordnung durch die Heuristik während der Suche nach einer Konfiguration eingehalten wird. Die Ordnung zwischen den Komponenteninstanzen wird durch die Heuristik zur Laufzeit verändert. Wenn neue Verbindungen zwischen Komponenteninstanzen erzeugt werden, muss durch die Heuristik sichergestellt werden, dass immer nur Verbindungen zwischen Instanzen höherer Ordnung $\rightarrow$ niedrigerer Ordnung aufgebaut werden.

Die Backtracking-Strategie geht in der Greedy Heuristik folgendermaßen vor: Wenn ein oder mehrere Komponenteninstanzen miteinander in Konflikt stehen, dann werden die Kosten der Komponenteninstanzen verglichen und die Komponenteninstanz rekonfiguriert, welche die niedrigsten Kosten besitzt. Falls zwei oder mehrere günstigste Komponenteninstanzen existieren entscheidet wieder die Positionsinformation welche Komponenteninstanz rekonfiguriert wird. So ist sichergestellt, dass die Ordnung durch die Heuristik erhalten bleibt.

In der folgenden Abbildung 18 ist die gleiche Beispielanwendung abgebildet wie in Kapitel 4.2.1. Zur Verdeutlichung ist die Ordnung für die Greedy Heursitik wieder mit Pfeilen eingezeichnet.

4.2.3 Mehrfache Optimierung

In [HY00] wird ein Verfahren vorgestellt, mit welchem DCSPs zumindest partiell gelöst werden können, falls zu viele Einschränkungen existieren, welche eine vollständige Lösung unmöglich machen. Das DCSP wird hier versucht in

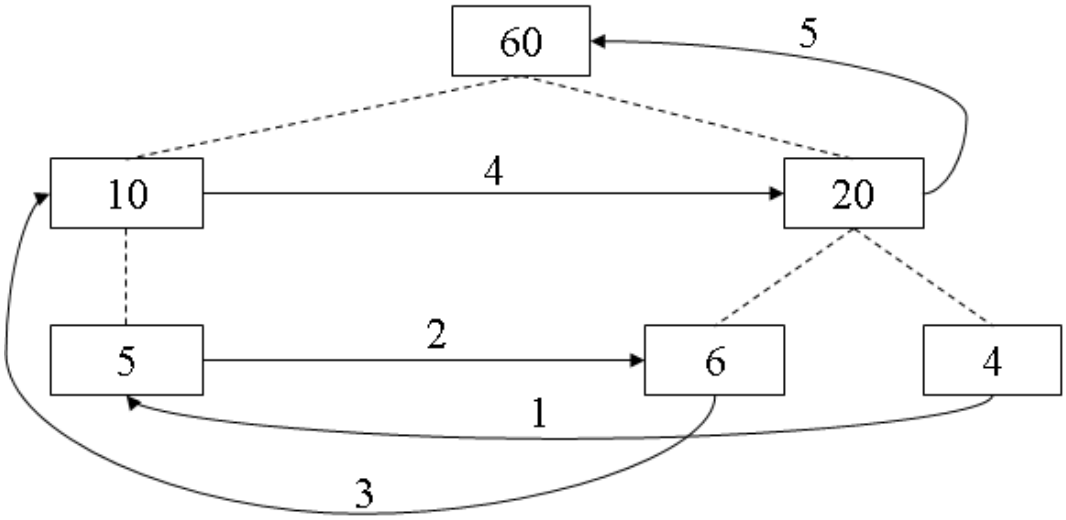


Abbildung 18: Greedy Rekonfigurationsstrategie

mehreren Durchläufen zu lösen. Grundsätzlich werden hier zwei Vorgehensweisen diskutiert:

- **Das Problem schrittweise verkleinern:**
  Für jeden Durchlauf wird eine Einschränkung entfernt, bis eine Lösung für das DCSP möglich ist.

- **Das Problem schrittweise vergrößern:**
  Hier wird das Problem bis auf eine Einschränkung verkleinert und dann mit jedem Durchlauf um eine Einschränkung vergrößert, bis keine Lösung mehr gefunden wird.

Den zweiten Ansatz kann man für die Suche nach einer möglichst optimalen Konfiguration in PCOM wie folgt modifizieren:

Im Gegensatz zu den weiter oben diskutierten Heuristiken wird im Folgenden versucht das Optimum durch mehrfaches Lösen des DCSP zu finden. Beispielsweise kann eine Benutzervorgabe sein, dass die Anwendung im Falle einer Rekonfiguration versuchen soll eine möglichst optimale, neue Konfiguration in vier Anläufen zu finden. Im ersten Durchlauf wird wie in den bisherigen Heuristiken eine gültige Konfiguration gesucht und im Anschluss bewertet. Sind die Kosten = 0, dann sind keine weiteren Durchläufe nötig. Falls die Konfiguration Kosten verursacht, wird die Komponente identifiziert, welche die größten Kosten verursacht hat. Im nächsten Durchlauf wird versucht diese Kosten zu vermeiden indem diese Komponente nicht rekonfigu-

riert wird. Am Ende des zweiten Durchlaufs sind folgende Ergebnisse möglich: (1) eine neue, gültige Konfiguration wurde gefunden, welche günstiger ist als die aus dem vorherigen Durchlauf. (2) es wurde eine teurere Konfiguration gefunden. (3) es ist keine Konfiguration möglich falls diese Komponente nicht rekonfiguriert wird.

In Fall 1 wird im dritten Durchlauf die zweite Konfiguration als Referenz versucht weiter zu optimieren. Im zweiten und dritten Fall wird versucht die zweitteuerste Komponente nicht zu rekonfigurieren und sich so weiter an das Optimum anzunähern.

5 Implementierung

5.1 Erweiterungen am PCOM-Simulator

Der PCOM-Simulator wurde im Rahmen dieser Diplomarbeit an verschiedenen Stellen erweitert. Einen großen Teil der Erweiterungen nimmt die Integration des Kostenmodells und der verschiedenen Rekonfigurationsstrategien ein. Im Folgenden werden die nötigen Erweiterungen beschrieben, um Rekonfigurationsszenarien im Simulator zu simulieren.

5.1.1 Ablauf einer Konfiguration/Rekonfiguration im PCOM-Simulator

Der PCOM-Simulator dient zur Simulation und Evaluation des Konfigurationsalgorithmus. Der Simulator erlaubt die Definition von Umgebungen mit verschiedenen Containern, welche Komponenten und Instanzen verwalten. Die Kommunikation zwischen den Containern findet über ein rundenbasiertes Nachrichtensystem statt. Im PCOM-Simulator lassen sich verschiedene Szenarien in kurzer Zeit ohne großen Setup-Aufwand simulieren.

Der Programmablauf im PCOM-Simulator läuft für eine Rekonfiguration in verschiedenen Phasen ab, siehe Abbildung 19:

Auf die einzelnen Phasen wird im Folgenden näher eingegangen:

1. Phase:
In der ersten Phase wird das Einsatzszenario für die Simulation generiert. Das Szenario besteht aus einer Umgebung mit mehreren Containern, auf denen die Komponenten verteilt werden. Die internen Kosten von Komponenten werden für die Simulation schon während der Szenarioerzeugung an die einzelnen Komponenten angehängt.

2. Phase:
Start des ursprünglichen Konfigurationsalgorithmus zur Suche nach einer initialen Konfiguration. Jede Instanz, welche zur Bereitstellung ihres Dienstes die Instanziierung weiterer Komponenten benötigt, hat für jede Komponente möglicherweise mehrere Optionen zur Auswahl. Die Optionen für die Komponente werden durchnummeriert (angefangen bei 0) und der Reihe nach durchprobiert, d.h. es wird zuerst versucht die nullte Option zu instanziieren. Falls keine gültige Konfiguration gefunden werden kann, wird die nächste Option versucht zu instanziieren.

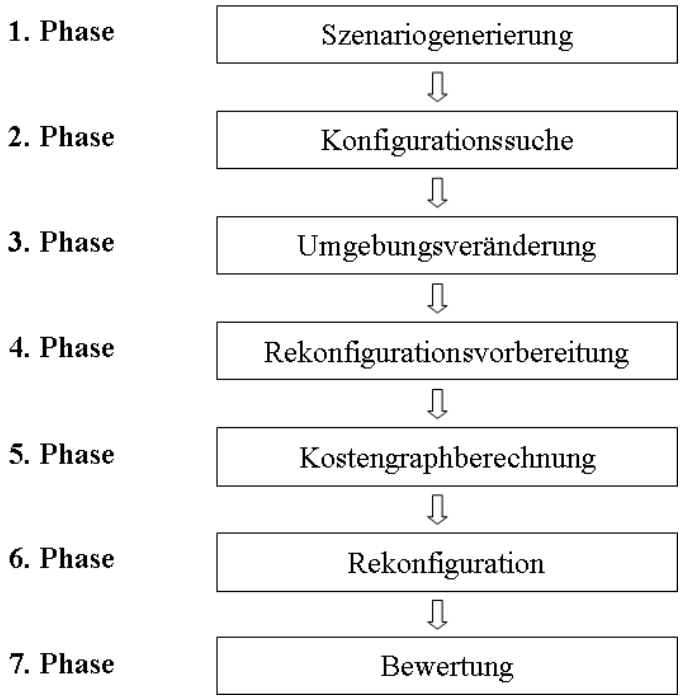


Abbildung 19: Simulationsablauf im PCOM-Simulator

In der ursprünglichen Version des PCOM-Simulator war es nicht nötig die Informationen über die Optionen lokal in der Instanz zu speichern. Da der Rekonfigurationsalgorithmus allerdings, vom Anker ausgehend, genau die Instanzen initialisieren soll, welche noch Teil der Anwendung sind, ist es nötig, den Simulator an dieser Stelle zu erweitern. Jede Instanz besitzt daher jetzt einen Vektor pro Komponentenabhängigkeit, in welchem Verweise auf die Komponentenoptionen gespeichert sind. Dieser Vektor wird während der Instanzerzeugung erstellt. In der Vorbereitungsphase für die Rekonfiguration wird es dann unter Umständen nötig, den Vektor neu aufzubauen (da Komponenten weggefallen sind) und den Vektor umzusortieren, siehe 4. Phase.

Wenn der Algorithmus erfolgreich terminiert, wurde eine gültige Konfiguration gefunden und die entsprechenden Komponenteninstanzen werden in den Containern erzeugt.

3. Phase:
Die Phasen drei bis fünf dienen zur Vorbereitung für den Rekonfigurationsalgorithmus. In der dritten Phase wird die Umgebung manipuliert, d.h. ein oder mehrere Container werden aus der Umgebung entfernt. Die Folge für die Anwendung ist, dass möglicherweise Anwendungsteile einer Anwendung nicht

mehr vorhanden sind, da diese auf den betroffenen Containern ausgeführt wurden. Der Simulator muss nach der Entfernung der betroffenen Container aus dem Szenario auch noch möglicherweise weitere Anwendungsinstanzen entfernen. Der Grund hierfür liegt darin, dass die verlorenen Instanzen möglicherweise eine Vaterkomponente von einem Teilbaum waren und damit ist dann der komplette Teilbaum für die Anwendung nicht mehr verfügbar und die entsprechenden Kinderinstanzen werden in der Umgebung wieder freigegeben, da sie keine Bindung mehr zu ihrer Vaterkomponente haben.

4. Phase:
Hier werden unter anderem die Positionsinformationen der einzelnen Anwendungsinstanzen angepasst. Die Positionsanpassung ist nötig, da das Szenario sich verändert hat. Die funktionsfähigen Instanzen werden in der sechsten Phase durch den Rekonfigurationsalgorithmus neu initialisiert. Da der Algorithmus zuerst die erste Option für eine Abhängigkeit auswählt und die funktionsfähigen Instanzen aus der initialen Konfiguration möglicherweise nicht die erste Option während der Konfiguration waren, müssen die Positionsinformationen der Instanzen angepasst werden (setzen des Optionswert in der Position auf 0), so dass diese Instanzen für die Rekonfiguration zuerst gewählt werden.

In Abbildung 20 ist ein Beispiel für die Anpassung der Positionsinformationen abgebildet. Die Instanz mit der Positionsinformation [0](2) [0](1) wird während der Positionsanpassung in die Instanz [0](0) [0](0) umbenannt. Die Positionsinformation setzt sich grob aus zwei Teilen zusammen: der Positionsinformation des Vaters, im Beispiel [0](2). An diese Positionsinformation wird dann die eigene Information angehängt, z.B. [0](1). Die Zahl in der eckigen Klammer gibt an, welche Abhängigkeit mit dieser Instanz erfüllt wird und die Zahl in der runden Klammer gibt an, welche Option diese Instanz darstellt.

Weiterhin besitzt jede Instanz für jede ihrer Abhängigkeiten einen Vektor mit Optionen für die Abhängigkeit. Da die Informationen aus diesem Vektor mit großer Wahrscheinlichkeit nicht mehr konsistent sind, da in der 3. Phase die Umgebung manipuliert wurde, muss der Vektor für jede funktionsfähige Instanz neu berechnet werden.

Damit der Rekonfigurationsalgorithmus beim Start auch die Instanzen initialisiert, welche auf Option 0 gesetzt wurden, müssen die Vektoren für die einzelnen Abhängigkeiten vor dem Start noch umsortiert werden, damit sichergestellt ist, dass alle noch lauffähigen Instanzen durch den Rekonfigurationsalgorithmus als erste Option gewählt werden. Beispielsweise müssen in der Instanz [0](0) aus Abbildung 20 die Vektoren für die Abhängigkeiten 0

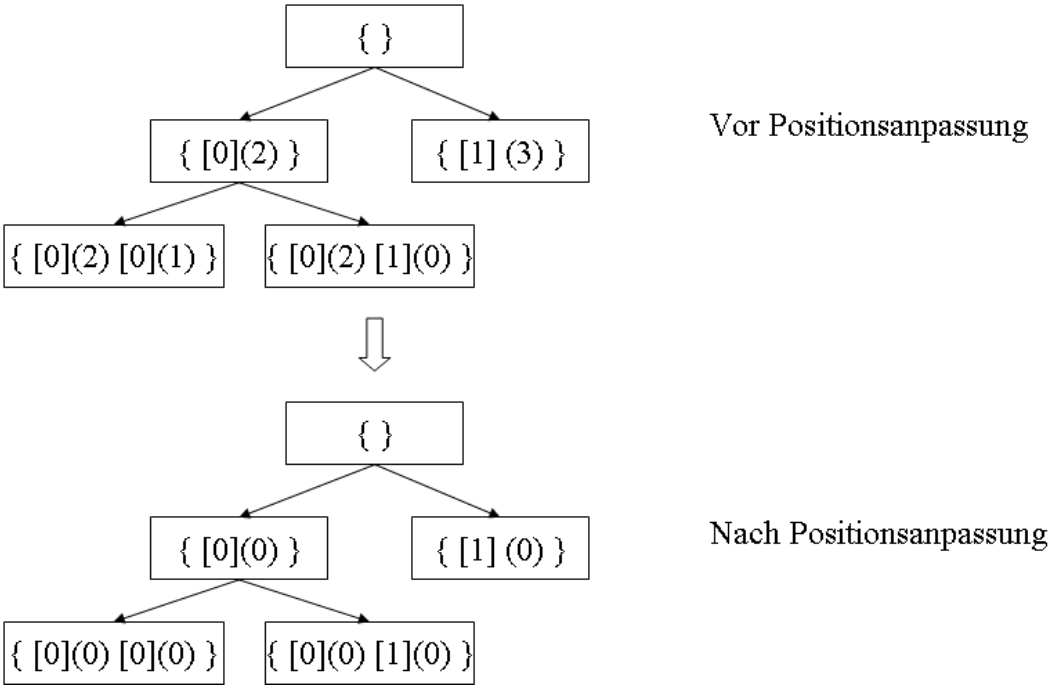


Abbildung 20: Anpassung der Instanzenpositionen

und 1 neu aufgebaut und umsortiert werden, d.h. der Eintrag für die Instanz [0](0) [0](0) wird an die erste Stelle im Vektor für Abhängigkeit 0 verschoben und [0](0) [1](0) an die erste Stelle im Vektor für Abhängigkeit 1.

5. Phase:
Der Verlust von Komponenteninstanzen in der 3. Phase leitet, nach der Rekonfigurationsvorbereitung in Phase 4, die Berechnung der Komponentenkosten ein. Die Berechnung der Komponentenkosten startet in der Ankerinstanz und wird an alle direkten Kinderinstanzen über Nachrichten weitergeleitet. Diese Kinderinstanzen verschicken dann wieder Nachrichten an ihre Kinder, etc. Das Kostenmodell wurde in Kapitel 4.1.3 vorgestellt und in Kapitel 5.1.2 wird auf die Integration und Implementierung genauer eingegangen.

Am Ende dieser Phase sind die Kosten für jede Instanz berechnet. Jede Instanz verschickt zusammen mit ihren globalen Kosten noch einen Kostenvektor, welcher die globalen Kosten aller Kinder enthält, d.h. die Ankerinstanz besitzt globales Wissen über alle lauffähigen Kinderinstanzen. Dieses Wissen wird danach an alle Kinder verteilt, so dass zu Beginn der 6. Phase jede Kinderinstanz die Kosteninformationen für die einzelnen Anwendungsinstanzen besitzt.

6. Phase:
In dieser Phase wird der Rekonfigurationsalgorithmus gestartet und berechnet eine neue, gültige Konfiguration (in Abhängigkeit der verwendeten Rekonfigurationsstrategie). Nach der Konfigurationsberechnung werden die Gesamtkosten für die gefundene Konfiguration berechnet.

7. Phase:
Abschließend wird die gefundenen Lösung bewertet. Damit eine Bewertung der gefundenen Lösung möglich ist, werden alle gültigen Konfigurationen des Szenario berechnet und die Kosten der Lösungen ermittelt. So ist nun eine abschließende Bewertung der gefundenen Rekonfiguration möglich.

5.1.2 Integration des Kostenmodells

Das Kostenmodell spiegelt die Rekonfigurationskosten für die einzelnen Komponenteninstanzen innerhalb der unterbrochenen Anwendung wider. Der Entwurf des Kostenmodells wurde ich Kapitel 4.1.3 ausführlich vorgestellt. Für die Integration des Kostenmodells werden im PCOM-Simulator folgende Nachrichten zusätzlich integriert:

- **starteRekonfiguration-Nachricht:**
  Wenn die Anwendung rekonfiguriert werden muss, wird an die Ankerinstanz eine starteRekonfiguration-Nachricht geschickt, welche die Berechnung der Kosten anstößt.

- **berechneKosten-Nachricht:**
  Da die Kosten für eine Instanz erst berechnet werden können, wenn bekannt ist, wie hoch die direkten Kinderinstanzkosten sind, wird an jede direkte Kinderinstanz eine berechneKosten-Nachricht verschickt. Die Empfänger der berechneKosten-Nachricht verschicken wiederum ebenfalls berechneKosten-Nachrichten an ihre direkten Kinderinstanzen. Jede funktionsfähige Instanz erhält eine berechneKosten-Nachricht, da im PCOM-Simulator keine Nachrichten verloren gehen. Der Empfang einer berechneKosten-Nachricht stößt die Berechnung der internen Kosten an.

- **übermittleKosten-Nachricht:**
  Die globalen Kosten werden über die übermittleKosten-Nachricht an die Vaterinstanz verschickt. Die globalen Kosten können erst berechnet werden, wenn die globalen Kosten der Kinder bekannt sind (über eine übermittleKosten-Nachricht empfangen oder keine Kinder existieren, in diesem Fall gilt globale Kosten = interne Kosten.

- **verteileKosten-Nachricht:**
  Wenn alle Kosten für die einzelnen Instanzen berechnet sind, werden die Kosteninformationen an alle Instanzen mit verteileKosten-Nachrichten verteilt

Während der Suche nach einer Konfiguration werden weitere Komponenteninstanzen erzeugt. In jeder neuen Komponenteninstanz wird mit der Instanziierung auch die Kosteninformationen aus dem Kostenmodell gespeichert.

5.2 Integration der Rekonfigurationsstrategien

5.2.1 Rekonfiguration ohne Beachtung von Kosteninformationen

Diese Heuristik sucht eine neue Konfiguration für die Anwendung ohne Beachtung der Kosteninformationen. Damit die gefundene Lösung auf ihre verursachten Kosten hin bewertet werden kann, werden die Kosteninformationen in jeder Instanz gespeichert. In der sechsten Phase (siehe Kapitel 5.1.1) startet die Suche nach einer neuen Konfiguration. Es werden die Instanzen identifiziert, welche auf Grund der veränderten Umgebung möglicherweise die Bindung zu anderen Instanzen verloren haben. An diese Instanzen wird eine Update-Nachricht versendet, welche eine Rekonfiguration für diese Instanz auslöst, d.h. die Instanz versucht eine neue Belegung für die zu instanziierenden Komponenten zu finden. Der Algorithmus läuft danach analog zu dem in [Rei04] vorgestellten Konfigurationsalgorithmus ab.

5.2.2 Greedy Heuristik

Die Greedy Heuristik sucht eine neue Konfiguration unter Beachtung der Kosteninformationen aus dem Kostenmodell. Die Kosteninformationen spielen zu verschiedenen Zeitpunkten eine wichtige Rolle bei der Suche nach einer Konfiguration.

Auf Grund der aktuellen Implementierung des PCOM-Simulator ist es nötig die in Kapitel 4.2.2 vorgestellte Heuristik zu modifizieren. Es dürfen immer nur Verbindungen zwischen Instanzen höherer Ordnung → niedrigerer Ordnung aufgebaut werden, damit keine Zyklen in der Ordnung entstehen. Die folgende Abbildung 21 demonstriert die im PCOM-Simulator implementierte Vorgehensweise für den Aufbau von Verbindungen.

Im PCOM-Simulator hält jede Instanz für die einzelnen Abhängigkeiten eine

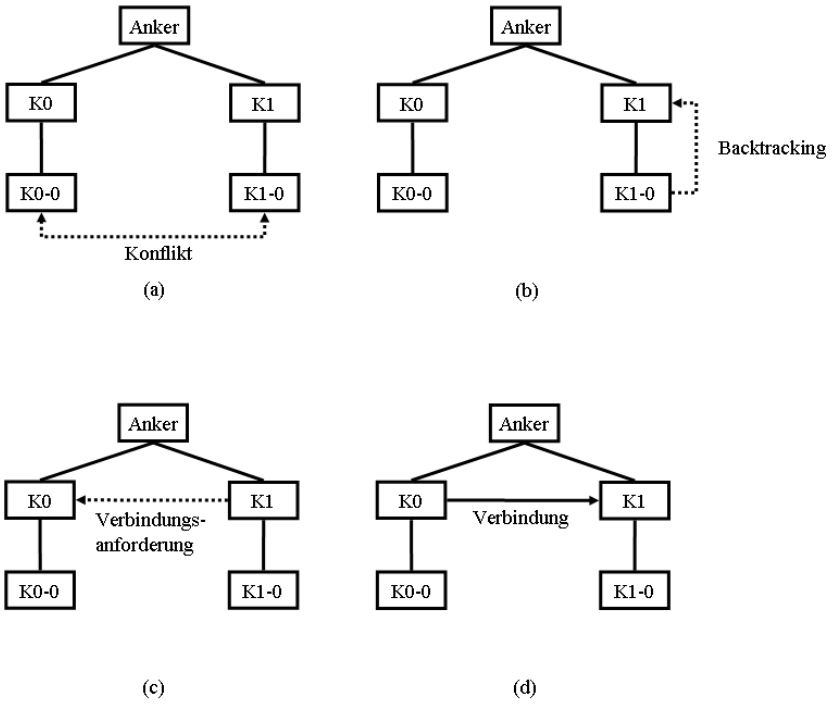


Abbildung 21: Verbindungsaufbau

Liste mit Instanzen, welche im Fall eine veränderten Variablenbelegung benachrichtigt werden müssen. Jeder Eintrag in dieser Liste symbolisiert eine Verbindung innerhalb des DCSP. Die Verbindung wird allerdings nicht zu der konfliktierenden Komponente eingezogen, sondern zwischen den entsprechenden Vaterkomponenten. In Abbildung 21 (a) konfliktieren beispielsweise die Komponenten K0-0 und K1-0 miteinander. Da die Komponente K1-0 die Komponente mit der niedrigeren Priorität ist, sendet diese in (b) eine Backtracking-Nachricht mit der Konfliktmenge an ihre Vaterkomponente. Die Vaterkomponente versendet in (c) eine Verbindungsanforderung an die Vaterkomponente von K0-0 und in (d) wird eine neue Verbindung zwischen den Komponenten K0 und K1 eingezogen. Die Ordnung ist weiterhin korrekt und zyklenfrei, da K0 eine höhere Priorität als K1 besitzt.

Wenn nun die Greedy Heuristik ohne Modifikation in den PCOM-Simulator implementiert wird, können durch den Verbindungsaufbau Zyklen entstehen. Abbildung 22 zeigt ein Beispiel für die Greedy Heuristik. Die Komponenten können über ihre Rekonfigurationskosten identifiziert werden. In (a) konfliktieren die Komponenten mit den Kosten 10 und 15. Die Greedy Heuristik rekonfiguriert in (b) die Komponente mit den niedrigsten Kosten. Die Vater-

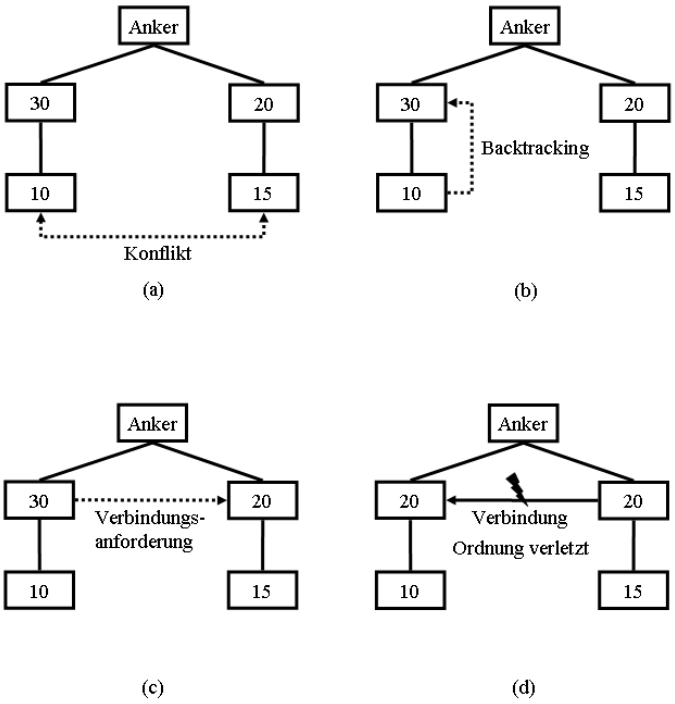


Abbildung 22: Unmodifizierte Greedy Heuristik

komponente von der Komponente 10 versendet in (c) eine Verbindungsanforderung an die andere Vaterkomponente. In (d) wird dann eine Verbindung zwischen der Komponente 20 und 30 aufgebaut. Diese Verbindung verletzt die Bedingung, dass nur Verbindungen zwischen Komponenten höherer Ordnung → niedrigerer Ordnung aufgebaut werden.

Damit die Heuristik die Ordnung immer einhält und nur Verbindungen zwischen Komponenten höherer Ordnung → niedrigerer Ordnung aufbaut, muss die Greedy Heuristik folgendermaßen modifiziert werden. Wenn ein oder mehrere Komponenteninstanzen miteinander in Konflikt stehen, dann werden die Kosten der Vaterkomponenten verglichen und die Komponente rekonfiguriert, welche den günstigsten Vater besitzt. Falls zwei oder mehrere günstigste Väter existieren entscheidet wieder die Positionsinformation. Falls mehrere Komponenten des günstigsten Vaters miteinander in Konflikt stehen, entscheiden die Kosten der Kinder darüber, welche Komponente rekonfiguriert wird. Im Allgemeinen ist die Komponente mit dem günstigsten Vater nicht immer die günstigste Komponente, allerdings ist so gewährleistet, dass immer nur Verbindungen zwischen Komponenten höherer Ordnung → niedrigerer Ordnung aufgebaut werden. Zur Verdeutlichung ist in Abbildung 23

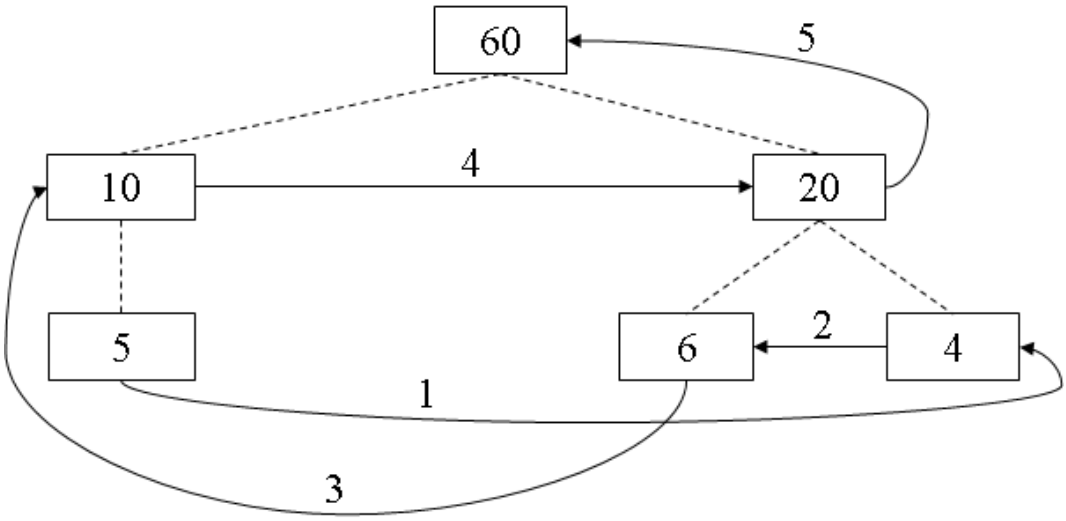
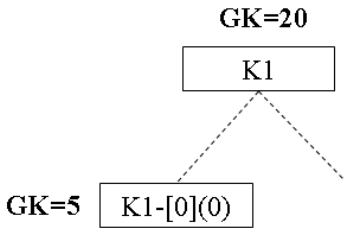


Abbildung 23: Modifizierte Greedy Heuristik

ein weiteres Beispiel für die Backtracking-Strategie abgebildet.

Wie in der Rekonfiguration ohne Beachtung von Kosteninformationen startet die Suche nach einer neuen Konfiguration in der sechsten Phase (siehe Kapitel 5.1.1). An die betroffenen Instanzen werden Update-Nachrichten versendet, welche den Rekonfigurationsalgorithmus starten. Die Suche nach einer neuen Belegung unterscheidet sich von der Suche in der Rekonfiguration ohne Beachtung von Kosteninformationen. Im Fall der Greedy Heuristik werden alle möglichen, gültigen Belegungen ermittelt und miteinander verglichen. Eine Belegung ist dann gültig, wenn sie keine der Einschränkungen verletzt. Die Menge der gültigen Belegungen werden danach bewertet. Für jede Belegung wird untersucht wie viel Kosten sie verursachen würde. Ziel ist die Auswahl einer Belegung, welche möglichst geringe Kosten verursacht, d.h. es wird versucht bereits instanziierte Komponenten beizubehalten. Falls mehrere Belegungen mit minimalen Kosten existieren, dann wird durch die Heuristik eine ausgewählt. Die folgende Abbildung ist ein Beispiel für die Suche nach einer Belegung. In Abbildung 24 (a) sucht die Instanz K1 eine Belegung für die zwei Abhängigkeiten. Für jede Abhängigkeit hat die Instanz zwei Optionen, wobei für die erste Abhängigkeit die erste Option (K1-[0](0)) noch gebunden ist, da sie Teil der Anwendung war. Die möglichen Belegungen mit den zugeordneten Kosten stehen in Abbildung 24 (b).

In dem obigen Bespiel existieren zwei Belegungen mit Kosten 5 und zwei Belegungen mit Kosten 0. Die Instanz wählt nun eine der zwei Belegungen mit Kosten = 0.



Abhängigkeit 0	Abhängigkeit 1	Belegungskosten
Option 0	Option 0	0
Option 0	Option 1	0
Option 1	Option 0	5
Option 1	Option 1	5

(a) (b)

Abbildung 24: Suche nach einer Belegung

Im Rekonfigurationsalgorithmus wird das Backtracking aus zwei verschiedenen Gründen angestoßen:

- **Ressourcenkonflikt in einem Container:**
  Wenn eine Ressource in einem Container instanziiert wird, dann wird nach der Instanziierung eine Ressourcenvalidierung in dem Container ausgeführt. Jeder Container verfügt über eine begrenzte Menge an einer Ressource. Die implementierte Greedy Heuristik geht davon aus, dass jede Instanz nur Bedarf an dieser einen Art von Ressource hat. Falls die Summe der angeforderten Ressourcen die Menge an verfügbaren Ressourcen übersteigt, tritt ein Konflikt auf, d.h. es müssen ein oder mehrere Instanzen in dem Container deaktiviert werden, damit die Ressourcenbeschränkungen in dem Container wieder erfüllt sind.

  Für die Ressourcenvalidierung werden die Instanzen nach ihrer Ordnung sortiert (die Instanz mit der höchsten Ordnung zuerst und dann absteigend die zweithöchste Instanz,...). Die Ordnung wird (wie in Kapitel 4.2.2 definiert) primär durch die Kosten der Vaterinstanzen bestimmt. Nach der Ordnung wird für die einzelnen Instanzen versucht die benötigen Ressourcen zu reservieren. Falls für eine Instanz die Reservierung fehlschlägt, wird die Instanz deaktiviert und eine Backtracking-Nachricht erzeugt. Die Backtracking-Nachricht wird an den Vater der Instanz versendet.

- **Belegungskonflikt in einer Instanz:**
  Wenn eine Instanz auf Grund von Ressourcenkonflikten oder mangelnder Verfügbarkeit von geeigneten Komponenten keine gültige Belegung für die zu instanziierenden Komponenten finden kann, wird eine Backtracking-Nachricht erzeugt. Die konfliktierenden Instanzen werden nach ihrer Ordnung sortiert und die Instanz mit der kleinsten Ordnung

wird rekonfiguriert, d.h. die Backtracking-Nachricht wird an den Vater der Instanz versendet.

6 Evaluation

In diesem Kapitel werden die in Kapitel 4.2 entworfenen Heuristiken evaluiert. Zuerst werden Vorüberlegungen diskutiert und danach Szenarien entworfen, in denen die Heuristiken zur Suche nach einer neuen Konfiguration eingesetzt werden. Die Ergebnisse werden abschließend dokumentiert.

6.1 Vorüberlegungen

Vor dem Entwurf von Szenarien für die Evaluation werden zuerst einige Überlegungen über die in Kapitel 4.2 entworfenen Heuristiken diskutiert.

Wie in Kapitel 3 bereits erläutert, gibt es für die Heuristiken mehrere Anforderungen, welche sie erfüllen müssen. Auf die folgenden Anforderungen wird für die Evaluation besonders Wert gelegt:

- **Finden einer Lösung, falls eine oder mehrere existieren:**
  Für die Evaluation werden Szenarien entworfen, welche verschiedene Umgebungen repräsentieren. Die entworfenen Heuristiken dürfen keine mögliche Lösung übersehen oder ignorieren. Falls keine Lösung existiert terminiert die Heuristik und meldet, dass keine Konfiguration gefunden werden konnte.

- **Finden einer Lösung, welche möglichst geringe Anpassungskosten verursacht:**
  Die von den Heuristiken ermittelte Konfiguration wird auf ihre Anpassungskosten hin untersucht. Die gefundene Konfiguration soll möglichst geringe Kosten verursachen. Eine vollständige, umfassende Bewertung der Konfiguration ist nur möglich, wenn alle möglichen Konfigurationen des Szenario bekannt sind, d.h. es werden für die Evaluation alle möglichen Konfigurationen berechnet und auf die verursachten Kosten hin untersucht. In Kapitel 6.2 werden Bewertungskriterien aufgestellt, welche die gefundene Konfiguration bewerten.

In Kapiel 6.3 werden zwei unterschiedliche Arten von Szenarien für die Evaluation entworfen. Das Ziel der entworfenen Szenarien ist die Demonstration der Vor- und Nachteile der Heuristiken. Weiterhin gilt es zu zeigen, dass in der Mehrzahl der Durchläufe die Greedy Heuristik eine Konfiguration ermittelt, welche sich näher am Optimum befindet als die Rekonfiguration ohne Beachtung von Kostenfaktoren.

Die Evaluation soll außerdem zeigen, dass die implementierte Greedy Heuristik nicht immer die optimale Konfiguration ermittelt. Der Grund dafür liegt in der Konfliktauflösungsstrategie der Heuristik. Die Greedy Heuristik versucht die Kosten für die Rekonfiguration möglichst niedrig zu halten, indem immer zuerst Komponenten mit niedrigen Kosten rekonfiguriert werden. Dieser Ansatz führt wie in Kapitel 6.3.2 gezeigt wird sehr oft zu einer optimalen Konfiguration. Der Ansatz hat allerdings auch Schwächen, welche in dieser Vorüberlegung anhand einer Beispielanwendung diskutiert wird.

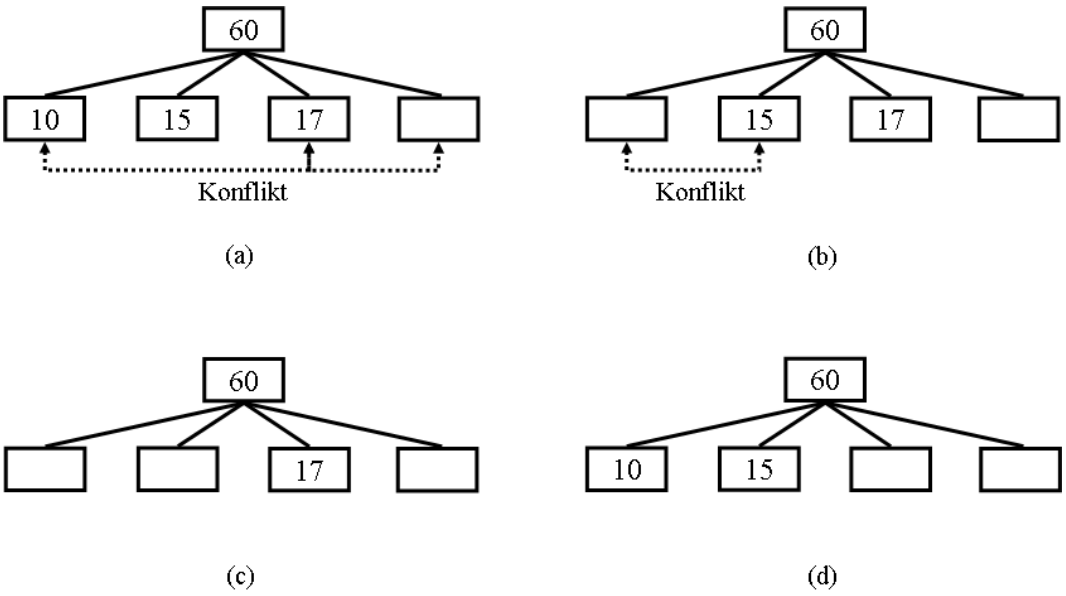


Abbildung 25: Greedy Heuristik Szenario

Die Abbildung 25 stellt eine Beispielanwendung dar, welche rekonfiguriert werden muss. In Abbildung 25 (a) muss für die vierte Abhängigkeit eine neue Komponente gefunden werden (die Komponente ohne Kosteninformation). Die neue Komponente für diese Abhängigkeit hat einen Konflikt mit den bestehenden Anwendungskomponenten. Entweder muss die Komponente mit den Kosten 10 oder die Komponente mit den Kosten 17 rekonfiguriert werden. Die Greedy Heuristik wählt aus den konfliktierenden Komponenten die Komponente mit den niedrigeren Kosten aus (die Komponente mit den Kosten 10). Die Rekonfiguration der Komponente mit den Kosten 10 verursacht eine erneute Rekonfiguration, da ein weiterer Konflikt entsteht, siehe Abbildung 25 (b). Um diesen neuen Konflikt aufzulösen, muss die Komponente mit den Kosten 15 rekonfiguriert werden. Das Endergebnis dieser Rekonfiguration ist in Abbildung 25 (c) abgebildet und hat Kosten von insgesamt 25 verursacht. Hätte die Heuristik für die erste Rekonfiguration die Kompo-

nente mit den höheren Kosten rekonfiguriert, dann wären die Kosten mit 17 niedriger gewesen, siehe Abbildung 25 (d), da in diesem Beispielszenario die Komponente mit den Kosten 17 ohne neue Konflikte ersetzt werden kann.

Schlussfolgerungen:

- Die Rekonfiguration mehrerer Komponenten mit niedrigen Kosten muss nicht besser sein als die Rekonfiguration weniger, teurer Komponenten.
- Über die Kosteninformationen kann die Rekonfiguration der Anwendung in der Greedy Heuristik beeinflusst werden. Beispielsweise kann an eine Anwendungskomponente gezielt hohe Kosten vergeben werden, damit zuerst alle anderen Anwendungskonfigurationen durch die Heuristik untersucht werden. Auf diese Weise kann unter anderem die Rekonfiguration von Ein- und Ausgabekomponenten verzögert werden.

6.2 Bewertungskriterien

6.2.1 Verursachte Kosten

Für die gefundene Konfiguration wird untersucht, wie viel Kosten sie verursacht. Um die Güte der gefundenen Konfiguration zu beurteilen müssen alle anderen möglichen Konfigurationen gesucht und auf ihre Kosten hin untersucht werden. So kann festgestellt werden wie nahe die gefundene Lösung am Optimum ist. Die möglichen Konfigurationen werden nach ihren Kosten geordnet. Interessant ist die Position der gefundenen Konfiguration innerhalb des Lösungsspektrum, d.h. wie viele Lösungen sind besser/schlechter als die gefundene Konfiguration und wie sind diese über den Lösungsraum hin verteilt.

6.3 Szenarien

Für die Evaluation der Heuristiken werden verschiedene Szenarien entworfen. Die Szenarien werden von verschiedenen Parametern beeinflusst.

6.3.1 Parameter

- **Anwendungsstruktur:**
  Für die Evaluation der Heuristiken wird eine Anwendung erstellt. Die

Anwendung setzt sich aus mehreren Komponenten zusammen. Die Ankerkomponente bildet die Schnittstelle zur Anwendung, d.h. diese Komponente bietet dem Benutzer als Dienst die Anwendung und benötigt die Bindung weiterer Komponenten. Jede Komponente bietet anderen Komponenten einen Dienst über eine Schnittstelle an. Um den Dienst anbieten zu können, benötigt eine Komponente möglicherweise die angebotenen Dienst weiterer Komponenten, welche über entsprechende Schnittstellen eingebunden werden.

Für die folgenden Szenarien werden Anwendungen mit einer vollständigen Baumstruktur aufgebaut. Es gibt zwei Parameter, welche die Struktur des Baums bestimmen: Tiefe und Breite. Die Tiefe gibt die Anzahl der Komponenten auf dem Weg von der Wurzel zu einem Blatt an. Die Breite die gibt die Anzahl der Kinder pro Knoten an. Die Abbildung 26 ist ein Beispiel für eine Anwendung mit den Parametern Tiefe: 3, Breite: 2. Für die Szenarien wird die Ankerkomponente auf einen extra reservierten Container gelegt und die anderen Komponenten werden zufällig über die restlichen Container hinweg verteilt.

- **Anzahl der Duplikate:**
  Die Anzahl der Duplikate erhöht die Anzahl der theoretisch möglichen Konfigurationen ohne Betrachtung der Ressourcenbeschränkungen auf den Geräten. Wenn ein Szenario ohne duplizierte Komponenten erzeugt wird, dann ist für eine Anwendung nur eine Konfiguration theoretisch möglich.

- **Ressourcenbeschränkungen:**
  Die Ressourcenbeschränkungen auf den Geräten beeinflussen die Anzahl der möglichen, gültigen Konfigurationen. In der Regel ist das eine Teilmenge der theoretisch möglichen Konfigurationen.

- **Anzahl der Container:**
  Die Anzahl der Container legt fest, wie viele Geräte in der Umgebung für die Konfiguration zur Verfügung stehen.

- **Kostenverteilung:**
  Die Verteilung der Kosten für die Komponenteninstanzen bestimmt die Backtrackingstrategie in der Greedy Heuristik.

- **Umgebungsmanipulation:**
  Nachdem der ursprüngliche Algorithmus zur Suche nach einer initialen Konfiguration eine Anwendungskonfiguration ermittelt hat, wird die Umgebung manipuliert, so dass eine Rekonfiguration der Anwendungskonfiguration erfolgen muss. Die Umgebungsmanipulation kann

auf verschiedene Arten erfolgen, welche in den einzelnen Szenarien genauer beschrieben werden.

6.3.2 1. Szenario

Die Evaluation der Heuristiken in diesem Szenario soll folgende Eigenschaften der Heuristiken demonstrieren:

- Die Rekonfiguration einer Anwendung ist in der Mehrzahl der Simulationsläufe die bessere Wahl als die Suche nach einer neuen, initialen Konfiguration.
- Die Greedy Heuristik findet in der Mehrzahl der Simulationsläufe eine Konfiguration, welche identisch oder näher am Optimum ist als die Rekonfiguration ohne Betrachtung der Kosteninformationen.
- Die Rekonfiguration von Komponenten mit einer hohen Priorität verursacht in der Regel höhere Kosten als die Rekonfiguration einer Komponente mit niedrigerer Priorität.

Für die Messungen wird eine Anwendungskompositionen mit den Parametern: Tiefe 3, Breite 2 generiert, siehe Abbildung 26.

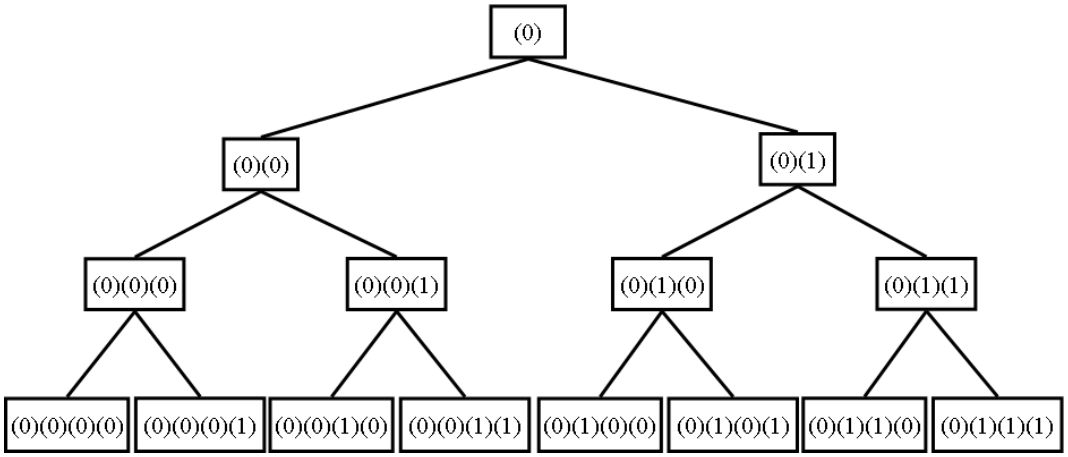


Abbildung 26: 1. Szenario

Die generierte Anwendung setzt sich aus 15 Komponenten zusammen, welche in der Umgebung auf 15 Container zufällig verteilt werden. Die Anzahl der Container wird in diesem Szenario auf die Anzahl der Anwendungskomponenten gesetzt, damit einzelne Komponenten gezielt aus der Anwendung

entfernt werden können. Jeder Container in der Umgebung verfügt über ausreichend Ressourcen für die Instanziierung von genau einer Komponente. Damit verschiedene Szenarien simuliert werden können, wird zusätzlich für jede Komponente (außer der Ankerkomponente) ein Duplikat erzeugt und zufällig über die Container verteilt, so dass jeder Container zwei Komponenten zur Verfügung stellt. Über die zufällige Verteilung der Duplikate in der Umgebung werden so für die Simulation verschiedene Szenarien erzeugt. Für die Demonstration der Vorteile der Heuristiken in dieser Evaluation wurden 10 zufällige Szenarien mit den oben beschriebenen Parametern erstellt. Für jedes Szenario wird untersucht, wie sich der Wegfall der einzelnen Container auf die Rekonfiguration auswirkt. Die Umgebung wird folgendermaßen manipuliert: (1) ein Container wird aus der Umgebung entfernt. Die instanziierten Komponenten auf diesem Container werden entfernt. Der Verlust eines Containers führt zur Rekonfiguration der Anwendung. (2) Der entfernte Container wird als neuer Container an das Ende der Containerliste im Simulator hinzugefügt. So ist sichergestellt, dass mindestens eine Lösung existiert, mit den Kosten von 0.

Für jedes Szenario werden drei Kostenverteilungsfunktionen simuliert:

- **Konstante Kostenverteilung:**
  Die internen Kosten der einzelnen Komponenten werden in diesem Fall auf konstant 10 gesetzt in jeder Komponente.

- **Absteigende Kostenverteilung:**
  In diesem Fall werden die internen Kosten der Ankerkomponente auf 15 gesetzt. Die internen Kosten der weiteren Komponenten werden absteigend, entlang der Priorität verteilt, d.h. die internen Kosten der Komponente mit der zweithöchsten Ordnung wird beispielsweise auf 14 gesetzt und für die Komponente mit der niedrigsten Ordnung (in dieser Anwendung die Komponente (0)(1)(1)(1)) bedeutet dies, dass die internen Kosten auf 1 gesetzt werden.

- **Aufsteigende Kostenverteilung:**
  Hier wird die Kostenverteilung genau entgegengesetzt zur absteigenden Kostenverteilung gesetzt.

Da jedes Szenario mit diesen drei Kostenverteilungsfunktionen simuliert wird, ergeben sich daher 3×10 Szenarien für die Simulation der Heuristiken. Jedes Szenario hat eine bestimmte Anzahl an möglichen Konfigurationen und die gefundenen Lösungen werden in verschiedene Kategorien eingeteilt: optimale Konfiguration, schlechteste Konfiguration und Anzahl der Konfigurationen, welche besser sind als die ermittelte Konfiguration, siehe 1. Da in jedem

Szenario der Wegfall der einzelnen Container untersucht wird, ergeben sich pro Szenario 14 Simulationsdurchläufe, was für alle 30 Szenarien dann 420 Simulationsdurchläufe ergibt.

Die folgende Tabelle fasst die Messergebnisse zusammen:

Anzahl der Durchläufe wo beide Heuristiken die optimale Lösung ermitteln	298 von 420 Durchläufen
Anzahl der Durchläufe wo die Rekonfiguration ohne Kostenbetrachtung die schlechteste Lösung ermittelt	21 von 420 Durchläufen
Anzahl der Durchläufe wo die Greedy Heuristik die schlechteste Lösung ermittelt	0 von 420 Durchläufen
Anzahl der Durchläufe wo beide Heuristiken eine identische, suboptimale Lösung ermitteln	36 von 420 Durchläufen
Anzahl der Durchläufe wo die Greedy Heuristik eine bessere Lösung ermittelt	81 von 420 Durchläufen
Anzahl der Durchläufe wo die Rekonfiguration ohne Kostenbetrachtung eine bessere Lösung ermittelt	5 von 420 Durchläufen

Tabelle 1: Messergebnisse 1. Szenario

Aus den Messergebnissen lassen sich folgende Schlussfolgerungen ziehen:

- Die Rekonfiguration findet in diesen Szenarien in über 70 Prozent der Durchläufe eine optimale Konfiguration, d.h. eine Rekonfiguration der Anwendung ist gegenüber einer erneuten Konfiguration einer Anwendung vorzuziehen.
- Der Wegfall von Komponenten, welche eine Instanziierung weiterer Komponenten benötigt, verursacht mehr Kosten für die Rekonfiguration als der Wegfall von Komponenten, welche direkt wieder ersetzt werden können.
- Die Rekonfiguration ohne Betrachtung von Kosteninformationen hat einen Nachteil in der Rekonfiguration von Komponenten mit einer hohen Ordnung.
- Die Greedy Heuristik passt die Ordnung dynamisch an die Kosteninformationen an und kann in den meisten Fällen eine Konfiguration ermitteln, welche gleich oder näher am Optimum ist.

- Die Greedy Heurstik ermittelt nicht immer die optimale Lösung, siehe Kapitel 6.1

Die detaillierten Messergebnisse der 30 Simulationen sind in Kapitel 6.4.1 aufgelistet.

6.3.3 2. Szenario

In diesem Szenario werden die zwei entworfenen Heuristiken zur Rekonfiguration miteinander verglichen. Es wird untersucht, wie oft die Greedy Heuristik eine Konfiguration findet, welche näher am Optimum liegt als die Rekonfiguration ohne Betrachtung der Kosteninformationen. Für dieses Szenario wird eine Anwendung mit den Parametern Tiefe 3, Breite 3 generiert und die Komponenten über 9 Container hinweg verteilt. Von jeder Komponente wird ein Duplikat erstellt und zufällig über die Container hinweg verteilt. Die internen Kosten der einzelnen Komponenten werden für jede Komponente auf eine Zufallszahl zwischen 1 und 50 gesetzt. Die Manipulierung der Umgebung wird in diesem Szenario auf zwei unterschiedliche Arten vorgenommen und wählt zwischen folgenden Manipulationsarten pro Simulationslauf zufällig aus:

- **Entfernen eines Containers aus der Umgebung:**
  Ein zufällig ausgewählter Container wird aus der Umgebung entfernt und die instanziierten Komponenten gelöscht. Danach wird der Container als neuer Container in die Containerliste des Simulators angehängt. Diese Manipulation ist identisch mit der Manipulation aus dem ersten Szenario.

- **Manipulation der Komponenten auf einem Container:**
  Ein Container (Quell-Container) wird zufällig ausgewählt und die Komponenten, welche dieser Container zur Verfügung stellt, werden auf einen anderen zufällig ausgewählten Container (Ziel-Container) verschoben. Der Quell-Container wird danach aus der Umgebung entfernt und es wird ein neuer Container in der Umgebung angelegt mit dem ursprünglichen Komponentenangebot des Ziel-Containers.

Für die Messungen wurden 500 Simulationsläufe mit den oben beschriebenen Parametern durchgeführt. Die Ergebnisse der Simulationsläufe sind in Abbildung 27 abgebildet.

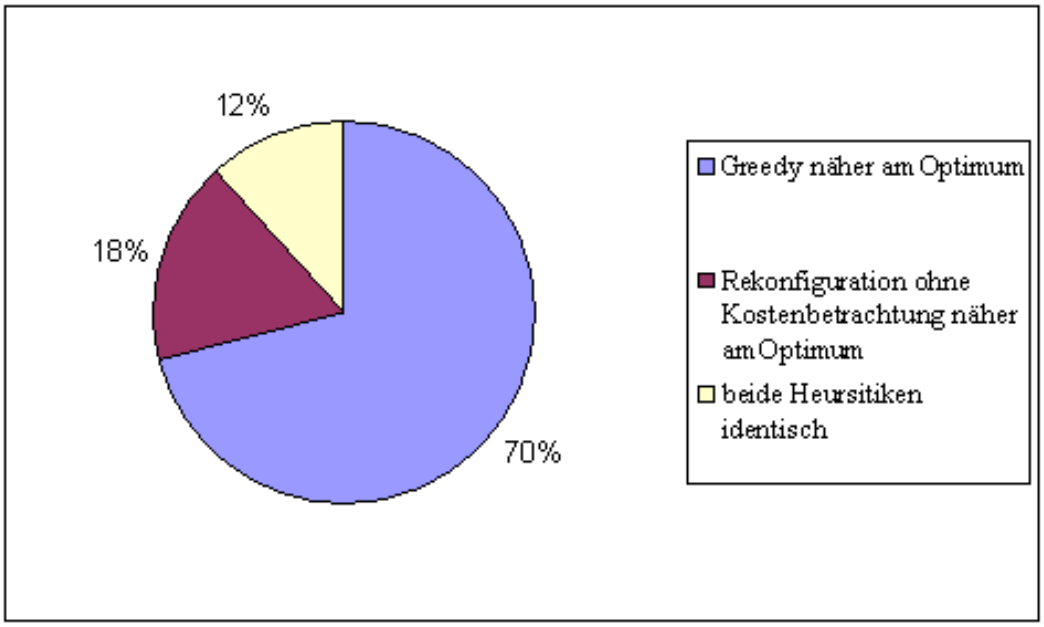


Abbildung 27: Messergebnisse 2. Szenario

Aus den Messergebnissen lassen sich folgende Schlussfolgerungen ziehen:

- Die Greedy Heuristik findet in 70 Prozent der Simulationsdurchläufe eine Konfiguration, welche näher am Optimum ist als die Rekonfiguration ohne Beachtung von Kosteninformationen.
- Im Gegensatz zum ersten Szenario schneidet die Rekonfiguration ohne Beachtung von Kosteninformationen im Vergleich zur Greedy Heuristik bedeutend schlechter ab. Daraus lässt sich folgern, dass eine steigende Anwendungskomplexität und eine höhere Anzahl von möglichen Lösungen für die Rekonfiguration ohne Beachtung von Kosteninformationen von Nachteil ist, da die Konfliktauflösungsstrategie sich nicht an die Kostensituation innerhalb der Anwendungen anpasst.

6.4 Messergebnisse

6.4.1 Messergebnisse 1. Szenario

Die folgenden Tabellen dokumentieren die Messergebnisse des Simulators. Es wurde gemessen wie sich der Ausfall eines Containers auf die Rekonfiguration auswirkt. In der ersten Spalte der Tabellen sind die einzelnen Container aufgelistet, welche in der Simulation entfernt wurden. Für die beiden Heuristiken wurde untersucht, wie viele bessere Konfigurationen existieren, oder ob die ermittelte Konfiguration das Optimum oder die schlechteste Konfiguration widerspiegelt.

Szenario 1-1, Seed: 11188399660011, 32 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	Optimum	4 K. besser	00	Optimum	8 K. besser	00	Optimum	4 K. besser
01	schl. K.	6 K. besser	01	schl. K.	6 K. besser	01	schl. K.	8 K. besser
000	7 K. besser	Optimum	000	7 K. besser	Optimum	000	9 K. besser	Optimum
001	1 K. besser	1 K. besser	001	2 K. besser	2 K. besser	001	1 K. besser	1 K. besser
010	Optimum	Optimum	010	Optimum	Optimum	010	Optimum	Optimum
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	1 K. besser	Optimum	0011	1 K. besser	1 K. besser	0011	1 K. besser	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 2: Messergebnisse Szenario 1-1

Szenario 1-2, Seed: 1118839966051, 8 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	Optimum	Optimum	00	Optimum	Optimum	00	Optimum	Optimum
01	schl. K.	Optimum	01	schl. K.	Optimum	01	schl. K.	Optimum
000	Optimum	Optimum	000	Optimum	Optimum	000	Optimum	Optimum
001	Optimum	Optimum	001	Optimum	Optimum	001	Optimum	Optimum
010	Optimum	Optimum	010	Optimum	Optimum	010	Optimum	Optimum
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	1 K. besser	Optimum	0000	1 K. besser	1 K. besser	0000	2 K. besser	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	1 K. besser
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 3: Messergebnisse Szenario 1-2

Szenario 1-3, Seed: 1118840215009, 8 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	2 K. besser	Optimum	00	2 K. besser	Optimum	00	2 K. besser	Optimum
01	schl. K.	Optimum	01	schl. K.	Optimum	01	schl. K.	Optimum
000	3 K. besser	Optimum	000	3 K. besser	Optimum	000	3 K. besser	Optimum
001	1 K. besser	1 K. besser	001	1 K. besser	1 K. besser	001	1 K. besser	Optimum
010	1 K. besser	1 K. besser	010	1 K. besser	Optimum	010	1 K. besser	1 K. besser
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 4: Messergebnisse Szenario 1-3

Szenario 1-4, Seed: 1118840328962, 32 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	Optimum	Optimum	00	Optimum	Optimum	00	Optimum	Optimum
01	schl. K.	8 K. besser	01	schl. K.	8 K. besser	01	schl. K.	8 K. besser
000	Optimum	Optimum	000	Optimum	Optimum	000	Optimum	Optimum
001	4 K. besser	Optimum	001	4 K. besser	Optimum	001	4 K. besser	Optimum
010	1 K. besser	1 K. besser	010	4 K. besser	Optimum	010	1 K. besser	1 K. besser
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	1 K. besser	Optimum	0000	2 K. besser	2 K. besser	0000	2 K. besser	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 5: Messergebnisse Szenario 1-4

Szenario 1-5, Seed: 1118840438149, 64 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	6 K. besser	6 K. besser	00	8 K. besser	8 K. besser	00	8 K. besser	8 K. besser
01	4 K. besser	4 K. besser	01	6 K. besser	6 K. besser	01	6 K. besser	6 K. besser
000	7 K. besser	Optimum	000	8 K. besser	Optimum	000	8 K. besser	Optimum
001	8 K. besser	Optimum	001	12 K. besser	Optimum	001	6 K. besser	Optimum
010	1 K. besser	Optimum	010	1 K. besser	Optimum	010	2 K. besser	Optimum
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	3 K. besser	Optimum	0001	4 K. besser	Optimum	0001	3 K. besser	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	1 K. besser	1 K. besser	0100	2 K. besser	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 6: Messergebnisse Szenario 1-5

Szenario 1-6, Seed: 1118847493585, 8 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	Optimum	Optimum	00	Optimum	Optimum	00	Optimum	Optimum
01	schl. K.	Optimum	01	schl. K.	Optimum	01	schl. K.	Optimum
000	Optimum	Optimum	000	Optimum	Optimum	000	Optimum	Optimum
001	Optimum	Optimum	001	Optimum	Optimum	001	Optimum	Optimum
010	Optimum	Optimum	010	Optimum	Optimum	010	Optimum	Optimum
011	2 K. besser	Optimum	011	2 K. besser	Optimum	011	2 K. besser	Optimum
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	1 K. besser	Optimum	0111	1 K. besser	Optimum	0111	1 K. besser	Optimum

Tabelle 7: Messergebnisse Szenario 1-6

Szenario 1-7, Seed: 1118841508108, 16 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	6 K. besser	2 K. besser	00	6 K. besser	2 K. besser	00	6 K. besser	2 K. besser
01	4 K. besser	4 K. besser	01	4 K. besser	4 K. besser	01	4 K. besser	4 K. besser
000	Optimum	Optimum	000	Optimum	Optimum	000	Optimum	Optimum
001	Optimum	Optimum	001	Optimum	Optimum	001	Optimum	Optimum
010	1 K. besser	1 K. besser	010	1 K. besser	Optimum	010	2 K. besser	2 K. besser
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	1 K. besser	Optimum	0011	1 K. besser	Optimum	0011	1 K. besser	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 8: Messergebnisse Szenario 1-7

Szenario 1-8, Seed: 1118841610956, 8 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	Optimum	2 K. besser	00	Optimum	Optimum	00	Optimum	2 K. besser
01	schl. K.	3 K. besser	01	schl. K.	3 K. besser	01	schl. K.	3 K. besser
000	2 K. besser	Optimum	000	1 K. besser	Optimum	000	2 K. besser	Optimum
001	1 K. besser	1 K. besser	001	1 K. besser	1 K. besser	001	1 K. besser	1 K. besser
010	Optimum	Optimum	010	Optimum	Optimum	010	Optimum	Optimum
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 9: Messergebnisse Szenario 1-8

Szenario 1-9, Seed: 1118841667307, 16 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	2 K. besser	2 K. besser	00	4 K. besser	4 K. besser	00	2 K. besser	2 K. besser
01	4 K. besser	4 K. besser	01	4 K. besser	4 K. besser	01	4 K. besser	4 K. besser
000	Optimum	Optimum	000	Optimum	Optimum	000	Optimum	Optimum
001	Optimum	Optimum	001	Optimum	Optimum	001	Optimum	Optimum
010	2 K. besser	Optimum	010	2 K. besser	Optimum	010	2 K. besser	Optimum
011	2 K. besser	2 K. besser	011	2 K. besser	2 K. besser	011	2 K. besser	2 K. besser
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 10: Messergebnisse Szenario 1-9

Szenario 1-10, Seed: 1118841745029, 8 mögliche Konfigurationen								
konstante Kostenverteilung			absteigende Kostenverteilung			aufsteigende Kostenverteilung		
Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy	Komp.	ohne Kosten	Greedy
00	1 K. besser	Optimum	00	2 K. besser	Optimum	00	1 K. besser	Optimum
01	schl. K.	2 K. besser	01	schl K.	2 K. besser	01	schl. K.	2 K. besser
000	Optimum	Optimum	000	Optimum	Optimum	000	Optimum	Optimum
001	2 K. besser	Optimum	001	2 K. besser	Optimum	001	2 K. besser	Optimum
010	Optimum	Optimum	010	Optimum	Optimum	010	Optimum	Optimum
011	Optimum	Optimum	011	Optimum	Optimum	011	Optimum	Optimum
0000	Optimum	Optimum	0000	Optimum	Optimum	0000	Optimum	Optimum
0001	Optimum	Optimum	0001	Optimum	Optimum	0001	Optimum	Optimum
0010	Optimum	Optimum	0010	Optimum	Optimum	0010	Optimum	Optimum
0011	Optimum	Optimum	0011	Optimum	Optimum	0011	Optimum	Optimum
0100	Optimum	Optimum	0100	Optimum	Optimum	0100	Optimum	Optimum
0101	Optimum	Optimum	0101	Optimum	Optimum	0101	Optimum	Optimum
0110	Optimum	Optimum	0110	Optimum	Optimum	0110	Optimum	Optimum
0111	Optimum	Optimum	0111	Optimum	Optimum	0111	Optimum	Optimum

Tabelle 11: Messergebnisse Szenario 1-10

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Ubiquitäre Umgebungen bieten für die Benutzer neue Möglichkeiten und verbesserte Unterstützung bei der täglichen Arbeit. In dem Komponentensystem PCOM lassen sich Anwendungen als Komposition von Komponenten zusammensetzen. Da sich die Art und Anzahl der verfügbaren Ressourcen in der Umgebung verändern kann, ist eine Anpassung der Anwendungen auf die veränderte Umgebung notwendig. Die Anpassung von Anwendungen verursacht Kosten. Im Rahmen dieser Diplomarbeit wurde ein Algorithmus zur Anpassung von Anwendungen in PCOM entwickelt, welcher die Kosten für die Anpassung möglichst gering hält.

Im Grundlagenteil wurden unter anderem die Themengebiete Rekonfiguration und verwandte Arbeiten näher betrachtet. Die Rekonfiguration von Anwendungen kann zu verschiedenen Zeitpunkten und an unterschiedlichen Orten erfolgen. Für den Algorithmus in dieser Arbeit wurde eine reaktive, strukturelle Rekonfiguration gewählt. Außerdem wurde untersucht, wie andere Systeme die Rekonfiguration unterstützen. Da der entwickelte Algorithmus eine Weiterentwicklung des Algorithmus zur initialen Konfiguration ist, wurden außerdem benötigte Grundlagen zu PCOM und dem Constraint Satisfaction Problem erläutert.

In Kapitel 3 wurden die Problemfelder der Diplomarbeit und die Lösungsmöglichkeiten analysiert. Die Anpassung von Anwendungen kann durch verschiedene Lösungsstrategien erfolgen. Für den in dieser Arbeit entwickelten Algorithmus wurde eine geeignete Heuristik entwickelt, da die Laufzeit für einen vollständigen Optimierungsalgorithmus im allgemeinen Fall zu hoch ist. Damit die Kosten für die Anpassung von Anwendungen in PCOM betrachtet werden können, wurden die verschiedenen Kostenfaktoren für die Anpassung von Anwendungen untersucht. Das entworfene Kostenmodell betrachtet die Sicherungskosten für die Anpassung von Anwendungen, da diese Kosten vor der Rekonfiguration berechnet werden können und einen großen Teil der Kosten für die Anpassung verursachen.

Für die Anpassung von Anwendungen wurden zwei Heuristiken implementiert. Die erste Heuristik rekonfiguriert Anwendungen ohne Beachtung von Kosteninformationen. Diese Heuristik wurde unter anderem implementiert um zu zeigen, dass die Beachtung von Kosteninformationen für die Rekonfiguration sinnvoll ist. Die Greedy Heuristik rekonfiguriert Anwendungen unter

Beachtung von Kosteninformationen. Für die Auflösung eines Anwendungskonflikt wird zuerst versucht die Komponente zu rekonfigurieren, welche die geringsten Kosten für die Anpassung verursacht. In der Evaluation wurde gezeigt, dass die Greedy Heuristik in den meisten Fällen eine Anpassung mit geringen Kosten vornimmt.

7.2 Ausblick

Für zukünftige Arbeiten in PCOM lassen sich zwei Forschungsbereiche herausgreifen:

- **Optimierung der Heuristik:**
  Im Rahmen dieser Diplomarbeit wurde die Rekonfiguration unter Beachtung von Kostenfaktoren untersucht. Die im Lösungsansatz entworfene Greedy Heuristik musste auf Grund der aktuellen Implementierung des PCOM-Simulator modifiziert werden. Die modifizierte Greedy Heuristik wählt aus den konfliktierenden Komponenten diejenige aus, welche den günstigsten Vater hat. Die gewählte Komponente ist nicht zwangsläufig die Komponente mit den günstigsten Kosten. Hier liegt noch Verbesserungspotential für die Greedy Heuristik.

  Für den Fall das die Heuristik eine Konfiguration ermittelt, welche nicht den Anforderungen genügt, wurde in Kapitel 4.2.3 ein Verbesserungsansatz diskutiert. Die ermittelte Konfiguration wird auf ihre verursachten Kosten bewertet und in einem zweiten Rekonfigurationsdurchlauf wird versucht die Konfiguration zu optimieren.

- **Optimierung des Kostenmodell:**
  Das im Rahmen dieser Diplomarbeit entworfene Kostenmodell modelliert die Sicherungskosten für den Wechsel vom Komponenten. Für eine detaillierte Modellierung von Kostenfaktoren ist unter anderem eine Betrachtung von Dienstgüte in PCOM nötig. Eine Optimierung des Kostenmodell ermöglicht eine genauere Suche nach einer Konfiguration mit möglichst geringen Kosten.

Literatur

[3PC] 3PC Projekthomepage http://www.3pc.info (Stand 01.03.2005)

[BDH+93] I. Barth, G. Dermler, T. Helbig, K. Rothermel, F. Sembach und T. Wahl: CINEMA: Eine konfigurierbare, integrierte Multimedia-Architektur. GI/ITG-Arbeitstreffen Verteilte Multimedia Systeme, Stuttgart, 1993.

[BHSR04] C. Becker, M. Handte, G. Schiele und K. Rothermel: PCOM - A Component System for Adaptive Pervasive Computing Applications. In Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 04), Orlando, USA.

[BSGR03] C. Becker, G. Schiele, H. Gubbels und K. Rothermel: BASE - A Micro-broker-based Middleware For Pervasive Computing. In Proceedings of the First IEEE International Conference on Pervasive Computing and Communication (PerCom), pp. 443-451, March 23-26, Fort Worth, USA, 2003.

[GDL+01] R. Grimm, J. Davis, E. Lemar et al.: Programming for Pervasive Computing Environments. UW-CSE 01-06-01, University of Washington, Department of Computer Science and Engineering, Seattle, WA, Juni 2001.

[GN05] X. Gu und K. Nahrstedt: Distributed Multimedia Service Composition with Statistical QoS Assurances. to appear in IEEE Transactions on Multimedia, 2005.

[GNY04] X. Gu, K. Nahrstedt und B. Yu: SpiderNet: An Integrated Peer-to-Peer Service Composition Framework. In Proceedings of IEEE International Symposium on High-Performance Distributed Computing (HPDC-13), Honolulu, Hawaii, 4-6 Juni 2004.

[GSS+02] D. Garlan, D. Siewiorek, A. Smailagic und P. Steenkiste: Project Aura: Toward Distraction-Free Pervasive Computing. In IEEE Pervasive Computing, pp.22-31, April-June 2002, http://computer.org/pervasive .

[HBR05] M. Handte, C. Becker und K. Rothermel: Peer-based Automatic Configuration of Pervasive Application. To appare in IEEE International Conference on Pervasive Services 2005 (ICPS05), Santorini, Greece.

[HR01] T. Härder und E. Rahm: Datenbanksysteme - Konzepte und Techniken der Implementierung. Springer Verlag, 2001.

[HY00] K. Hirayama und M. Yokoo: An Approach to Over-constrained Distributed Constraint Satisfaction Problems: Distributed Hierachical Constraint Satisfation. In Proceedings of International Conference on Multiagent Systems, 2000.

[KCM+00] F. Kon, R. Campbell, M. D. Mickunas, K. Nahrstedt und F. J. Ballesteros: 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. 9th IEEE International Symposium on High Performance Distributed Computing. Pittsburgh. 1-4 August, 2000.

[Keil03] R. Keil: Unterstützung zustandsbehafteter Dienste in Ubiquitous Computing. Studienarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2003.

[Omg02] Object Management Group: Corba Components Version 3.0, 2002, http://www.omg.org/technology/documents/formal/components.htm

[Rei04] M. Reinsch: Entwicklung und Evaluation eines Algorithmus zur initialen Konfiguration von Anwendungen in PCOM. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2004.

[RHC02] M. Roman,, H. Ho und R. Campbell: Application Mobility in Active Spaces. In 1st International Conference on Mobile and Ubiquitous Multimedia, Oulu, Finland, Dec 11-13, 2002.

[RHC+02] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell und K. Nahrstedt: A Middleware Infrastructure for Active Spaces. In IEEE Pervasive Computing, Oktober-Dezember 2002 (Vol. 1, No. 4), pp.74-83.

[Sat96] M. Satyanarayanan: Mobile Information Access. In IEEE Personal Communications, Februar 1996.

[SG02] J. P. Sousa und D. Garlan: Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In Software Architecture: System Design, Development and Maintenance (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture) Bosch, Gentleman, Hofmeister, Kuusela (Eds), Kluwer Academic Publishers, pp. 29-43, August 2002.

[SG03] J. P. Sousa und D. Garlan: The Aura Software Architecture: An Infrastructure for Ubiquitous Computing. Technical Report CMU-CS-03-183, Carnegie Mellon University School of Computer Science, August 2003.

[Sun05] Sun Microsystems: Enterprise JavaBeans Specification 3.0 Early Darft Review 2, Februar 2005, http://java.sun.com/products/ejb/docs.html

[Szy02] C. Szyperski: Component Software - Beyond Object-Oriented Programming (Second Edition). Addison-Wesley / ACM Press, 2002

[Weiser] M. Weiser: Ubiquitous Computing Homepage, http://www.ubiq.com/hypertext/weiser/UbiHome.html (Stand 27.01.05).

[Wei91] M. Weiser: The Computer for the Twenty-First Century, Scientific American, pp. 94-10, September 1991.

[Wei93] M. Weiser: Some Computer Science Issues in Ubiquitous Computing. In Communication of the ACM, Vol 36 Issue 7, pages 75-84, Juli 1993.

[WNGX01] D. Wichadakul, K. Nahrstedt, X. Gu und D. Xu: $2K^{Q+}$: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework*. In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, Nov. 2001.

[YDIK98] M. Yokoo, E. Durfee, T. Ishida und K. Kuwabara: The Distributed Constraint Satisfaction Problem: Formalization and Algorithms, IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No.5, pp.673-685, September-October 1998.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

(Andreas Störzbach)