Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften
Department Informatik

Diplomarbeit

Kontextabhängige und eigenverantwortliche Migration von Software-Agenten in heterogenen Umgebungen

Dirk Bade

---

dirk.bade@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 5101139

Erstgutachter: Professor Dr. W. Lamersdorf
Zweitgutachter: Dr. D. Moldt

Man muss die Welt nicht verstehen,
man muss sich nur darin zurechtfinden.
– *Albert Einstein*

Danksagungen

Diese Arbeit entstand am Arbeitsbereich *Verteilte Systeme und Informationssysteme* (VSIS) des Department Informatik an der Universität Hamburg. Zahlreiche Menschen, bei denen ich mich an dieser Stelle bedanken möchte, haben mich bei der Anfertigung fachlich und freundschaftlich unterstützt. An erster Stelle möchte ich mich daher ganz besonders bei Lars Braubach und Alexander Pokahr, die mit ihrem *Jadex*-Projekt den Rahmen für diese Arbeit gestellt haben, für die hervorragende fachliche Betreuung bedanken. Des Weiteren bedanke ich mich bei Prof. Dr. W. Lamersdorf und Herrn Dr. D. Moldt für die Möglichkeit zur Durchführung dieser Arbeit, die gegebenen Inspirationen sowie das produktive Arbeitsumfeld. Weiterer Dank gilt außerdem Christian Kunze, der mir mit seinen Kenntnissen im Bereich des *Mobile Computing* immer zu Rate stand, sowie meinen unermüdlichen Korrektoren Ursula Salow, Jürgen Bade und Ilse Bade für die orthographischen Hilfestellungen.

Neben der fachlichen Unterstützung möchte ich mich aber auch bei meinen Freunden und Kommilitonen bedanken, die mir in den letzten Monaten mit kleinen und großen Gesten immer zur Seite standen. Unter diesen gilt mein besonderer Dank Tanja Döring, Telse Först, Nicolas Haase, Sascha Jockel, Viola Korte, Frank Ploss, Anne Schick, Oke Schober, Christoph Weber, Stefan Weber und insbesondere Alexandra Vetter. Zuletzt möchte ich mich auch ganz herzlich bei meiner Familie für ihren gebotenen Rückhalt bedanken und hier insbesondere bei Frank Bade, der jederzeit für mich da war, bei meinem Vater Jürgen Bade, ohne dessen Unterstützung und Hilfe ich nicht so weit gekommen wäre und bei meiner Mutter Jutta Bade, die mich überall begleitet hat und die sich sicherlich sehr über diese Arbeit gefreut hätte.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 3
1.2 Zielsetzung . . . 4
1.3 Aufbau der Arbeit . . . 5

2 Grundlagen der Resource Awareness **7**
2.1 Begriffsdefinition und -abgrenzung . . . 8
2.1.1 Kontext . . . 8
2.1.2 Ressourcen . . . 9
2.1.3 Awareness . . . 9
2.2 Die Umwelt eines Agenten . . . 10
2.2.1 Was ist eine Umwelt? . . . 11
2.2.2 Umweltmodelle . . . 12
2.2.3 Entitäten in Umweltmodellen . . . 18
2.2.4 Ereignisse in Umweltmodellen . . . 19
2.3 Discovery in Infrastrukturnetzwerken . . . 21
2.3.1 Anforderungen und Merkmale . . . 21
2.3.2 Jini . . . 24
2.3.3 Service Location Protocol . . . 26
2.3.4 Universal Plug and Play . . . 27
2.3.5 Salutation . . . 29
2.3.6 JXTA . . . 30
2.3.7 Diskussion . . . 32
2.4 Discovery in mobilen ad-hoc Netzwerken . . . 37
2.4.1 Nutzung von Routing-Informationen . . . 37
2.4.2 Bluetooth SDP . . . 40
2.4.3 Konark . . . 41
2.4.4 DEAPspace . . . 43
2.4.5 Scalable Service Discovery for MANET . . . 44
2.4.6 CARD . . . 45
2.4.7 Diskussion . . . 46
2.5 Repräsentation von Ressourcen . . . 51
2.5.1 Resource Description Framework . . . 52
2.5.2 DAML+OIL . . . 55
2.5.3 OWL Web Ontology Language . . . 56
2.5.4 Web Services Description Language . . . 58
2.5.5 Zusammenfassung . . . 62
2.6 Zusammenfassung . . . 62

3 Grundlagen der Migration in mobilen Systemen **65**
3.1 Grundlagen der Mobilität . . . 65
3.1.1 Abstraktionen . . . 66
3.1.2 Mechanismen . . . 68
3.1.3 Technologien . . . 71
3.1.4 Entwurfsparadigmen . . . 74
3.1.5 Java als de-facto Standard für MCS . . . 77
3.1.6 Zusammenfassung . . . 82
3.2 Mobile Agenten . . . 82
3.2.1 Einführung . . . 82
3.2.2 Definitionen . . . 83
3.2.3 Charakteristiken . . . 84
3.2.4 Vor- und Nachteile . . . 85
3.2.5 Optimierung der Migrationseffizienz . . . 88
3.2.6 Mobile Agenten und MANETs . . . 91
3.2.7 Zusammenfassung . . . 93
3.3 Mobilitätsmodelle . . . 93
3.3.1 Einführung . . . 93
3.3.2 Das Kalong-Modell . . . 94
3.3.3 Mobilitätssprache . . . 101
3.3.4 Zusammenfassung . . . 106
3.4 Zusammenfassung . . . 107

4 Konzeption der kontextabhängigen und eigenverantwortlichen Migration **109**
4.1 Motivation . . . 111
4.2 Anwendungsbeispiele . . . 113
4.2.1 Beispiel 1: Verteilte Berechnungen . . . 113
4.2.2 Beispiel 2: Selbstorganisierende Dienstnetzwerke . . . 115
4.2.3 Beispiel 3: Ad-hoc Tauschbörsen . . . 116
4.2.4 Beispiel 4: Kaufhausagent . . . 117
4.2.5 Beispiel 5: Mobiler Nachrichtendienst . . . 119
4.3 Anforderungsanalyse . . . 120
4.3.1 Benutzer . . . 121
4.3.2 Agenten . . . 121
4.3.3 Ausführungsumgebung . . . 122
4.3.4 Sicherheit . . . 125
4.4 Spezifikation des Umweltmodells . . . 126
4.4.1 Entwicklung eines ressourcenbasierten Umweltmodells . . . 127
4.4.2 Identifikation von Umweltereignissen . . . 129
4.5 Resource Awareness-Komponente . . . 131
4.5.1 Organisation des Umweltmodells . . . 132
4.5.2 Erstkontaktmechanismen . . . 134
4.5.3 Verbreitung von Umweltinformationen . . . 136
4.5.4 Repräsentationssprachen . . . 137
4.5.5 Proprietäres Basisprotokoll . . . 137
4.5.6 Sparsamer Umgang mit Daten . . . 139

4.5.7 Konfigurierbarkeit . . . 139
4.6 Spezifikation des Mobilitätsmodells . . . 140
4.6.1 Erweiterung der Mobilitätssprache . . . 140
4.6.2 Beschreibung des Mobilitätsmodells . . . 142
4.7 Migrationskomponente . . . 145
4.7.1 Versenden von Agenten . . . 145
4.7.2 Empfangen von Agenten . . . 145
4.7.3 Sicherheitskopien . . . 146
4.7.4 Verschieben, Kopieren und Klonen . . . 146
4.7.5 Verwaltungsoperationen . . . 147
4.7.6 Entfernte Befehlsausführung . . . 147
4.7.7 Transfermechanismen . . . 147
4.7.8 Fehlerbehandlung . . . 148
4.7.9 Konfigurierbarkeit . . . 148
4.8 Verwandte Arbeiten . . . 149
4.8.1 ASCML . . . 149
4.8.2 ProNav . . . 150
4.9 Zusammenfassung . . . 153

5 Prototypische Umsetzung des Konzeptes 155
5.1 Ausführungsumgebung . . . 155
5.2 Resource Awareness-Komponente . . . 156
5.2.1 Anforderungen . . . 156
5.2.2 Realisierung als Agent . . . 158
5.2.3 Architektur . . . 159
5.2.4 Funktionsweise . . . 162
5.2.5 Zusammenfassung . . . 166
5.3 Migrationskomponente . . . 166
5.3.1 Anforderungen . . . 167
5.3.2 Architektur . . . 168
5.3.3 Funktionsweise . . . 169
5.3.4 Zusammenfassung . . . 177
5.4 Zusammenfassung . . . 177

6 Diskussion, Zusammenfassung und Ausblick 181
6.1 Diskussion . . . 181
6.1.1 Risiken . . . 181
6.1.2 Chancen . . . 182
6.2 Zusammenfassung . . . 182
6.3 Ausblick . . . 184

Literaturverzeichnis 189

Abbildungsverzeichnis

2.1 Interaktion eines Agenten mit seiner Umwelt (nach [Russell und Norvig 2003]) . . . 7
2.2 Soziale Umwelt: Koordination, Kooperation und Kompetition [Odell u. a. 2002] . . 15
2.3 Allgemeines Umweltmodell für MAS (nach [Weyns u. a. 2004, Weyns u. a. 2005]) . 16
2.4 Klassifizierung von Umweltereignissen (eigene Darstellung) . . . 20
2.5 Jini-Protokolle: discovery / join / lookup (nach [Cheng 2002]) . . . 24
2.6 SLP-Agenten: Registrieren und Suchen von Diensten (nach [Bettstetter und Renner 2000]) . . . 26
2.7 UPnP: Anbieten und Suchen von Diensten über Multi- und Unicast (nach [UPnP 2003]) . . . 28
2.8 Salutation: Übersicht über die Salutation-Architektur (nach [Wiklander 2001]) . . . 30
2.9 JXTA: Virtuelles JXTA-Netzwerk (nach [Traversat u. a. 2003]) . . . 31
2.10 Bluetooth: Topologie eines Bluetooth ScatterNets (nach [Alterovitz 2001]) . . . 40
2.11 Konark: Der Konark Discovery Stack (nach [Helal u. a. 2003]) . . . 41
2.12 Konark: Baumstruktur des Diensteverzeichnis (nach [Helal u. a. 2003]) . . . 42
2.13 CARD: Netzwerksicht eines mit Kontakten verbundenen Knotens (nach [Helmy u. a. 2005]) . . . 45
2.14 CARD: Kontaktauswahl-Algorithmus (nach [Nahata u. a. 2002, Poster]) . . . 46
2.15 RDF-Graph gehörend zum Beispiel aus Listing 2.1 (eigene Darstellung) . . . 52

3.1 Architektur herkömmlicher verteilter Systeme (oben) versus Mobile Code Systems (unten) (nach [Fuggetta u. a. 1998]) . . . 66
3.2 Interne Struktur zweier Ausführungseinheiten (nach [Fuggetta u. a. 1998]) . . . 67
3.3 Klassifikation von Mobilitätsmechanismen (nach [Fuggetta u. a. 1998]) . . . 68
3.4 Klassifikation verschiedener Migrationsaspekte (nach [Illmann u. a. 2000]) . . . 69
3.5 Verschiedene Entwurfsparadigmen von Mobile Code Systems (nach [Carzaniga u. a. 1997]) . . . 75
3.6 Architektur der Java-Umgebung (nach [Bouchenak u. a. 2004]) . . . 79
3.7 Java Thread State (nach [Bouchenak u. a. 2004]) . . . 81
3.8 Client/Server versus mobile Agenten (nach [Pham und Karmouch 1998]) . . . 83
3.9 Klassifikation von Kernpunkten der Leistungsoptimierung bei mobilen Agenten (nach [Braun und Rossak 2005]) . . . 89
3.10 Die sechs Phasen des Migrationsprozesses und die drei Sichten des Mobilitätsmodells (nach [Braun und Rossak 2005]) . . . 94
3.11 Kalong: Beispiel für eine Migrationsstrategie (nach [Kern und Braun 2005]) . . . 100

4.1 Anwendungsbeispiel: Verteilte Berechnung der Mandelbrotmenge. Die angegebenen Zahlenwerte repräsentieren die zu berechnenden Bildzeilen (eigene Darstellung) . . . 114
4.2 Anwendungsbeispiel: Mitnahme eines persönlichen Nachrichtenagenten (eigene Darstellung) . . . 119
4.3 Erweitertes allgemeines Umweltmodell für MAS (eigene Darstellung) . . . 127

4.4 Standard-Nachrichtenkanäle der RA-Komponente (eigene Darstellung) 133
4.5 ProNav Infrastrukturkomponenten (nach [Erfurth und Rossak 2002]) 151

5.1 RF: Zusammenspiel der Komponenten (eigene Darstellung) 160
5.2 RF: Abonnieren eines Nachrichtenkanals (eigene Darstellung) 163
5.3 RF: Kommunikation von Ereignissen (eigene Darstellung) 164
5.4 RF: Auffinden von Kontakten (eigene Darstellung) 165
5.5 MM: Initiierung einer Migration (eigene Darstellung) 170
5.6 MM: Aufforderung zur entfernten Migration (eigene Darstellung) 171
5.7 MM: Schematische Darstellung der Continuation-Ausführungskomponente (eigene Darstellung) 173

1 Einleitung

Das Paradigma der agentenorientierten Programmierung gewinnt seit Beginn der achtziger Jahre zunehmend an Bedeutung. Aufgrund der höheren Abstraktionen, die dieses Paradigma z.B. im Vergleich zu der objektorientierten Programmierung bietet, eignet es sich insbesondere für die Erstellung komplexer und verteilter Systeme [Luck u. a. 2005]. Der Grund hierfür findet sich in den konstituierenden Eigenschaften der Software-Agenten, wie der Fähigkeit die Umwelt wahrzunehmen und autonom reaktiv oder proaktiv in dieser zu agieren und sich durch den Austausch von Nachrichten mit anderen Agenten zu koordinieren [Wooldridge und Jennings 1995, Braun und Rossak 2005].

Die zusätzliche Eigenschaft der Mobilität ermöglicht es Software-Agenten darüber hinaus selbständig ihren Ausführungsort in einem verteilten System zu wechseln und so zwischen einzelnen Knoten in einem Netzwerk zu migrieren. In der Literatur wird diesem Aspekt insbesondere der Vorteil höherer Verarbeitungseffizienz zugesprochen, denn auf diese Weise ist es möglich, den Programmcode zu den Daten zu transferieren, anstatt eine unter Umständen große Menge an Daten für die Verarbeitung zu dem Programmcode übertragen zu müssen [Braun und Rossak 2005, Shiao 2004]. Hinzu kommt, dass manche Ressourcen, zum Beispiel ein Prozessor, ein Dateisystem oder ein bestimmter Dienst, unter Umständen nur lokal in einer Ausführungsumgebung zugreifbar sind und nicht über ein Netzwerk in Anspruch genommen werden können. Auch solche Ressourcen lassen sich durch Verwendung mobiler Agenten nutzbar machen.

Von besonderem Interesse ist das Paradigma mobiler Agenten daher im Zuge der aufkommenden allgegenwärtigen Informationsverarbeitung (engl. *ubiquitous computing* oder *pervasive computing*). Denn mit zunehmender Dichte informationsverarbeitender Komponenten sowie deren Vernetzung wächst auch die Anzahl potenziell nutzbarer Ressourcen. Noch einen Schritt weiter geht das technologische Paradigma der Umgebungsintelligenz (engl. *ambient intelligence*), welches darauf abzielt, Komponenten in der Umgebung miteinander interagieren zu lassen, um Ziele und Aktivitäten von Benutzern im Alltag zu unterstützen. Die Umgebung ist also der Präsenz des Benutzers gewahr, kann diesem personalisierte Dienste anbieten und vermag sich dem Verhalten des Benutzers anzupassen [Pirker u. a. 2004]. Es besteht Konsens darüber, dass hierfür unter anderem Autonomie, Verteilung und Adaptivität die Schlüsselmerkmale dieser Komponenten sein müssen und sie diesbezüglich dieselben Eigenschaften mit Agenten teilen [Luck u. a. 2005].

Auch die Arten der Vernetzung ändern sich in diesen Paradigmen, weg von einer starren Struktur und hin zu dynamischen und offenen Systemen, in welchen sich Geräte spontan miteinander verbinden, Daten austauschen und im nächsten Augenblick wieder voneinander trennen. Derartige Netze, auch als *ad-hoc Netzwerke* bezeichnet, finden sich vor allem im Bereich der mobilen Informationsverarbeitung (engl. *mobile computing* oder auch *nomadic computing*), welche mit zunehmender Verbreitung leistungsstarker mobiler Geräte ebenfalls an Bedeutung gewinnt [Roth 2005]. Die besonderen Charakteristiken dieser Art von Netzwerken liegen zum einen in der bereits erwähnten hohen Dynamik hinsichtlich der Teilnehmer und der Topologie des Netzwerkes, zum anderen aber auch in der Qualität der Verbindungen, die häufig instabil und durch geringe Datenraten sowie hohe Nachrichtenlaufzeiten geprägt sind.

Derartige Umgebungen stellen besondere Anforderungen, einerseits an das Auffinden von Ressourcen und den Austausch von Informationen über diese Angebote, andererseits an die Kom-

munikation mit diesen Ressourcen und die Art des Zugriffs auf diese. Sogenannte *Discovery*-Verfahren übernehmen in diesem Kontext die Aufgabe, Ressourcenanbieter und -suchende zusammenzuführen, indem sie den Austausch von Ressourceninformationen zwischen einzelnen Teilnehmern regeln. Um jedoch der Heterogenität und Dynamik dieser Umgebungen gerecht zu werden, reicht der Einsatz eines einzigen Verfahrens häufig nicht aus. Unterschiedliche Verfahren müssen parallel unterstützt und je nach Beschaffenheit der Umgebung dynamisch für den Informationsaustausch eingesetzt werden können. Dies erfordert eine generische Repräsentation der Umwelt, in die verfahrenseigene Repräsentationsformen bidirektional überführt werden können.

Die unter Umständen kurze Zeitspanne des Bestehens von ad-hoc Netzen sowie deren Verbindungscharakteristiken erfordern außerdem effiziente und vor allem verlässliche Migrationsmechanismen für die Übertragung eines Agenten. Es müssen zudem Vorkehrungen getroffen werden, um den Verlust eines Agenten oder einer Kommunikationsverbindung zu diesem zu vermeiden. Hierzu gehören zum Beispiel das Hinterlegen von Sicherheitskopien, Weiterleiten von Nachrichten an migrierte Agenten sowie Transaktionskontrollen, die alle Phasen eines Migrationsvorgangs abdecken. Die hierfür notwendigen Mechanismen unterstützen darüber hinaus eine einfache verteilte Bearbeitung einer Aufgabe durch im Netzwerk verteilte Duplikate eines Agenten. Dies kann unter Umständen die Effektivität der Bearbeitung erhöhen und somit die notwendige Bearbeitungszeit verringern, was ebenfalls der Kurzlebigkeit eines ad-hoc Netzwerkes zu Gute kommt.

Alle aufgeführten Mechanismen sollten darüber hinaus an die aktuellen Gegebenheiten der Umwelt angepasst werden können, um der Dynamik der adressierten Umwelt gerecht zu werden. Einerseits soll der Benutzer etwaige Vorgaben machen dürfen, andererseits soll jedoch auch die Ausführungsumgebung eigenmächtig die Mechanismen dynamisch an die Umweltbedingungen adaptieren. Hierzu gehören unter anderem die adaptive Aktualisierung des Umweltmodells sowie die Erstellung effizienter Migrationsstrategien zur Verminderung des zu übertragenden Datenvolumens. Auf diese Weise lässt sich letztendlich eine bessere Ausnutzung der in der Umwelt des Benutzers verfügbaren Ressourcen erreichen. Damit sich ein Benutzer jedoch auf die eigentliche Funktion einer Anwendung konzentrieren kann, müssen die Mechanismen der kontextabhängigen Migration transparent und ohne Eingriff des Benutzers ablaufen. Dies erfordert insgesamt eine höhere Autonomie und Verantwortung der Agenten gegenüber ihrem Benutzer. Ein Agent muss etwaige Migrationsentscheidungen eigenmächtig in Abhängigkeit von seiner aktuellen Situation und den Gegebenheiten der Umwelt treffen können und dabei mögliche Konsequenzen einer Migration berücksichtigen. Hierfür muss der Agent unter Umständen eigenverantwortlich Maßnahmen seitens der Ausführungsumgebung anfordern, die den Zugriff des Benutzers auf den Agenten und dessen Daten ermöglichen und sicherstellen.

Das Ziel dieser Arbeit ist daher die Entwicklung von Modellen und Methoden, welche Software-Agenten die kontextabhängige und eigenverantwortliche Migration in heterogenen Umgebungen ermöglichen. Hierfür sollen zwei Komponenten konzipiert und umgesetzt werden, welche als Teil einer Middleware die Wahrnehmung der Umwelt sowie, in Abhängigkeit von dieser, eine verlässliche und effiziente Migration erlauben. Im Zuge dessen soll ein generisches Umweltmodell entworfen werden, in welches heterogene Umweltinformationen aus unterschiedlichen Quellen integriert werden können. Dieses Umweltmodell stellt die Basis für ein Mobilitätsmodell dar, in welchem Anforderungen an die Mechanismen der Migration in Abhängigkeit von den Umweltbedingungen formuliert werden können, um Agenten eine verlässliche, effiziente und schnelle Bearbeitung ihrer Aufgaben in dynamischen, verteilten und offenen Systemen zu ermöglichen.

1.1 Motivation

Motiviert wurde das in dieser Arbeit vorgestellte Konzept der kontextabhängigen und eigenverantwortlichen Migration durch den Wunsch, dass ein Benutzer seine Anwendungen ständig mit sich führen und überall auf diese zugreifen können soll. Die Anwendungen wiederum sollen hierdurch nach Möglichkeit in ihrer Funktion nicht beeinträchtigt werden, sondern jede sich bietende Möglichkeit zur Fortführung ihrer Aufgabe nutzen, auch wenn sie auf unterschiedlichen, möglicherweise mobilen Geräten des Benutzers ausgeführt und mit fortwährend neuen Umgebungen konfrontiert werden.

Im Mittelpunkt dieses Konzeptes steht jedoch der Benutzer selbst, welcher letztlich von den erweiterten Möglichkeiten seiner Anwendungen profitieren soll. Dabei geht es nicht um die Funktionen einer Anwendung selbst, sondern um die grundlegenden Fähigkeiten einer Anwendung ihren Benutzer begleiten zu können. So hat der Benutzer jederzeit und allerorten nicht nur Zugriff auf seine persönlichen Daten, sondern auch auf seine bereits an ihn angepassten Anwendungen. Anstatt auf jedem Gerät, mit welchem der Benutzer arbeitet, manuell bestimmte Anwendungen installieren, konfigurieren und aktualisieren zu müssen, reduziert sich diese Aufgabe auf eine einzige Anwendungsinstanz. Diese stellt dem Benutzer immer dieselbe Benutzungsschnittstelle zur Verfügung, sodass sich dieser nicht fortlaufend an neue Arbeitsumgebungen anpassen muss[1]. Vorstellbar wäre darüber hinaus auch, dass eine Anwendung durch ständigen Kontakt zu dem Benutzer zusätzlich von dessen Aktionen lernt und sich so im Laufe der Zeit optimal an dessen Gewohnheiten anpassen kann.

Anhand eines Beispiels soll die Motivation im Folgenden näher verdeutlicht werden. Eine Mail-Anwendung verwaltet für ihren Benutzer nicht nur alle Nachrichten und Kontaktdaten, sondern bietet darüber hinaus auch einen lernfähigen Werbefilter, der sich an die Interessen des Benutzer anpasst. Unabhängig davon, ob der Benutzer vor seinem Heim-Computer sitzt, mit seinem mobilen Gerät unterwegs ist oder im Büro arbeitet, verwendet er immer dieselbe Instanz dieser Mail-Anwendung. Sobald der Benutzer einen Ort verlässt, folgt ihm die Anwendung samt ihrer Daten autonom auf dem mobilen Gerät und bietet unterwegs Zugriff auf alle Nachrichten und Kontakte. Außerdem kann die Anwendung automatisch neue Nachrichten abholen, sobald auf dem Weg eine entsprechende Internet-Ressource von Dritten angeboten wird. Erreicht der Benutzer schließlich wieder sein Zuhause oder Büro, migriert die Anwendung auf den entsprechenden Computer und steht für weitere Aufgaben zur Verfügung.

Haben Anwendungen nicht die Möglichkeit zu migrieren und ihrem Benutzer zu folgen, stellt sich obiges Szenario wie folgt dar: die Mail-Anwendung muss von dem Benutzer auf jedem Gerät manuell installiert, konfiguriert und wenn erforderlich aktualisiert werden. Unter Umständen benötigt der Benutzer auch für jede Installation eine gültige Anwendungslizenz. Empfängt er dann beispielsweise an seinem Heim-Computer eine neue Nachricht und verlässt anschließend mit seinem mobilen Gerät das Haus, so muss er gegebenenfalls den Datenbestand der beiden Anwendungsinstanzen manuell synchronisieren um auch unterwegs Zugriff auf diese Nachricht zu haben. Möchte er auf dem Weg neue Nachrichten abrufen, so muss er eigens eine unter Umständen

[1] Ein ähnliches Ziel verfolgen auch sogenannte Anwendungsdienstleister (engl. *application service provider*) im Internet. Diese bieten ihren Kunden Anwendungen oder Anwendungsumgebungen an, mit welchen der Kunde über eine Netzwerkverbindung interagieren kann. Die Anwendungen laufen hierbei auf den Geräten des Diensterbringers, während das Gerät des Kunden lediglich der Darstellung dient [SpiegelOnline 2007, GoPC 2007, Nivio 2007]. Auch bei einem solchen Dienst werden die Anwendungen an einen Benutzer und nicht an dessen Geräte gebunden. Der Zugriff auf Anwendungen und Daten erfordert jedoch eine permanente und schnelle Internetverbindung, sodass sich dieses Konzept nicht für nomadische Benutzer und den Einsatz unterwegs eignet. Außerdem ist der Benutzer nicht Eigentümer einer Anwendung, sondern mietet diese lediglich für einen bestimmten Zeitraum.

teure Verbindung ins Internet aufbauen. Da die Daten des zu Hause sorgfältig angelernten Werbefilters bei der Synchronisation nicht berücksichtigt werden konnten, lädt die Mail-Anwendung auch alle unerwünschten Werbenachrichen aus dem Internet auf das mobile Gerät. Erreicht der Benutzer schließlich wieder sein zu Hause oder sein Büro, so muss er die Datenbestände des stationären und des mobilen Gerätes wiederholt miteinander synchronisieren um deren Konsistenz zu wahren.

Wie dieses Anwendungsbeispiel zeigt, kann es durchaus sinnvoll sein, Anwendungsinstanzen an den Benutzer, anstatt an dessen Geräte, zu binden. Der Benutzer wird hierdurch in seinem Alltag nicht nur von etwaigen Installations- und Synchronisationsaufgaben befreit, er profitiert darüber hinaus von der ständigen Verfügbarkeit seiner eigenen Anwendungen und kann auf seine persönlichen Daten jederzeit und überall zugreifen. Herkömmliche Anwendungen können einen derartigen Komfort nicht bieten, sondern verlangen multiple Installationen sowie die manuelle Synchronisation der Daten zwischen diesen.

1.2 Zielsetzung

Das Ziel dieser Arbeit besteht in der Konzeption und Umsetzung zweier Komponenten, mittels derer die kontextabhängige und eigenverantwortliche Migration von Agenten realisiert werden kann. Die erste Komponente, welche für die Wahrnehmung der Umwelt zuständig ist, soll Agenten und ihrer Ausführungsumgebung Informationen über verfügbare Ressourcen in Form eines generischen Umweltmodells zur Verfügung stellen. Anhand dieser Informationen können Agenten schließlich Migrationsentscheidungen treffen. Der Wunsch zu migrieren soll dann von der zweiten Komponente bearbeitet werden, welche den Vorgang der Migration unter Berücksichtigung der Umweltgegebenheiten einerseits möglichst verlässlich und andererseits möglichst effizient durchführen soll.

Der Umwelt eines Agenten sollen hierbei möglichst wenige Einschränkungen auferlegt werden. Ziel ist es, die Heterogenität der Software-, Hardware- und Kommunikationsinfrastruktur in dem Konzept zu berücksichtigen. Das bedeutet, dass auf unterschiedliche Geräte und Geräteklassen (bspw. stationäre und mobile Geräte), sowie deren spezifische Beschränkungen hinsichtlich verfügbarer Ressourcen, bei dem Entwurf der Komponenten eingegangen werden soll. Des Weiteren gilt es, bestehende Software-Infrastrukturen, zum Beispiel Verfahren zum Austausch von Informationen, in das Konzept mit einzubeziehen und einerseits für die Interoperabilität mit diesen zu sorgen, andererseits aber auch offen für zukünftige Entwicklungen zu bleiben. Außerdem soll das Konzept sowohl Infrastruktur- als auch mobile ad-hoc Netzwerke berücksichtigen und bei der Kommunikation in derartigen Netzen die jeweils spezifischen Charakteristiken eines Netzwerks (Dynamik, Netzwerkparameter etc.) beachten.

Um zu zeigen, dass das Konzept in die Praxis umsetzbar ist und die erarbeiteten Anforderungen hinreichend sind, sollen schließlich Prototypen der Komponenten für eine bereits existierende Agentenplattform entwickelt werden. Diese Prototypen sollen grundlegende Funktionen bereitstellen, auf denen aufbauend Agenten kontextabhängig und eigenverantwortlich in heterogenen und dynamischen Umgebungen migrieren können, um ihre Aufgaben schnell und zuverlässig zu bearbeiten.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in sechs Kapitel. In diesem ersten Kapitel wurde bereits eine thematische Einführung sowie die Motivation zur Bearbeitung dieses Themas und die Zielsetzung gegeben. Die nächsten beiden Kapitel 2 und 3 beschäftigen sich ausführlich mit den zum Verständnis notwendigen Grundlagen. Eine wesentliche Grundlage stellt die *Resource Awareness* dar, welche die Verfahren und Modelle beschreibt, mit Hilfe derer die Wahrnehmung der Umwelt realisiert werden kann. Ausgehend von der Begriffsklärung und Abgrenzung gegenüber anderen Begriffen werden in Kapitel 2 verschiedene Umweltmodelle vorgestellt, unter anderem auch das allgemeine Umweltmodell für Multiagentensysteme, welches fortan als Basis für weitere Betrachtungen dient. Anschließend werden eine Reihe sogenannter Discovery-Verfahren, welche dem Austausch von Informationen in einem Netzwerk dienen, hinsichtlich ihrer Eignung für heterogene Umgebungen anhand eines Kriterienkatalogs miteinander verglichen. Den Abschluss des Kapitels bildet die Untersuchung verschiedener Standards von Repräsentationssprachen, welche einen syntaktischen und semantischen Rahmen zur Beschreibung von Ressourcen bieten.

Der zweite wichtige Grundlagenbereich wird schließlich in Kapitel 3 behandelt. Dieses Kapitel beschäftigt sich allgemein mit der Migration von Ausführungseinheiten in mobilen Systemen. Nach einem Überblick über die Grundlagen der Mobilität, welcher die wesentlichen Abstraktionen, Mechanismen, Technologien und Paradigmen umfasst, wird das Paradigma mobiler Agenten weitergehend vertieft. Hierbei werden neben gängigen Definitionen vor allem die Charakteristiken mobiler Agenten beschrieben sowie deren Vor- und Nachteile diskutiert. Des Weiteren werden Mobilitätsmodelle und insbesondere das *Kalong*-Modell vorgestellt, sowie mit der *Mobility Language* eine Beschreibungssprache für Mobilitätsmodelle eingeführt [Braun und Rossak 2005].

Aufbauend auf diesen Grundlagenkapiteln wird in Kapitel 4 das Konzept der kontextabhängigen, eigenverantwortlichen Migration erarbeitet. Aus einer Reihe verschiedener Anwendungsbeispiele wird eine Anforderungsanalyse durchgeführt, die als Grundlage der Beschreibung zweier Middleware-Komponenten dient. Die Aufgabe der Resource Awareness-Komponente ist die Wahrnehmung der Ressourcenumwelt einer Ausführungsumgebung. Die Migrationskomponente ist für alle Belange der Migration, inklusive etwaiger Sicherungsmaßnahmen, verantwortlich. Auf dem Zusammenspiel dieser beiden Komponenten beruht schließlich das Konzept der kontextabhängigen und eigenverantwortlichen Migration.

Im nachfolgenden Kapitel 5 wird, ausgehend von der Beschreibung der beiden Middleware-Komponenten, die Umsetzung des Konzeptes für die *Jadex*-Agentenplattform dargelegt. Im Rahmen dessen werden zwei unabhängige Komponenten vorgestellt, welche grundlegende Funktionen für die Wahrnehmung der Umwelt und für die Migration implementieren und die gemeinsam die Möglichkeit zur kontextabhängigen und eigenverantwortlichen Migration bieten.

Den Abschluss der Arbeit bildet das Kapitel 6, in welchem die Chancen und Risiken des Konzeptes diskutiert werden und eine Zusammenfassung der Ergebnisse dieser Arbeit gegeben wird. Am Ende werden darüber hinaus in einem Ausblick die konzeptionellen, technischen und anwendungsbezogenen Visionen der weiteren Entwicklung vorgestellt.

2 Grundlagen der Resource Awareness

Nach [Russell und Norvig 2003, Weiss 1999] ist ein Agent unter anderem dadurch definiert, dass dieser autonom seine Umwelt über Sensoren wahrnimmt und in dieser Umwelt mittels Aktuatoren agiert. Diese Idee ist in Abbildung 2.1 dargestellt. Der Begriff der *Resource Awareness* bezeichnet allgemein die Eigenschaft, dass eine Entität sich den für sie interessanten, erreichbaren Ressourcen in ihrer Umwelt gewahr ist. Im Kontext von Multiagentensystemen werden diese Entitäten unter anderem durch Agentenplattformen bzw. Agenten repräsentiert und die interessanten Ressourcen in erster Linie durch andere erreichbare Systeme und Agentenplattformen samt ihrer angebotenen Dienste und beheimateten Agenten.

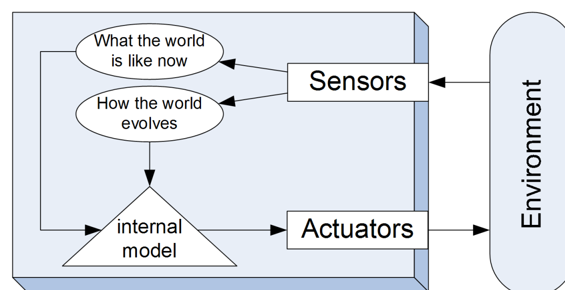

Abbildung 2.1: Interaktion eines Agenten mit seiner Umwelt(nach [Russell und Norvig 2003])

Bei den meisten Multiagentensystemen obliegt dem Administrator der Plattform bzw. dem Programmierer die Aufgabe, die Umweltinformationen der Plattform bzw. den Agenten explizit bekannt zu machen. Da sich die Umwelt in vielen Fällen dynamisch ändern kann und jederzeit und unvorhersehbar Ressourcen verschwinden bzw. neue Ressourcen hinzukommen können [Odell u. a. 2002], müssen die Umweltinformationen fortlaufend aktualisiert werden. Die manuelle Aktualisierung ist jedoch mühsam, fehleranfällig und mit steigender Komplexität der Umwelt und den Anforderungen an die Umweltinformationen nicht mehr zeitnah durch Menschen zu bewerkstelligen. Aus diesem Grund wäre es wünschenswert, wenn die Entitäten selbst ihre Umwelt wahrnehmen könnten, ohne dass der Mensch ihnen etwaige Vorgaben machen müsste. Auf diese Weise könnten sie unter Berücksichtigung ihres momentanen Zustandes die Wahrnehmung ihren eigenen Bedürfnissen anpassen, nach eigenen Interessen filtern und in selbstbestimmten Intervallen aktualisieren.

In den folgenden Abschnitten werden, beginnend mit einer allgemeinen Definition und Beschreibung der Resource Awareness, verschiedene Arten und Konzepte von Umweltmodellen, wie sie in der Literatur zu finden sind, aufgezeigt. Es werden Charakteristiken von Modellen zur Beschreibung und Klassifizierung vorgestellt (Abschnitt 2.2.1) und anschließend konkrete Umweltmodelle betrachtet (Abschnitt 2.2.2). Die Abschnitte 2.2.3 und 2.2.4 widmen sich den Entitäten, die in einer Umwelt existieren bzw. den Ereignissen, die in einer Umwelt auftreten.

Bis zu jenem Abschnitt wird davon ausgegangen, dass die Umwelt von sich aus die Perzepte an die Agenten heranträgt. Doch Informationen in einer logischen Umwelt verbreiten sich - anders als zum Beispiel das Licht in der natürlichen Welt - nicht von alleine, sondern müssen aktiv zwischen einzelnen Entitäten ausgetauscht werden. Die Abschnitte 2.3 und 2.4 widmen sich daher der *Discovery*, also dem Auffinden von Ressourcen, und stellen verschiedene Verfahren vor, wie

Informationen unter Entitäten ausgetauscht, verbreitet und gefunden werden können. Die Art und Weise, wie Informationen repräsentiert werden sollen, ist ein weiterer Kernpunkt des Konzeptes der Resource Awareness, denn dies beeinflusst wesentlich die Interoperabilität mit bereits vorhandenen Software-Infrastrukturen. Dieser Punkt wird in Abschnitt 2.5 diskutiert.

2.1 Begriffsdefinition und -abgrenzung

Der Begriff der *Resource Awareness* leitet sich von dem gebräuchlichen Konzept der *Context Awareness* ab. Die Unterschiede zwischen diesen beiden Begriffen sollen im Folgenden beschrieben werden. Hierzu wird zuerst der Begriff des Kontextes näher erläutert. Im Anschluss daran wird dem Begriff des Kontextes das Konzept einer Ressource, so wie es in dieser Arbeit gebraucht wird, gegenübergestellt. Darauf aufbauend folgen gebräuchliche Definitionen von Context Awareness und demgegenüber schließlich die Abgrenzung der Ressource Awareness.

2.1.1 Kontext

Der Begriff des Kontextes ist ein gebräuchlicher Begriff des Alltags. Doch in vielerlei Situationen werden diesem Begriff leicht unterschiedliche Bedeutungen angehaftet. Daher soll zuerst eine recht allgemein gehaltene Definition gegeben werden.

> *„[Context is] the interrelated conditions in which something exists or occurs“* (nach [Merriam-Webster 2006])

Das Objekt dieser Aussagen bezieht sich allgemein auf irgend etwas (engl. *something*). Es wird nicht näher beschrieben, welche Art von Dingen oder Ereignissen existieren oder geschehen sollen. Wichtig ist nur, dass dies unter besonderen Gegebenheiten geschieht, die in wechselseitiger Beziehung zueinander stehen (engl. *interrelated conditions*). Diese Gegebenheiten werden als Kontext bezeichnet. Da diese Definition sehr weit gefasst ist, trägt sie nicht besonders viel zum Verständnis des Konzeptes im Bereich der Informatik bei. Andere Autoren haben daher versucht, eigene Definitionen von Kontext, die stärker auf die Informatik zugeschnitten sind, zu etablieren. [Abowd u. a. 1999] definieren Kontext zum Beispiel folgendermaßen:

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

Der erste Teil dieser Definition ist ebenfalls sehr allgemein gehalten, fügt aber einen zeitlichen Aspekt in die Beschreibung von Kontext ein, indem er einen Situationskontext beschreibt. Während in der obigen Definition von Merriam-Webster noch von *something* gesprochen wurde, wird dieses Etwas nun als Person, Platz oder Objekt näher konkretisiert. Sehr spezifisch wird diese Definition in der Art des Kontextes, den sie beschreibt. In dieser Definition geht es nämlich um den Kontext der Interaktion zwischen einem Benutzer und einer Anwendung. In [Schilit u. a. 1994, Chen und Kotz 2000] wird der Begriff des Kontextes über eine Aufzählung von Aspekten beschrieben und gleichzeitig in drei Kategorien unterteilt:

> *„Three important aspects of context are: where you are, who you are with, and what resources are nearby“*

Computing Context („*what resources*") Zu dieser Kontext-Kategorie gehören zum Beispiel das Leistungsvermögen von Geräten (Speicher- und Rechenkapazität, Energie), Eigenschaften der Netzwerkverbindungen (Datenrate, Latenzzeit, Kosten) erreichbare andere Geräte (Drucker, Bildschirme, Computer), vergleiche auch Abschnitt 4.4.1.

User Context („*where you are, who you are with*") Diese Kategorie stellt den Benutzer in den Vordergrund und umfasst zum Beispiel ein Profil des Benutzers, seinen Aufenthaltsort, andere Menschen in seiner Umgebung sowie unter Umständen auch die soziale Stellung des Benutzers.

Physical Context („*where you are ... what resources are nearby*") Diese Art des Kontextes kann sich entweder auf die physikalischen Eigenschaften, also Lichtverhältnisse, Umgebungsgeräusche, Temperatur etc. beziehen oder aber allgemein auf die natürliche Umwelt samt ihrer Charakteristiken und Objekte (vergleiche auch Abschnitt 2.2.2).

2.1.2 Ressourcen

Das im Rahmen dieser Arbeit vorgestellte Konzept der Resource Awareness verwendet den Begriff einer Ressource allgemein als Quelle des Angebots, sei es das Angebot an Informationen oder das Angebot der Unterstützung bei der Bearbeitung von Aufgaben [Merriam-Webster 2006]. Damit bildet eine Ressource quasi eine Schnittmenge der oben erwähnten Kategorien des Kontextes nach Schilit. Zum Beispiel kann eine Ressource ein Computer sein, der seine Hardware für die Ausführung von Aufgaben zur Verfügung stellt (also Teil des Computing Context ist), es kann auch ein Dienst sein, der physikalische Sensordaten anbietet (und somit dem Physical Context zuzuordnen ist). Letztendlich sind Ressourcen zum Beispiel auch Softwareagenten in der Umgebung, mit denen ein anderer Softwareagent gegebenenfalls kommunizieren und interagieren möchte. In diesem Fall wären die Softwareagenten in den User Context (ggf. auch in den sozialen Kontext) des anderen Softwareagenten einzuordnen.

Man sieht, dass die semantische Trennung zwischen den Konzepten des Kontextes und der Ressource unscharf ist. Da der Begriff des Kontextes aber mehr Interpretationsspielraum lässt als der Begriff der Ressource, wird im weiteren Verlauf der Arbeit anstatt etwaiger Kontextkategorien einfach der Begriff der Ressource verwendet.

2.1.3 Awareness

Der Begriff *Awareness* bezeichnet allgemein das Bewusstsein - oder Gewahrsein, ferner auch Aufmerksamkeit oder Bewusstheit (zur Betonung einer aktiven Handlung) - gegenüber einem Sachverhalt [Wikipedia 2006c]. Er wird in verschiedenen Zusammenhängen verwendet, zum Beispiel gebraucht man im Bereich der Informatik - wie oben bereits erläutert - den Begriff der Context Awareness. Weitere Verwendungen sind Public Awareness, Social Awareness, Location Awareness etc. Im Bereich der biologischen Psychologie beschreibt Awareness die Perzeption und kognitive Reaktion eines Menschen oder Tieres auf einen Zustand oder ein Ereignis. Hierbei wird nicht notwendigerweise auch das Verstehen des Zustandes oder Ereignisses mit eingefasst [Wikipedia 2006d].

Die Essenz aus diesen beiden Beschreibungen lässt sich wie folgt zusammenfassen: Jemand oder etwas ist sich einer Sache bewusst. Dieses Bewusstsein resultiert aus der Wahrnehmung eines Zustandes oder der Veränderung eines Zustandes und führt wiederum zu der Veränderung einer internen Repräsentation. Diese Aussagen spiegeln sich auch in den Untersuchungen über

Context Awareness von Dey u.a. wieder, wobei dort vielfach auch noch die Fähigkeit der Anpassung an einen Sachverhalt beschrieben wird.

Resource Awareness sei im Folgenden verstanden als die Wahrnehmung der Existenz von Ressourcen in der Umwelt sowie die kontinuierliche Integration von Ereignissen, die den Zustand der Ressourcen sowie das Verschwinden von Ressourcen betreffen, in ein Modell der Umwelt.

Für den weiteren Verlauf dieses Kapitels sind noch zwei weitere Abgrenzungen des Begriffs der Awareness gegenüber den Konzepten des *Lookups* und der *Discovery* von Nöten. Resource Lookup bezieht sich zwar im weitesten Sinne auch auf die Wahrnehmung von Ressourcen, allerdings bezeichnet der Begriff eher das Nachschlagen oder verorten einer Ressource. Es ist ein passiver Prozess, der von einem Suchenden angestoßen wird und die Existenz einer Art von Verzeichnis voraussetzt, in dem mittels der Angabe bestimmter Kriterien nach einer Ressource gesucht wird [McGrath 2000]. Etwas weitergehend ist der Begriff der Resource Discovery[1]. Dieser Ausdruck bezieht sich auf einen eher spontanen Prozess, bei welchem sich Entitäten gegenseitig zu finden versuchen und sich dabei auch aktiv darstellen (im Sinne von bewerben) [McGrath 2000]. Im Gegensatz dazu ist Awareness zwar eng verbunden mit dem Begriff der Discovery, bedeutet darüber hinaus aber noch, dass sich Entitäten gegenseitig über Änderungen ihres Zustandes informieren. Es handelt sich hierbei um einen fortlaufenden Prozess, der - anders als bei der Discovery - kein definiertes Ende hat.

2.2 Die Umwelt eines Agenten

Der Begriff der Umwelt hat - abhängig von dem Betrachter - eine Vielzahl an Bedeutungen. Für den einen steht die global ökologische Perspektive im Vordergrund, ein anderer versteht sie als lokal räumliche Beziehung. Man kann die Umwelt auf rein gegenständliches, greifbares reduzieren, man kann aber auch die nicht sichtbaren sozialen Beziehungen in den Mittelpunkt einer Umwelt setzen. Alle diese verschiedenen Sichten auf den Begriff der Umwelt haben in gewisser Hinsicht ähnliche Merkmale und beruhen auf ähnlichen, grundlegenden Prozessen und Prinzipien (siehe Abschnitt 2.2.1). Aber sie haben durch ihren individuellen Fokus unter Umständen im Ganzen nur wenig miteinander gemein.

Im Kontext von mobilen Multiagentensystemen gibt es gleich mehrere interessante Sichten. Man kann zum Beispiel die natürliche Umwelt [Odell u. a. 2002] betrachten, in der ein Agent (z.B. ein Roboter) existiert, sich bewegt (oder auf einem mobilen Gerät bewegt wird) und deren Eigenschaften und Entitäten von diesem wahrgenommen werden, um zum Beispiel Hindernissen auszuweichen oder Werkzeuge zu erkennen und zu benutzen. Auf der anderen Seite steht die logische bzw. virtuelle Umwelt [Mertens u. a. 2005, Weyns u. a. 2004], deren Entitäten (z.B. Agentenplattformen, Netzwerkverbindungen, Agenten) zwar nicht unbedingt in der natürlichen Welt repräsentiert, aber von existenzieller Bedeutung für (mobile) Softwareagenten sind, um zum Beispiel zu kommunizieren und zu agieren. Die soziale Umwelt [Odell u. a. 2002] konzentriert sich gänzlich auf Beziehungen und deren Bedeutung zwischen Entitäten. Sie führt Konzepte wie Interaktionsprotokolle, Rollen, Gruppen etc. ein, die eine sinnvolle Interaktion und Koordination zwischen Agenten ermöglichen.

Im Folgenden werden verschiedene Definitionen für den Begriff der Umwelt erläutert sowie die allgemeinen Charakteristiken einer Umwelt vorgestellt. Diese Charakteristiken erlauben sowohl eine Beschreibung als auch eine Klassifizierung von Umweltmodellen und werden im Rah-

[1] In den folgenden Abschnitten wird oft auch der Begriff *Service Discovery* verwendet. Resource Discovery ist weitaus allgemeiner gehalten, basiert aber auf den gleichen grundlegenden Prinzipien wie die Service Discovery.

men dieser Arbeit für den Entwurf eines eigenen ressourcenbasierten Umweltmodells verwendet (vgl. Abschnitt 4.4.1). Nachfolgend werden einzelne Sichten auf den Begriff der Umwelt vorgestellt sowie ein allgemeines Modell einer Umwelt im Kontext von Multiagentensystemen präsentiert. Im Anschluss daran werden die abstrakten Entitäten, die in einer Umwelt beheimatet sind, identifiziert und beschrieben, bevor abschließend eine Klassifizierung und Betrachtung von Umweltereignissen folgt.

2.2.1 Was ist eine Umwelt?

In der Literatur finden sich viele unterschiedliche Sichten auf den Begriff der Umwelt im Kontext von Multiagentensystemen. In den folgenden Abschnitten sollen die wesentlichen Sichten erläutert werden, bevor schließlich in Abschnitt 4.4.1 die in dieser Arbeit geltende Sicht entwickelt wird. Nachfolgend seien aber zuerst die Eigenschaften und Prinzipien, die für eine Umwelt im Allgemeinen gelten, sowie ausgewählte allgemeine Definitionen des Umweltbegriffs aufgeführt.

> *„Eine Umwelt legt die Gegebenheiten und Bedingungen fest, unter denen eine Entität (z.B. ein Agent oder ein Objekt) existiert. Sie definiert die Eigenschaften der Welt in der sich ein Agent befindet und kommuniziert“* (übersetzt nach [Odell u. a. 2002])

Diese Definition von Odell erlaubt es, den Begriff der Umwelt beliebig für jeden Kontext (für eigene Anforderungen) anzupassen. Indem man die Gegebenheiten und Bedingungen spezifiziert, die eine Entität wahrnimmt und im Rahmen derer sie agieren kann, schafft man so eine individuelle Umwelt für diese Entität oder eine Menge von Entitäten. Zu den Gegebenheiten und Bedingungen gehören nicht nur alle Entitäten einer Umwelt, sie umfassen auch die Prinzipien und Prozesse die in der Umwelt gelten und wirken.

> *„Die Umwelt eines Lebewesens ist die nähere oder weiter entfernte Umgebung, die einen direkten oder indirekten Einfluss auf dieses Lebewesen und seine Lebensbedingungen ausübt. [...]“* (nach [Wikipedia 2006f])

Diese Definition der Umwelt lässt sich von Lebewesen (biologischen Agenten) ebenso auf Softwareagenten übertragen, da diese mit ihren mentalistischen und sozialen Charakteristiken einem Lebewesen nachempfunden wurden. Sie hebt den räumlichen Aspekt einer Umwelt, also die unmittelbare Umgebung einer Entität, und die Wirkung der Umwelt auf diese Entität hervor.

Charakteristiken einer Umwelt

Unter anderem in [Weiss 1999, Russell und Norvig 2003, Wooldridge 2001, Wikipedia 2006b] sind wesentliche Charakteristiken einer Umwelt aufgeführt. Mit ihrer Hilfe lassen sich verschiedene Klassen beschreiben, denen man jede beliebige Umwelt zuordnen kann. Diese Charakteristiken spielen darüber hinaus eine besondere Rolle beim Entwurf einer Umwelt bzw. der in einer Umwelt existierenden Entitäten.

Observierbar bzw. teilweise observierbar Ein Agent nimmt, laut Definition, einen Teil seiner Umwelt wahr und führt Aktionen in dieser Umwelt aus. Daher muss ein Agent zumindest den Teil der Umwelt wahrnehmen, der für die Ausführung seiner angestrebten Aktionen relevant ist. Die Wahrnehmung der vollständigen Umwelt mit all ihren Entitäten ist nur in einer sehr einfachen Umwelt praktikabel.

Deterministisch, stochastisch oder strategisch In einer vollständig *deterministischen Umwelt* hängt der Folgezustand der Umwelt ausschließlich von dem vorhergehenden Zustand sowie der Aktion des Agenten ab. Das heißt, der Agent kann schon im Vorwege voraussagen, wie sich die Umwelt durch Ausführung einer Aktion ändert. Besteht die Möglichkeit, dass Interferenzen oder Elemente mit einer gewissen Ungewissheit auf die Umwelt wirken, so spricht man von einer *stochastischen Umwelt*. Eine deterministische, aber nur teilweise observierbare Umwelt erscheint hierbei dem Agenten als eine stochastische Umwelt. Von einer *strategischen Umwelt* spricht man, wenn die Umwelt gänzlich deterministisch ist, aber der Folgezustand von den Aktionen mehrerer Agenten abhängt.

Episodisch oder sequentiell Falls eine Aufgabe, die ein Agent ausführen soll, nicht von der Ausführung vorheriger Aufgaben abhängt und nicht die Ausführung zukünftiger Aufgaben beeinflusst, dann spricht man von einer *episodischen Umwelt*. Falls dies nicht der Fall ist sagt man, die Umwelt sei *sequentiell*.

Statisch oder dynamisch In einer *statischen Umwelt* ändert sich der Umweltzustand nicht, solange der Agent mit der Auswahl seiner nächsten Aktion beschäftigt ist. Die einzigen Änderungen des Zustandes werden durch den Agenten selbst herbeigeführt. Eine *dynamische Umwelt* hingegen verändert sich auch ohne dass ein Agent eine Aktion ausgeführt hat.

Diskret oder kontinuierlich Die Unterscheidung zwischen einer diskreten oder kontinuierlichen Umwelt bezieht sich auf die Anzahl der möglichen Umweltzustände. Hat man es mit einer endlichen Menge möglicher Zustände zu tun, so spricht man von einer *diskreten Umwelt*, andernfalls von einer *kontinuierlichen Umwelt*. Ist die Anzahl möglicher Umweltzustände endlich, aber extrem hoch, so bezeichnet man dies auch als *nahezu kontinuierlich*.

Mit diesen Charakteristiken lässt sich beispielsweise die Umwelt eines Solitairespiels als observierbar, deterministisch, sequentiell, statisch und diskret beschreiben. Die natürliche Umwelt hingegen ist nur teilweise observierbar, stochastisch, sequentiell, dynamisch und kontinuierlich.

2.2.2 Umweltmodelle

Nachdem beschrieben wurde, was eine Umwelt ist und wie sich verschiedene Umweltmodelle charakterisieren und klassifizieren lassen, sollen an dieser Stelle verschiedene Sichten aufgezeigt werden. Wie eingangs erwähnt, fokussieren verschiedene Sichten unterschiedliche Aspekte einer Umwelt. Während zum Beispiel die natürliche Umwelt ein ganzheitliches Bild mit allem zeichnet, was erlebbar und erfahrbar ist, konzentriert sich die soziale Umwelt auf die Beziehungen und Strukturen zwischen den Entitäten einer Umwelt.

Natürliche Umwelt

Die natürliche Umwelt ist die Grundlage für alle Umweltmodelle. Sie umfasst alle lebenden und nicht-lebenden Dinge auf dieser Welt und diktiert diejenigen Prinzipien und Prozesse, durch die eine Population von Entitäten beeinflusst und unterstützt wird.

> *„The physical environment provides those principles and processes that govern and support a population of entities.“* (nach [Odell u. a. 2002])

Die Prinzipien der natürlichen Umwelt, zum Beispiel die Gesetze der Physik, gelten sowohl für Menschen als auch für Agenten. Nur sind für einen Agenten nicht unbedingt alle Prinzipien von

gleichem Interesse wie für den Menschen. Ein Softwareagent kann zum Beispiel die Gravitations- oder Beschleunigungskräfte bei der Migration von einem Ort zu einem anderen vernachlässigen. Dennoch gibt es Szenarien, in denen Agenten (z.B. ein Roboter) durchaus Kenntnis über die Prinzipien der natürlichen Umwelt haben müssen. Diese lassen sich in diesem Fall als Gesetze, Regeln, Bedingungen, Einschränkungen und Richtlinien auffassen, die die Existenz von Agenten bestimmen und an denen sich der Agent orientieren muss [Odell u. a. 2002].

Formal ausgedrückt ist die natürliche Umwelt ein Tupel

$$Umwelt = < Zustand_u, Prozess_u > \tag{2.1}$$

$Zustand_u$ ist hierbei die Menge aller möglichen Zustände, die die Umwelt einnehmen kann. Es gibt keine Einschränkungen bezüglich der Struktur, Domäne oder Variabilität der Zustände, weshalb diese Definition auch ohne weiteres auf andere Arten von Umwelt (vgl. logische Umwelt) übertragen werden kann. Ein einzelner Zustand wiederum ist eine Menge von Werten, die die Umwelt komplett beschreiben, inklusive aller enthaltenen Agenten und Objekte [Odell u. a. 2002, Weyns u. a. 2004].

$Prozess_u$ ist eine autonom ausgeführte Abbildung, die den Zustand der Umwelt verändert bzw. in einen neuen Zustand überführt. Autonom ausgeführt bedeutet hierbei, dass der Prozess läuft, ohne von einer externen Entität angestoßen zu werden. Die Umwelt ist also selbst aktiv bzw. dynamisch und in erster Linie unabhängig von den Aktionen der in ihr existierenden Agenten. [Odell u. a. 2002, Weyns u. a. 2004].

Die natürliche Umwelt ist für jeden Betrachter nur teilweise observierbar, stochastisch, sequentiell, dynamisch und kontinuierlich (vgl. Abschnitt 2.2.1) [Russell und Norvig 2003]. Es handelt sich hierbei somit um die komplexeste Art von Umwelt. Damit ein Softwareagent Informationen der natürlichen Umwelt sinnvoll wahrnehmen und verarbeiten kann, muss er sich auf einen für ihn interessanten Teilausschnitt dieser Umwelt beschränken. Aus diesem Grund werden Umweltmodelle aus der natürlichen Umwelt abgeleitet, die einen ganz bestimmten Fokus haben und einem besonderen Anwendungszweck genügen. Die im Folgenden vorgestellten Umweltmodelle abstrahieren daher von der natürlichen Umwelt und stellen einem Softwareagenten lediglich einen für ihn interessanten Ausschnitt der natürlichen Welt in einer angemessenen Repräsentation [2] dar.

Logische Umwelt

Der Begriff der logischen Umwelt tritt in der Literatur in leicht unterschiedlichen Formen und Bedeutungen auf [3]. So kann die logische Umwelt als eine strukturierte Institution angesehen werden, die einer Anwendung eine Schnittstelle zur natürlichen Umwelt zur Verfügung stellt (vgl. [Mertens u. a. 2004, Mertens u. a. 2005])[4].

Nach [Odell u. a. 2002] stellt die logische Umwelt darüber hinaus diejenigen Prinzipien, Prozesse und Strukturen bereit, die es einer Infrastruktur für Agenten erlaubt Ideen, Wissen, Informationen und Daten zu kommunizieren [Mertens u. a. 2005], sowie soziale Konzepte (z.B. Gruppen und Rollen, siehe *Soziale Umwelt*) zu etablieren. Odell u.a. beschreiben die folgenden Prin-

[2] Eine Repräsentation wird nachfolgend als eine strukturierte Ansammlung von Symbolen angesehen, die sich auf Dinge in der Umwelt rückbeziehen [Weyns u. a. 2004]

[3] So bezeichnen zum Beispiel Odell die logische Umwelt als *Kommunikations-Umwelt* [Odell u. a. 2002] und Weyns entweder als *Virtuelle Umwelt* [Weyns u. a. 2004, Weyns u. a. 2005] oder als *Logische Umwelt* [Weyns u. a. 2005]

[4] Ein Beispiel hierfür wäre eine logische Netzwerkverbindung zwischen zwei Computern, die lediglich eine abstrakte Repräsentation einer physikalischen Netzwerkverbindung über mehrere Zwischenstationen darstellt.

zipien der Kommunikation, welche die Basis für eine Gesellschaft (von Agenten) bildet, die sich auf Interaktionen, Bräuchen, Normen, Werten, Verpflichtungen, Abhängigkeiten etc. begründet.

Kommunikationssprache Agenten kommunizieren, um sich gegenseitig zu verständigen. Um die Verständigung jedoch überhaupt zu ermöglichen, bedarf es einer Übereinkunft bezüglich der Syntax, Semantik und Pragmatik der kommunizierten Inhalte. Es müssen verschiedene Nachrichtentypen unterschieden werden können (Annahmen, Anfragen, Antworten, Absagen usw.) sowie Ontologien zur gemeinsamen Konzeptualisierung festgelegt werden. Eine Auswahl an bereits verbreiteten Kommunikationssprachen für Agenten sind FIPA ACL [FIPA 2002b] und KQML [Finin u. a. 1994].

Interaktionsprotokolle Ein Interaktionsprotokoll beschreibt den Ablauf einer Kommunikation als eine Sequenz von Nachrichtentypen, die in festgelegter Reihenfolge zwischen Entitäten ausgetauscht werden müssen. FIPA hat bereits eine Reihe von Interaktionsprotokollen, zum Beispiel das *contract net protocol* [FIPA 2002d] oder das *publish/subscribe protocol* [FIPA 2002e], als Kommunikationsmuster standardisiert.

Koordinationsstrategien Agenten kommunizieren untereinander um ihre Ziele bzw. die Ziele ihrer Gruppe zu erreichen. Hierbei greifen sie auf die Prinzipien der Koordination, Konkurrenz, Planung und Verhandlung zurück, um in einer gemeinsamen Umwelt mit oder gegen andere Agenten bestehen zu können.

Soziale Übereinkunft Ein annehmbares soziales Verhalten von Agenten gegenüber anderen Agenten und Gruppen muss durch Befugnisse und Verpflichtungen in einem Multiagentensystem geregelt und durchgesetzt werden.

Kultur Die oben aufgeführten Prinzipien können von den kulturellen Prägungen der Agenten oder Gruppen in einer Umwelt beeinflusst werden. Hierzu gehören eine Menge von Werten, Überzeugungen, Wünschen und Vorhaben sowie Vertrauen und Moral.

Einleitend erwähnt wurde, dass eine logische Umwelt neben den oben genannten Prinzipien auch bestimmte Prozesse (vgl. *Natürliche Umwelt*) bereitstellen sollte, damit Agenten untereinander interagieren können. [Odell u. a. 2002] identifizierten hierfür folgende Prozesse:

Interaktionsmanagement Die Interaktion zwischen Entitäten wird von einem Interaktionsmanagement überwacht um sicherzustellen, dass die Kommunikation dem Ablauf des gewählten Interaktionsprotokolls folgt (hat ein Agent bekommen, was er wollte/brauchte/erwartete). Diese Überwachung kann durch die Teilnehmer des Protokolls selbst erfolgen oder als Kontrollprozess in der Umwelt implementiert werden.

Sprachverarbeitung und -kontrolle Bei der Interaktion von Entitäten ist einerseits eine semantische Übereinkunft erforderlich, auf der darunter liegenden Ebene muss andererseits aber vor allem auch eine korrekte Syntax verwendet werden. Der Prozess der Sprachverarbeitung kontrolliert, ob a) eine Sprache korrekt geparst werden kann, b) korrekt geparst werden kann, aber offensichtlich falsch oder widersprüchlich ist, oder c) korrekt, aber in dem aktuellen Kontext nicht zweckdienlich ist.

Service für Koordinationsstrategien Um das Prinzip der Koordinationsstrategien zu unterstützen, kann die Umwelt ein oder mehrere Service-Prozesse bereitstellen, die die Koordination zwischen Entitäten erleichtert. Zu diesen Services gehören zum einen der sogenannte

Directory Service (unterstützt *white-page*, *yellow-page* und *green-page* Funktionen) und zum anderen der sogenannte *Mediation Service*, der als unabhängige Instanz Aktivitäten zwischen Entitäten vermittelt.

Richtliniendurchsetzung Agenten werden von der Umwelt oder den sozialen Gruppen, denen ein Agent angehört, kontrolliert. Ein Verhalten des Agenten, welches nicht den von der Umwelt vorgegebenen Richtlinien entspricht, kann somit sanktioniert werden.

Soziale Differenzierung Der Prozess, durch den sich eine Gruppe oder Gemeinschaft von anderen Gruppen separiert oder ausprägt, wird soziale Differenzierung genannt. Durch diesen Prozess können sich Gruppen institutionalisieren und Rollen für ihre Mitglieder einführen, um erfolgreich bestehen zu können.

Aufbau einer sozialen Ordnung Der Aufbau einer sozialen Ordnung ist ein Prozess, der eine Struktur von Beziehungen zwischen sozialen Agenten herstellt. Die Struktur entsteht entweder durch formal festgelegte Verfahrensweisen oder als Ergebnis eines selbstorganisierenden Prozesses.

Die oben beschriebenen Prinzipien und Prozesse unterstützen, wie eingangs erwähnt, nicht nur den Austausch von Informationen und Daten, sondern ermöglichen darüber hinaus auch eine verbesserte Kommunikation zwischen verschiedenen Teilnehmern durch die Unterstützung von sozialen Konzepten, welche im folgenden Abschnitt näher behandelt werden.

Soziale Umwelt

Odell versteht die soziale Umwelt als Teilmenge der logischen Umwelt (siehe Abbildung 2.2), in der Agenten koordiniert miteinander interagieren um ihre jeweils eigenen Ziele oder ein gemeinsames Ziel zu verfolgen. Die Basis hierfür liefern die Prinzipien und Prozesse der Kommunikation (siehe *Logische Umwelt*), welche die Grundlage für Interaktionen, Bräuche, Normen, Werte, Verpflichtungen und Abhängigkeiten bilden. Wie in Abbildung 2.2 angedeutet, wird nicht jede Form von Kommunikation als soziales Verhalten angesehen, aber soziales Verhalten setzt Kommunikation zwingend voraus.

> *„A social environment is a communication environment in which agents interact in a coordinated manner.“* (nach [Odell u. a. 2002])

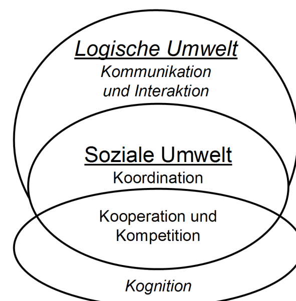

Abbildung 2.2: Soziale Umwelt: Koordination, Kooperation und Kompetition [Odell u. a. 2002]

Die soziale Umwelt besteht laut [Odell u. a. 2002] aus:

1. Einer Menge von Agenten, die über ein gemeinsames Interesse oder Ziel miteinander assoziiert sind und somit eine soziale Einheit, auch *Gruppe* genannt, bilden. Diese kann aus keinem, einem oder mehreren Agenten sowie weiteren Untergruppen bestehen und in der Umwelt selbst als sozialer Akteur auftreten.

2. *Rollen*, welche eine abstrakte Repräsentation einer Funktion, eines Services oder einer Identifikation eines Agenten innerhalb einer sozialen Einheit verkörpern. Durch Rollen werden Muster von Abhängigkeiten und Interaktionen zwischen Agenten bestimmt.

3. Allen anderen Mitgliedern, die Rollen in sozialen Einheiten darstellen.

Nach einer solchen Beschreibung kann man laut [Odell u. a. 2002] eine soziale Umwelt auch als Gesellschaft auffassen, in der Agenten in einer geordneten Gemeinschaft miteinander interagieren. Das im Folgenden vorgestellte allgemeine Modell beschreibt anhand verschiedener Schichten die Umwelt eines Multiagentensystems. Bei dieser Beschreibung lassen Weyns u.a. die Konzepte einer sozialen Umwelt allerdings unberücksichtigt. Diese werden erst in Abschnitt 4.4.1 in eine Erweiterung des allgemeinen Modells integriert.

Allgemeines Umweltmodell für MAS

Während die bisher betrachteten Umweltmodelle eine gewisse Allgemeingültigkeit besaßen, beschreibt der folgende Abschnitt ein konkretes Modell, welches auf den Kontext von Multiagentensystemen zugeschnitten ist. Weyns u.a. untersuchten in [Weyns u. a. 2004, Weyns u. a. 2005] verschiedene Umweltmodelle und fassten ihre Beobachtungen in einem dreischichtigen Umweltmodell für Multiagentensysteme (siehe Abbildung 2.3) zusammen.

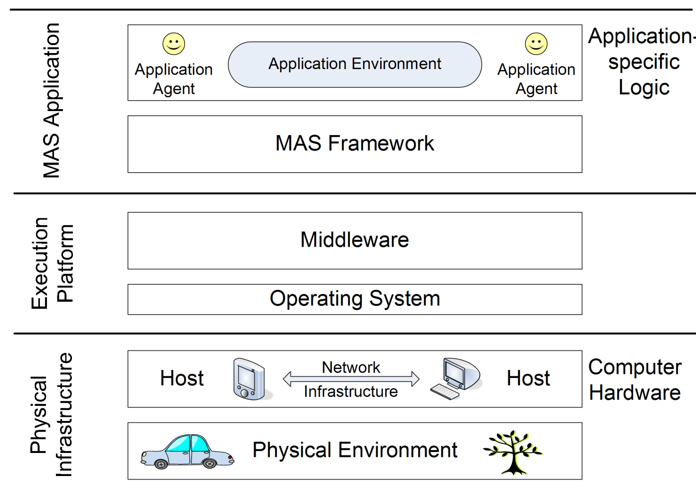

Abbildung 2.3: Allgemeines Umweltmodell für MAS (nach [Weyns u. a. 2004, Weyns u. a. 2005])

Bevor dieses Modell im Folgenden näher beschrieben wird, ist es wichtig eine Abgrenzung zwischen einem Agenten und seiner Umwelt zu finden, da diese in der Literatur oftmals nicht berücksichtigt wird. Weyns listet hierzu eine Reihe von Aussagen auf [Weyns u. a. 2004]:

- Agenten sind, was der Entwickler für eine bestimmte Anwendung programmiert. Die Software- und Hardwareinfrastruktur, auf der die Agentenanwendung läuft, stellt die Umwelt dar.

- Die Umwelt stellt für die Agenten den Kontext zur Verfügung, in dem die Agenten miteinander kommunizieren und Informationen über das zu lösende Problem zusammentragen.
- Agenten sind autonom und verfolgen proaktiv ihre Ziele. Im Gegensatz zur Umwelt können die Agenten Ziele haben, mittels derer sie einen Zustand erreichen, während die Umgebung im eigentlichen Sinn keine Ziele hat bzw. nur Ziele (oder Prozesse), die einen Zustand erhalten.
- Die Umwelt ist weitreichend und unbeschränkt, Agenten hingegen sind beschränkt und in der Umwelt ortsgebunden.
- Die Umwelt verkörpert die von einer Problemdomäne gegebene Dynamik bzw. deren „Gesetze der Physik". Die Agenten reagieren auf diese Gesetze.
- Die Umgebung lässt sich (teilweise) untersuchen. Sie implementiert, was jeder glaubt in der Domäne sehen zu können. Die lokalen Entscheidungen von Agenten hingegen sind nicht öffentlich sichtbar und können nicht zwangsläufig von anderen nachvollzogen oder geändert werden.

Nachdem durch diese Aussagen eine ungefähre Abgrenzung zwischen einem Agenten und der Umwelt aufgezeigt wurde, wird im Folgenden das Modell und die Position von Agenten und ihrer Umwelt in drei Schichten beschrieben (siehe Abbildung 2.3) [Weyns u. a. 2004, Weyns u. a. 2005]:

1. *Multiagenten-Anwendungsschicht*
2. *Ausführungsumgebung*
3. *Physikalische Infrastruktur*

Die Multiagenten-Anwendungsschicht besteht aus zwei Unterschichten: 1) der domänenspezifischen Anwendungslogik, zu der auch die Agenten sowie die Anwendungsumwelt gehören und 2) dem Multiagenten-Rahmenwerk (engl. *framework*), welches dem Entwickler höhere Programmierabstraktionen zur Verfügung stellt. Wie bereits oben erwähnt, sind in dieser Schicht Agenten und ihre Umwelt voneinander getrennt. Die Anwendungsumwelt stellt den Agenten eine Repräsentation der Domäne zur Verfügung und zusammen repräsentieren sie die Lösung für ein spezifisches Problem. Das MAS-Rahmenwerk hingegen ist anwendungsunabhängig und bietet den Agenten eine Reihe vordefinierter Mechanismen zur Kommunikation, Deliberation, Migration, Ressourcenauffindung etc.

Die Ausführungsumgebung setzt sich zusammen aus einer allgemeinen (verteilten) Middleware-Infrastruktur und virtuellen Maschinen, die auf einem Betriebssystem aufsetzen. Die Middleware bzw. die virtuellen Maschinen bieten Zugriff auf Systemressourcen der unteren Systemebenen und verstecken damit die Details von Hard- und Software. Das Gleiche gilt für das Betriebssystem, welches der Middleware standardisierte Schnittstellen zum indirekten Zugriff auf die physikalische Infrastruktur eines Systems gestattet und für die Verwaltung der Ressourcen verantwortlich ist. Die physikalische Infrastruktur, als unterste Schicht des Modells, besteht aus der Computerhardware eines Systems und der natürlichen Welt. Die Computerhardware umfasst unter anderem den Speicher, den Prozessor, die physikalischen Netzwerkverbindungen etc. auf die letztlich alle anderen Komponenten des Modells indirekt zugreifen. Die natürliche Welt bezieht sich auf - falls vorhanden - alle natürlichen (im Sinne von physikalischen) Bestandteile eines Multiagentensystems.

Ein vergleichbares Umweltmodell hat auch Mertens vorgestellt (vgl. [Mertens u. a. 2004, Weyns u. a. 2004]). In seinem Modell geht Mertens ebenfalls von verschiedenen Schichten aus, 1) der Anwendungsumwelt, welche den Kontext für Agenten bereitstellt, in dem sie Aktionen ausführen, kommunizieren und Informationen über das zu lösende Problem sammeln und 2) der Ausführungsumwelt, welche Operationen auf die physikalische Hardware abbildet. Das Modell von Weyns ist jedoch detaillierter und erhebt im Gegensatz zu dem Modell von Mertens Anspruch auf allgemeine Anwendbarkeit in Multiagentensystemen.

Das hier beschriebene Umweltmodell wird in Abschnitt 4.4.1 in leicht erweiterter Form weitergehend behandelt. In jenem Abschnitt wird ein ressourcenbasiertes Umweltmodell entwickelt, welches Agenten die Möglichkeit bieten soll, die in diesem Modell enthaltenen Entitäten zu erfassen und sich somit ihrer Umwelt gewahr zu werden.

2.2.3 Entitäten in Umweltmodellen

In den vorhergehenden Abschnitten wurden Umweltzustände, -prozesse und -prinzipien aus der Sicht verschiedener Umweltmodelle vorgestellt. Im Rahmen dessen wurde der Begriff *Entität* gebraucht, um von Objekten oder Akteuren, die in einer Umwelt existieren, zu abstrahieren. In diesem Abschnitt soll nun der Begriff Entität im Kontext von Umweltmodellen für Multiagentensysteme näher betrachtet und eine Klassifizierung verschiedener Arten von Entitäten vorgenommen werden.

Odell ersetzt den Begriff der Entität in [Odell u. a. 2002] durch die Begriffe Agent, Objekt, soziale Einheit oder allgemein durch Kommunikationspartner. Weyns sieht darüber hinaus auch die Umwelt selbst als aktive Entität [Weyns u. a. 2004] und beschreibt die in ihr enthaltenen Entitäten, mit Ausnahme der Agenten, als *Artefakte* [Weyns u. a. 2005]. Während ein Agent im Grunde genommen eine autonome, zielorientiert und sozial handelnde Entität ist, ist ein Artefakt im Gegensatz dazu eine Softwareinstanz, die lediglich eine Funktion oder einen Dienst zur Verfügung stellt, die ein Agent zum Erreichen seiner Ziele in Anspruch nehmen kann. Diese Charakterisierung führt zu einer Unterscheidung zwischen *zielorientierten Entitäten* (Agenten) und *funktionsorientierten Entitäten* (Artefakten) [Weyns u. a. 2005]. Ein Artefakt kann spezifiziert werden durch seine Funktion, seine Benutzungsschnittstelle oder eine Bedienungsanleitung für diese Schnittstelle.

Funktion Mittels einer logischen oder natürlichsprachlichen Beschreibung wird die Funktion oder der Dienst, welche von diesem Artefakt angeboten wird, sowie mögliche Vorbedingungen für die Inanspruchnahme beschrieben.

Benutzungsschnittstelle Die Beschreibung der Schnittstelle aller Operationen, die auf dem Artefakt aufgerufen werden können. Diese Beschreibung muss sowohl die Eingaben als auch die Ausgaben einer jeden Operation umfassen.

Bedienungsanleitung Die Bedienungsanleitung beschreibt in einer für andere Objekte verarbeitbaren Form, wie ein Artefakt zu benutzen ist bzw. wie die angebotenen Operationen aufgerufen werden müssen. Entsprechend ist die Bedienungsanleitung eng an die Beschreibung der Benutzungsschnittstelle gebunden.

Eine weitere Unterscheidung zwischen Artefakt und Agent beruht darauf, dass ein Artefakt, im Gegensatz zu einem Agenten, verteilt auf mehreren Knoten eines Netzwerkes laufen kann. Auch

lassen sich einzelne Artefakte zu komplexen Artefakten zusammenfügen (vgl. Dienstkompositi-on) [Weyns u. a. 2005]. Dies führt zu der Möglichkeit eines inkrementellen Modellentwurfs und hilft aus einer technischen Perspektive bei dem Entwurf und der Konstruktion von komplexen Umweltmodellen für Multiagentensysteme. Aus der Sicht eines Analytikers können Artefakte darüber hinaus verwendet werden, um existierende Umweltmodelle in einer abstrakten, allgemeinen und einheitlichen Form zu beschreiben und zu vergleichen. Hierzu lassen sich Artefakte in drei Kategorien unterteilen:

Ressourcenartefakte Auf einem abstrakten Level repräsentiert ein Ressourcenartefakt eine virtuelle oder natürliche Entität. Es vermittelt den Zugang zu Ressourcen oder stellt eine direkte Repräsentation einer Ressource in der Umwelt der Agenten dar.

Koordinationsartefakte Ein Koordinationsartefakt stellt den Agenten eine Funktion oder einen Dienst zur Verfügung, mittels dessen die Kommunikation und Koordination zwischen Agenten unterstützt wird.

Organisationsartefakte Diese Art von Artefakten haben eine Organisations- oder Sicherungsfunktion in der Umwelt der Agenten. Als Beispiel hierfür gelten die sogenannten Grenzartefakte, welche den Zugang von Agenten zu anderen Artefakten einschränken können.

2.2.4 Ereignisse in Umweltmodellen

Viele Umweltmodelle repräsentieren eine sich ständig verändernde, dynamische Umwelt. Will ein Agent seiner Umwelt gewahr sein, so muss er den aktuellen Zustand der Umwelt kennen und informiert werden, sobald sich dieser Zustand ändert. Die Änderung eines Umweltzustandes ist immer mit einem Ereignis verknüpft, welches diese Änderung hervorruft. Dieses Ereignis stellt für einen Agenten das Signal dar, sein internes Umweltmodell zu aktualisieren.

Wie in Abschnitt 2.2.2 bereits erwähnt, wird ein Umweltmodell als ein Tupel, bestehend aus einer Menge von Zuständen und einem Prozess, der die Umwelt autonom von einem Zustand in einen Folgezustand abbildet, definiert. Neben diesem autonomen Prozess, welcher der Umwelt eine Eigendynamik verleiht, können aber natürlich auch die in der Umwelt existenten Entitäten durch ihre Aktionen einen neuen Umweltzustand herbeiführen. Ein Zustandsübergang wird nachfolgend als *Umweltereignis* bezeichnet. Umweltereignisse lassen sich - wie in Abbildung 2.4 dargestellt - entsprechend ihren Urhebern in drei Klassen einteilen [Mertens u. a. 2004, Mertens u. a. 2005]:

1.Klasse: In diese Klasse fallen Ereignisse, die von dem autonomen Prozess der Umwelt ausgelöst wurden. In jeder dynamischen Umwelt gibt es einen solchen Prozess, der einen externen Eingriff in die Umwelt repräsentiert. Das Trennen einer Netzwerkverbindung durch Ziehen des Netzwerkkabels stellt in einer logischen Umwelt zum Beispiel solch einen externen Eingriff dar.

2.Klasse: Zu dieser Klasse gehören Ereignisse, die durch eine Aktion einer Entität einen internen Zustandswechsel in einer anderen Entität und somit ggf. im Umweltmodell hervorrufen. Beispiele hierfür wären die Kommunikation zwischen verschiedenen Entitäten oder der Absturz einer kompletten Ausführungsumgebung durch einen schweren internen Fehler eines ausgeführten Agenten.

3.Klasse: Zu dieser letzten Klasse zählen alle Zustandsänderungen von Entitäten, die nicht durch externe Ursachen begründet sind und ggf. einen Einfluss auf die Umwelt haben. Beispiele sind unter anderem das Fertigstellen einer von einem Agenten ausgeführten Berechnung oder das Wiedererwachen eines Agenten als Reaktion auf einen internen Stimulus.

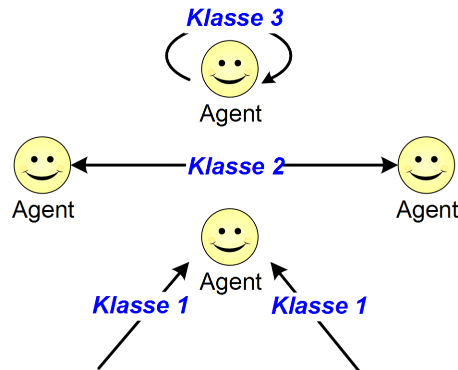

Abbildung 2.4: Klassifizierung von Umweltereignissen (eigene Darstellung)

Eine andere mögliche Kategorisierung der Ereignisse beruht auf der Einteilung nach den von einem Ereignis betroffenen Entitäten. So könnte man eine Ereignisgruppe definieren, die ausschließlich einen Effekt auf eine bestimmte Art von Artefakten oder sogar ein konkretes Artefakt hat. Zum Beispiel die Entstehung eines neues Koordinations-Artefaktes innerhalb einer Umwelt ist zwar für alle Agenten dieser Umwelt von Interesse, nicht aber für andere Artefakte.

Tritt nun ein Ereignis in der Umwelt auf, stellt sich die Frage, wie die Entitäten, die in der Umwelt existieren, von diesem Ereignis in Kenntnis gesetzt werden. In der natürlichen Umwelt geschieht dies implizit. Entitäten, die die Fähigkeit haben, den Effekt eines Ereignisses über Sensoren wahrzunehmen, tun dies, ohne von einer anderen Entität explizit darüber informiert zu werden. Wenn zum Beispiel eine Lichtquelle eingeschaltet wird, können alle Entitäten, die Photosensoren besitzen oder in irgendeiner Weise auf Licht reagieren, diesen neuen Zustand der Umwelt wahrnehmen, ohne dass ihnen dieses Ereignis explizit mitgeteilt werden muss.

In einer logischen Umwelt, in der Agenten beheimatet sind, verhält sich die Wahrnehmung anders. Da Agenten im Allgemeinen lediglich Nachrichtenkanäle als Sensoren für etwaige Umweltereignisse besitzen, müssen diese den Agenten explizit in Form von Nachrichten mitgeteilt werden. Für diese Form der Ereignisübermittlung gibt es zwei verschiedene Ansätze: 1) Nach dem Client/Server-Prinzip werden Ereignisse einer zentralen Instanz in der Umwelt gemeldet und von dieser an alle registrierten Entitäten in Form von Nachrichten übermittelt, oder 2) nach dem Peer to Peer-Prinzip übermittelt diejenige Entität, die ein Ereignis wahrnimmt (oder der Urheber des Ereignisses ist), diese Information an bekannte Netzwerkknoten und diese wiederum leiten sie weiter an ihnen bekannte Knoten usw. Auf diese Weise verbreitet sich die Nachricht, dass ein Ereignis stattgefunden hat und sich die Umwelt in einem neuen Zustand befindet, idealerweise durch die gesamte Umwelt.

Das Client/Server-Prinzip hat sich als ungeeignet erwiesen, da es mit der Größe bzw. Komplexität eines Umweltmodells und der Menge der enthaltenen Entitäten nicht skaliert und außerdem einen sogenannten *Single Point of Failure* darstellt, das bedeutet bei Ausfall dieser zentralen Instanz findet überhaupt keine Ereignisübermittlung in der Umwelt mehr statt. Der Peer to Peer-Ansatz verspricht diese Probleme zu umgehen, bringt allerdings auch neue Probleme mit sich. Was passiert zum Beispiel, wenn eine Entität aus der Umwelt ausscheidet ohne dieses Ereignis

anderen Entitäten mitzuteilen? Oder andersherum, was passiert, wenn eine neue Entität der Umwelt beitritt, aber keine anderen Entitäten kennt, denen sie dieses Ereignis mitteilen kann? Und wie verhindert man, dass die Umwelt mit Ereignisnachrichten überflutet wird? Für diese Probleme existieren verschiedene Lösungsansätze, die in den folgenden beiden Abschnitten 2.3 und 2.4 näher vorgestellt werden.

2.3 Discovery in Infrastrukturnetzwerken

Der Begriff *Discovery* (dt. Auffinden) im Kontext von verteilten Systemen bezeichnet das Anbieten, Suchen und Aufrufen von verteilten Diensten, Dienstkomponenten oder allgemein Ressourcen (vgl. Abschnitt 2.1.3). Die Grundidee hierbei ist, dass Anwendungen bzw. Geräte Aufgaben an andere Anwendungen und Geräte delegieren, die entweder speziell für die Bearbeitung bestimmter Aufgaben ausgelegt sind (z.B. Drucker), über besondere Ressourcen verfügen, die nicht allen Geräten zur Verfügung stehen (z.B. ein Internetzugang), oder die zur Zeit ungenügend ausgelastet sind, sodass sie andere bei der Bearbeitung von Aufgaben unterstützen können. Dieses Geben und Nehmen von Ressourcen setzt allerdings ein gemeinsames Verständnis der Beschreibung von Aufgaben und Ressourcen, des Anbietens und Suchens von Ressourcen und des Ausführens von Aufgaben voraus. Dieses gemeinsame Verständnis resultiert in Verfahren, welche allgemein als *Service-* oder *Resource Discovery-Verfahren* bezeichnet werden und den Austausch von Ressourceninformationen regeln.

Die Art und Weise, wie Ressourceninformationen in Netzwerken gefunden und ausgetauscht werden, hängt im Wesentlichen von der Beschaffenheit des Netzwerkes ab. So werden in der Literatur eine Reihe von Verfahren beschrieben, die das Auffinden von Ressourcen in Infrastrukturnetzwerken thematisieren und andere Verfahren behandeln wiederum ausschließlich mobile ad-hoc Netzwerke. Aufgrund der Unterschiede zwischen beiden Arten von Netzwerken ist es nicht möglich ein allgemeines Verfahren zu entwickeln, welches optimal für jedweden Anwendungszweck ist. In diesem Abschnitt werden daher eine Reihe verschiedener Verfahren für Infrastrukturnetzwerke vorgestellt, bevor im nächsten Abschnitt 2.4 Verfahren für mobile ad-hoc Netzwerke beleuchtet werden.

Zu den in der Literatur am stärksten beachteten und daher auch im Folgenden vorgestellten Verfahren für Infrastrukturnetzwerke gehören *Jini*, das *Service Location Protocol*, *Universal Plug and Play*, *Salutation* und *JXTA*. Die wichtigsten Eigenschaften hinsichtlich der Anforderungen, Architektur und eingesetzten Protokolle jedes dieser Verfahren werden in den nächsten Abschnitten näher betrachtet. In Abschnitt 2.3.7 folgt schließlich eine Diskussion, in der die einzelnen Verfahren anhand einer Reihe von Kriterien miteinander verglichen und schließlich auf deren Einsatztauglichkeit speziell in mobilen ad-hoc Netzwerken untersucht werden. Die einzelnen Kriterien, welche die Grundlage des Vergleichs und der Diskussion darstellen, werden im Folgenden vorgestellt.

2.3.1 Anforderungen und Merkmale

Bevor die einzelnen Verfahren näher betrachtet werden, sollen zuerst eine Reihe von allgemeinen Merkmalen und Anforderungen an diese Verfahren erarbeitet werden, anhand derer die Verfahren im Anschluss miteinander verglichen werden können. Letztlich ist das Ziel, unter anderem die Eignung der einzelnen Verfahren besonders für den Einsatz in mobilen ad-hoc Netzwerken zu diskutieren, welche im Kontext dieser Arbeit speziell berücksichtigt werden sollen. Sollte sich

abzeichnen, dass keines der Verfahren ausreichend für den Einsatz in derart dynamischen Umgebungen ist, so werden weitere Möglichkeiten der Interoperabilität mit anderen, speziell für diesen Einsatzkontext ausgelegten Verfahren, beleuchtet.

Die im Folgenden vorgestellten Kriterien orientieren sich daher einerseits an den Anforderungen mobiler Geräte, zum Beispiel der sparsame Umgang mit Ressourcen, andererseits an den Anforderungen mobiler ad-hoc Netzwerke, die durch ihre Dynamik hohe Ansprüche an Robustheit, Fehlertoleranz und Performanz der Verfahren stellen. Außerdem werden die allgemeinen Charakteristiken von Discovery-Verfahren ausgearbeitet, zum Beispiel die Architektur, die Repräsentation von Ressourcen und die intendierte Einsatzumgebung.

Dezentrale Operabilität

Das Anbieten von Diensten sowie das Suchen und Auffinden kann entweder über eine zentrale Instanz innerhalb des Netzwerkes oder dezentral zwischen allen Teilnehmern geschehen. Eine zentrale Instanz, welche ein Verzeichnis oder einen Index führt, hat den Vorteil, dass Anfragen schnell bearbeitet werden können, die Menge an auszutauschenden Daten im Netzwerk gering bleibt, eine globale Sicht auf das Netzwerk möglich ist und durch Replikation oder Verteilung dieser Instanz über mehrere Netzwerkknoten hinweg ein Verfahren gut skalieren kann. Auf der anderen Seite bedeutet eine zentrale Instanz aber auch, dass ein gewisser Konfigurationsaufwand betrieben werden muss und ein sogenannter *Single Point of Failure* im Netzwerk existiert, was insbesondere in mobilen ad-hoc Netzwerken ein kritischer Punkt ist, da eine ständige Verfügbarkeit und Erreichbarkeit aufgrund instabiler Verbindungen oder der Partitionierung des Netzwerkes nicht sichergestellt werden kann [McGrath 2000, Marin-Perianu u. a. 2005].

Interoperabilität

In einem verteilten System kommunizieren unter Umständen eine Vielzahl unterschiedlicher Geräte miteinander, die sich durch ihre Hard- und Software-Ausstattung grundlegend voneinander unterscheiden können. Damit ein Verfahren überhaupt sinnvoll eingesetzt werden kann, muss es mit dieser Heterogenität der Plattformen zurecht kommen. Idealerweise sollte es gänzlich plattformunabhängig sein oder zumindest möglichst viele gängige Plattformen unterstützen. Einen Schritt weiter geht die Anforderung, dass einzelne Verfahren sogar miteinander interoperieren können sollen, sich also die Protokolle und Repräsentationen von einem Verfahren entsprechend für andere Verfahren übersetzen lassen. Hierzu ist es notwendig, offene Architekturen zu bieten und vorhandene Standards und Protokolle zu berücksichtigen.

Um die Interoperabilität der Verfahren untereinander zu ermöglichen, wurden für einige Verfahren sogenannte *Proxies* und *Bridges* entwickelt. Mit ihrer Hilfe können sich Geräte und Dienste, die eigentlich mit unterschiedlichen Protokollen nach Diensten suchen und Dienste anbieten, gegenseitig finden. So können zum Beispiel SLP und Salutation über das Bluetooth SDP Dienste nachfragen und anbieten [Bettstetter und Renner 2000]. SLP kann mit Hilfe der *Java Driver Factory* Dienstobjekte bei einem Jini Lookup Service registrieren [Guttman 1999]. Und die Java-Version von Salutation-Lite kann sowohl die Jini-Funktionen emulieren als auch zwischen Salutation und SLP abbilden [Bettstetter und Renner 2000].

Awareness-Unterstützung

Wie bereits in Abschnitt 2.1 dargelegt, gibt es einen entscheidenden Unterschied zwischen *Discovery* und *Awareness*. Von Awareness spricht man, wenn eine Entität fortlaufend über Änderungen im System informiert wird ohne dass sie aktiv periodisch Statusinformationen der ihr bekannten Ressourcen abfragen muss. Auf diese Weise kann eine Entität sofort reagieren, wenn eine neue Ressource im System angeboten wird oder eine bereits angebotene Ressource nicht mehr verfügbar ist. Diese Eigenschaft ist besonders in dynamischen Umgebungen, zum Beispiel mobilen ad-hoc Netzwerken, von großem Nutzen, hat aber den Nachteil eines erhöhten Datenübertragungsvolumens im gesamten Netzwerk.

Unterstützung von Lease-Mechanismen

Lease - auf deutsch mieten - bedeutet, dass eine Ressource lediglich für eine bestimmte Zeit zur Verfügung gestellt wird. Im Kontext von Service Discovery-Verfahren sind dies zum Beispiel die Register eines Verzeichnisses, in die sich ein Dienst für eine bestimmte Mietzeit (engl. *lease time*) eintragen kann. Endet die Mietzeit, ohne dass sie seitens des Dienstes verlängert wurde, wird der Eintrag wieder aus dem Register entfernt. Auf diese Weise kann sich ein System von Altlasten befreien, zum Beispiel von registrierten, aber nicht mehr vorhandenen Diensten. Insbesondere für mobile ad-hoc Netze ist ein derartiger Mechanismus wichtig, da Geräte unvorhersehbar aus dem Netzwerk ausscheiden können und in diesem Fall für die Geräte keine Möglichkeit besteht, ihre Dienste ordnungsgemäß abzumelden.

Ressourcenbedarf

Der Bedarf an Ressourcen eines Verfahren ist für manche Anwendungsszenarien von großer Bedeutung. Speziell für den Bereich der mobilen Geräte bzw. der mobilen Netzwerke hängt die Entscheidung für oder gegen ein bestimmtes Verfahren nicht zuletzt von dessen Speicher-, Prozessor- und Netzwerkanforderungen ab. Zum Beispiel stellt ein in der Programmiersprache Java implementiertes Verfahren prinzipbedingt höhere Minimalanforderungen an die Geräteressourcen, als Implementationen in anderen Sprachen, da Java eine interpretierte Sprache ist, deren Ausführungsumgebung relativ hohe Speicher- und Prozessoranforderungen hat. Ähnliches gilt für Verfahren, die intensiv Gebrauch von Broad- oder Multicast-Nachrichten machen, da hierbei unter Umständen ein hohes und nicht zwangsläufig kalkulierbares Datenaufkommen im gesamten Netzwerk entsteht.

Reichweite

Ob ein Verfahren nur in lokalen Netzwerken oder auch in Weitverkehrsnetzen eingesetzt werden kann, wird durch die Reichweite eines Verfahren bestimmt. Ausschlaggebend hierfür sind die verwendeten Mechanismen und Technologien. Zum Beispiel sind Verfahren, die auf den Versand von Broad- oder Multicast-Nachrichten angewiesen sind, auf administrativ definierte Netzwerkdomänen (z.B. ein Unternehmensnetzwerk) beschränkt und werden über deren Grenzen hinaus nicht weitergeleitet. Das bedeutet, Dienste können nur in einer bestimmten Domäne angeboten und gefunden werden und Verfahren in unterschiedlichen Domänen können ohne jedwede Berührungspunkte nebeneinander koexistieren. Neben dieser technischen Beschränkung kann es darüber hinaus sinnvoll sein, die Reichweite zum Beispiel einer Multicast-Nachricht von sich aus zu begrenzen, damit ein Netzwerk nicht mit Nachrichten überflutet wird (vgl. *Convergence*

Multicast-Verfahren [Friday u. a. 2004]). Kann ein Verfahren hingegen auch in Weitverkehrsnetzen operieren, ist die Reichweite einer Nachricht zwar prinzipiell nicht beschränkt, doch ist es in diesem Fall zusätzlich wünschenswert, dass ein Verfahren Mechanismen bietet, die es Geräten hinter technischen Barrieren (z.B. *NAT-Router*, *Firewalls* etc.) erlaubt, Dienste für andere Geräte anzubieten und selbst Dienste anderer Geräte in Anspruch zu nehmen.

Dienstbeschreibung und -filterung

Eines der wichtigsten Ziele bei der Suche nach Diensten ist es, genau den gewünschten Dienst zu finden, der den eigenen Anforderungen entspricht. Um dies zu unterstützen sollte die Möglichkeit bestehen, bereits in den Suchanfragen genaue Dienstbeschreibungen und -attribute spezifizieren zu können und diese von dem Verzeichnisdienst auswerten und die Ergebnisse vorab filtern zu lassen. Je höher die Anzahl und Vielfalt der angebotenen Dienste ist, desto wichtiger ist dieses Kriterium. Zusätzlich ist es von Vorteil, wenn der Anfragende als Ergebnis seiner Suche eine Beschreibung der gefundenen Dienste erhält, die detailliert genug ist, dass der Anfragende sich anhand dieser Informationen letztendlich für einen Dienst entscheiden kann.

2.3.2 Jini

Die Firma *Sun Microsystems Inc.* hat mit dem *Jini*-Projekt [Sun 2001a, Sun 2001b] ein Rahmenwerk und eine Architektur zum Programmieren verteilter Anwendungen in heterogenen Netzwerken spezifiziert. Ziel ist es, Gruppen von Geräten und Softwarekomponenten in einem einzigen dynamischen, verteilten System zusammenzuführen. Grundgedanke ist hierbei, dass Komponenten einer verteilten Anwendung Dienste bereitstellen, die in einem Netzwerk angeboten und von anderen Komponenten nachgefragt werden können.

Jini spezifiziert die Infrastruktur und die Protokolle für das Bereitstellen und Auffinden von Diensten. Wie in Abbildung 2.5 dargestellt, ist das zentrale Element der Jini-Architektur ein sogenannter *Lookup Service*. Dieser Dienst wird von einem Knoten innerhalb des Netzwerkes angeboten und führt ein zentrales Diensteverzeichnis, welches jedoch zwecks besserer Skalierbarkeit und Ausfallsicherheit auch über eine Kaskade von mehreren Knoten repliziert und gegebenenfalls föderiert werden kann [McGrath 2000]. Anwendungen können sich bei diesem Lookup Service als Dienstanbieter registrieren, anhand bestimmter Vorgaben andere Dienstanbieter suchen und schließlich Dienste mittels *Remote Method Invocation* direkt aufrufen. Hierzu spezifiziert Jini drei verschiedene Protokolle: 1) das *Discovery*-, 2) das *Join*- und 3) das *Lookup*-Protokoll.

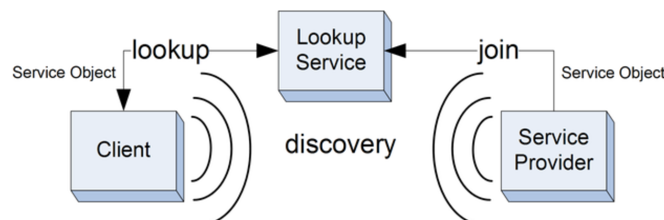

Abbildung 2.5: Jini-Protokolle: discovery / join / lookup (nach [Cheng 2002])

Mittels des Discovery-Protokolls kann ein Client nach einem Lookup Service suchen um bei diesem Dienste zu registrieren oder nach Diensten zu suchen. Hierbei wird über eine Multicast-Anfrage im Netzwerk nach einem passenden Lookup Service gesucht. Zusätzlich versendet der Lookup Service von sich aus periodisch Multicast-Nachrichten, die seine Verfügbarkeit anzeigen.

Sobald ein Lookup Service im Netzwerk ausgemacht wurde, kann sich ein Dienst mittels des Join-Protokolls bei dem Lookup Service registrieren, um fortan von anderen Diensten gefunden und in Anspruch genommen werden zu können. Jede Registrierung von Diensten bei einem Lookup Service geschieht nur für einen bestimmten Zeitraum (engl. *lease interval*). Vor Ende dieses Zeitraums muss ein Dienst seine Registrierung auffrischen, ansonsten wird er nicht weiter in dem Index geführt. Im Rahmen der Registrierung muss der anbietende Dienst ein Objekt (einen sogenannten *RMI-Stub*) mit der Schnittstellenbeschreibung seiner angebotenen Methoden an den Lookup Service übermitteln, dieses enthält lediglich Informationen wie bestimmte Methoden aufgerufen werden können, jedoch keinerlei qualitative Aussagen über den Dienst an sich.

Um einen bestimmten Dienst zu finden, sendet ein Client eine Dienstanfrage mittels des Lookup-Protokolls an den Lookup Service. Eine solche Suchanfrage ist im Grunde eine einfache Schablone um Strings zu vergleichen. Empfängt der Lookup Service eine solche Anfrage, so vergleicht er die spezifizierten Attribute mittels eines einfachen String-Vergleichs mit den registrierten Dienstbeschreibungen in seinem Verzeichnis [McGrath 2000]. Als Ergebnis übersendet er dem Anfragenden eine Liste von *RMI Stubs* auf denen der Anfragende direkt die Dienstmethoden aufrufen kann. Die Ausführung erfolgt dann transparent mittels *Remote Method Invocation* über das Netzwerk bei dem entsprechenden Dienstanbieter. Auch dieses Schnittstellenobjekt wird vom Lookup Service mit einer Lease-Zeit versehen, die es dem Client erlaubt, den Dienst lediglich in einem bestimmten zeitlichen Rahmen aufzurufen. So kann sichergestellt werden, dass zu jeder Zeit eine maximale Anzahl an Clients auf einen Dienst zugreifen [Bettstetter und Renner 2000, McGrath 2000].

Außerdem bietet Jini die Möglichkeit, dass sich Entitäten bei Eintreten eines bestimmten Ereignisses informieren lassen. Ein solches Ereignis könnte zum Beispiel das Registrieren, Deregistrieren oder die Zustandsänderung eines Dienstes sein. Zusätzlich kann sich ein Dienst benachrichtigen lassen, sobald der Lookup Service eine Anfrage erhalten hat, die auf diesen Dienst passt. Dieser Mechanismus ermöglicht die spontane Dienstkomposition in dynamischen Umgebungen, erlaubt aber auch - zusammen mit den oben beschriebenen *Lease*-Mechanismen - eine Selbstheilung des Jini-Systems [McGrath 2000].

In Situationen, in denen kein Lookup Service gefunden werden kann, gibt es seitens der Dienstsuchenden die Möglichkeit des sogenannten *Peer Lookup*. Anstatt eine Anfrage per Unicast an den Lookup Service zu schicken, versendet der Client eine Identifikationsanforderung, welche normalerweise von dem Lookup Service versendet wird, per Multicast an das gesamte Netzwerk. Auf diese Anforderung hin versuchen sich nun die Dienstanbieter mit ihren Dienstbeschreibungen bei dem Client zu registrieren, als ob er ein Lookup Service wäre. Der Client seinerseits durchsucht die erhaltenen Registrierungen nach passenden Dienstbeschreibungen und verwirft die anderen [Sun 2001a].

Jini ist plattformunabhängig, da es komplett auf der Programmiersprache Java basiert. Das Anbieten und Suchen von Diensten basiert auf dem Versand von serialisierten Java-Objekten, was es schwierig macht, den Nachrichtenaustausch unabhängig von der Programmiersprache und interoperabel mit anderen Verfahren zu gestalten [McGrath 2000]. Eine Lösung hierfür stellt die Errichtung sogenannter Brücken (engl. *bridges*) dar. In [Guttman 1999] wird zum Beispiel eine Brücke vorgestellt, die es dem Service Location Protocol (nachfolgend beschrieben) erlaubt, mittels spezieller Treiber Java-Objekte zu erstellen und diese für die Kommunikation mit Jini-Plattformen zu verwenden.

2.3.3 Service Location Protocol

Das *Service Location Protocol* (SLP) [5] ist ein Protokoll, vorgeschlagen von der Internet Engineering Task Force (IETF), zum Auffinden von Diensten in einem lokalen TCP/IP-basierten Netzwerk. Es existieren eine Reihe von Referenzimplementationen für verschiedene Plattformen und kann daher, auch durch die Berücksichtigung verschiedener Standards wie DHCP, LDAP, IPv6 etc, in heterogenen Umgebungen eingesetzt werden [Guttman 1999, Guttmann u. a. 1999]. Das Konzept von SLP sieht drei Rollen in einem Netzwerk vor: 1) einen *User Agent* (UA), welcher im Auftrag eines Benutzers oder einer Anwendung nach Diensten sucht, 2) einen *Service Agent* (SA), der einen Dienst anbietet und bewirbt und 3) einen *Directory Agent* (DA), der Informationen über angebotene Dienste von den SAs sammelt und auf Anfrage den UAs zur Verfügung stellt [6].

Bevor ein UA oder ein SA einen DA kontaktieren kann, um Dienste zu registrieren oder nach Diensten zu suchen, muss der Agent einen DA im Netzwerk kennen bzw. suchen. Das SLP sieht hierfür vier verschiedene Vorgehensweisen vor: 1) statische, 2) aktive und 3) passive Ermittlung sowie 4) Ermittlung über einen DHCP-Server [Perkins 1998]. Bei der statischen Ermittlung wird den Agenten die Adresse eines DAs durch eine Konfigurationsdatei oder manuell durch den Administrator mitgeteilt. Bei der aktiven Ermittlung der Adresse schicken die UAs bzw. SAa Multicast-Anfragen an das gesamte Netzwerk und warten, bis ein DA oder SA antwortet. Hierbei wird durch Verwendung des *SLP Multicast Convergence Algorithmus* (siehe [Guttman 1999, Friday u. a. 2004]) sichergestellt, dass der Anfragende nicht durch eine übergroße Anzahl an Antworten überflutet wird indem der Wert des sogenannten *Time To Live*-Feldes in den Nachrichten auf einen möglichst niedrigen Anfangswert gesetzt und bei Bedarf schrittweise inkrementiert wird. Bei der passiven Ermittlung warten die UAs bzw. SAs, bis sie eine Multicast-Nachricht eines DAa erhalten, welche periodisch von diesem ausgesandt wird. Und bei der Ermittlung eines DAs über DHCP wird den Agenten mittels eines Optionsfeldes in den DHCP-Nachrichten die Adresse von diesem mitgeteilt [Perkins 1998, Guttman 1999].

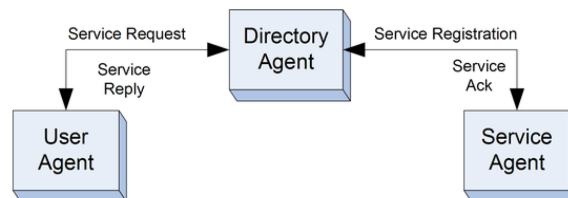

Abbildung 2.6: SLP-Agenten: Registrieren und Suchen von Diensten (nach [Bettstetter und Renner 2000])

Sowohl das Registrieren als auch das Suchen von Diensten (siehe Abbildung 2.6) folgt einem ähnlichen Schema, welches auch bei Jini Verwendung findet. Dienstanbieter registrieren eine Beschreibung ihres Dienstes bei dem DA (vgl. *Join*-Protokoll bei Jini) und Dienstsuchende stellen Anfragen an den DA und erhalten die Adresse eines passenden Dienstanbieters (vgl. *Lookup*-Protokoll bei Jini). Und ähnlich, wenn auch einfacher als das Lease Management von Jini, müssen auch die SLP-Dienstbeschreibungen bei den DAs regelmäßig seitens der SAs erneuert werden. Andernfalls werden sie gelöscht [Bettstetter und Renner 2000, Guttman 1999].

[5] Hier vorgestellt ist SLP, Version 2. Die IETF hat bereits die dritte Version dieses Protokoll-Standards in Arbeit, welches auch Weitverkehrsnetze adressiert, aber in der Literatur noch wenig Beachtung gefunden hat. Weitere Informationen über SLPv3 findet man unter *http://www3.ietf.org/proceedings/99mar/I-D/draft-ietf-svrloc-wasrv-01.txt*

[6] In diesem Kontext wird ein *Agent* lediglich als Stellvertreter angesehen und hat wenig mit dem in dieser Arbeit gebräuchlichen Begriff eines Softwareagenten gemein

Die Suchanfragen eines UAs können einen zu suchenden Diensttyp oder eine Reihe von Dienstattributen beinhalten. Der Vergleich von Attributen wird durch eine Schablone und ein *LDAPv3*-Prädikat[7] spezifiziert, welches eine mächtige Syntax für einen Vergleich bietet [Perkins 1998]. Als Antwort auf eine Suchanfrage erhält der Anfragende schließlich eine Dienst-URL, zum Beispiel *service:printer:lpr://hostname*, über die ein Dienst aufgerufen werden kann [McGrath 2000].

SLP kann - ebenso wie Jini - in zwei verschiedenen Operationsmodi laufen: der erste Modus sieht - wie oben bereits erläutert - drei Agentenrollen in einem Netzwerk vor, der zweite Modus kommt ohne den DA aus. In dem ersten Modus stellt ein Gerät den DA zur Verfügung, welcher als zentrales Diensteverzeichnis fungiert, aber auch zwecks höherer Ausfallsicherheit, besserer Lastverteilung und somit höherer Skalierbarkeit auf mehreren Knoten repliziert werden kann. Dieses zentrale Verzeichnis ist jedoch optional. Im zweiten Modus ohne einen DA kommunizieren die UAs und die SAs direkt miteinander indem Dienstanfragen und Dienstangebote über Multicast-Nachrichten an alle Teilnehmer des Netzwerkes verschickt werden. Dieser Modus bietet sich aufgrund seiner einfachen Mechanismen besonders für sehr kleine Netzwerke an, eignet sich jedoch aufgrund des erhöhten Verkehrsaufkommens nicht für größere Netze [Guttman 1999].

2.3.4 Universal Plug and Play

Universal Plug and Play (UPnP) wurde Ende der Neunziger von einem Industriekonsortium unter der Führung von *Microsoft* entwickelt [UPnP 2000, UPnP 2003]. Ziel ist es, Geräte (Computer, Router, Drucker, Stereoanlagen etc.) in einem kleinen, lokalen Netzwerk einfach miteinander kommunizieren zu lassen ohne eine manuelle Konfiguration durchführen zu müssen. Hierfür definiert UPnP Mechanismen zur automatischen Konfiguration von Geräten, zum Auffinden von Diensten und zur Kontrolle der Geräte. Diese Mechanismen bauen auf einer Reihe von standardisierten IP-basierten Netzwerkprotokollen und Datenformaten auf. Insbesondere die XML-Kodierung der Nachrichten macht UPnP prinzipiell interoperabel mit anderen Nicht-IP basierten Netzwerken, verlangt von deren Teilnehmern jedoch den Umgang mit diesem Format [McGrath 2000].

Die grundlegenden Elemente in einem UPnP-Netzwerk sind *Geräte*, *Dienste* und *Kontrollpunkte*. Geräte dienen lediglich als eine Art Behälter für Dienste und Kontrollpunkte und werden durch eine kurze XML-Beschreibung des Gerätes und der von diesem angebotenen Dienste repräsentiert. Ein Dienst ist die kleinste Kontrolleinheit. Er wird durch eine eigene XML-Beschreibung repräsentiert, die die angebotene Schnittstelle sowie den aktuellen Zustand beschreibt. Diese Beschreibungen sind universell und UPnP beschränkt nicht den Inhalt der Beschreibungen. So können Dienste in ihre Beschreibungen unter anderem auch Logos, Nutzungsbedingungen, Statusinformationen etc. einschließen.

Jedes Gerät, welches von anderen Geräten angebotene Dienste in Anspruch nehmen möchte, muss einen Kontrollpunkt implementieren. Ein Kontrollpunkt ist eine Steuereinrichtung, die andere Geräte und Dienste im Netzwerk finden und kontrollieren kann. Nachdem ein Kontrollpunkt ein Gerät oder einen Dienst gefunden hat, kann er sich die jeweiligen XML-Beschreibungen holen, Dienste aufrufen und den Status von Geräten und Diensten überwachen. Wenn ein Gerät dem Netzwerk beitritt, kann dieses über das *Discovery*-Protokoll den anderen Kontrollpunkten im Netzwerk eigene Dienste anbieten bzw. Kontrollpunkte können über dieses Protokoll nach Diensten anderer Geräte suchen (siehe Abbildung 2.7). Das verwendete

[7] siehe hierzu http://www.ietf.org/rfc/rfc2254.txt

Discovery-Protokoll stützt sich auf das von der IETF vorgeschlagene *Simple Service Discovery Protocol* (SSDP) [UPnP 1999] und beruht auf dem periodischen Versenden von Multicast-Nachrichten [Bettstetter und Renner 2000].

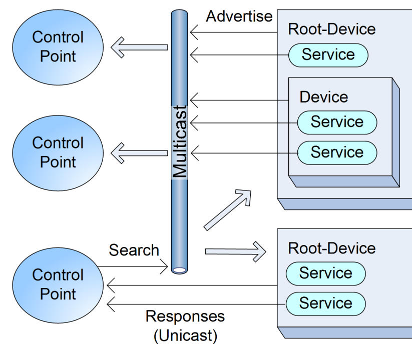

Abbildung 2.7: UPnP: Anbieten und Suchen von Diensten über Multi- und Unicast (nach [UPnP 2003])

Bei der Suchanfrage können Kontrollpunkte XML-Tags und -Attribute (oder eine beliebige Kombination) angeben, die von den Diensten mit ihrer eigenen Beschreibung verglichen werden. Als Antwort auf einen erfolgreichen Vergleich wird den Anfragenden lediglich eine URL und ein Gerätetyp übermittelt. Über diese URL kann in einem zweiten Schritt die detaillierte XML-Beschreibung abgerufen werden [McGrath 2000]. Kontrollpunkte, die einen angebotenen Dienst in Anspruch nehmen möchten, finden in den detaillierten XML-Beschreibungen die genauen Schnittstellen-Spezifikationen der Dienste, können daraufhin bei Bedarf über diese Schnittstellen Aktionen ausführen (mittels des *Simple Object Access Protocol, SOAP* [W3C 2003]) und sich über die Ergebnisse informieren lassen.

Hinsichtlich der Awareness gibt es zwei Möglichkeiten, die es Teilnehmern eines UPnP-Netzwerkes ermöglichen, fortlaufend über den Zustand des Systems informiert zu werden. Die erste - recht simple Möglichkeit - ist das periodische Versenden von Multicast-Anfragen seitens eines Kontrollpunktes, der einen bestimmten Dienst in Anspruch nehmen möchten. Erhält der Kontrollpunkt keine Antwort, so wird die Anfrage in bestimmten Intervallen erneut versandt [Friday u. a. 2004]. Wenn schließlich eine Antwort auf die Anfrage eintrifft, ist davon auszugehen, dass der Dienst gerade neu angeboten wurde und zur Nutzung durch den Kontrollpunkt bereitsteht. Die zweite und elegantere Möglichkeit betrifft die Zustandsüberwachung von Diensten. Wenn ein Dienst seinen Zustand ändert, wird über die *Generic Event Notification Architecture* (GENA) ein Ereignis generiert. Dienste und Kontrollpunkte können sich für bestimmte Ereignisse bei einem Dienst registrieren und werden benachrichtigt, sobald dieses Ereignis eintritt [Friday u. a. 2004].

Das UPnP-Verfahren wurde für kleine, lokale Netzwerke entwickelt. Typischerweise zeichnen sich derartige Netzwerk durch eine relativ hohe Datenrate und geringe (oder keine) Übertragungskosten aus. Die Protokolle gehen daher nicht besonders sparsam mit dem Datenübertragungsvolumen um. Da beispielsweise ein Großteil der Kommunikation über das Versenden unverlässlicher Multicast-Nachrichten stattfindet, werden diese gleich in dreifacher Ausführung versendet um den möglichen Verlust einer Nachricht zu kompensieren. Außerdem verschicken alle Dienste in regelmäßigen Intervallen (normalerweise 30 Minuten) erneut eine Dienstbeschreibung an alle Teilnehmer des Netzwerkes [Friday u. a. 2004].

Ein Blick auf die Architektur und die Protokolle offenbart eine Reihe von Ähnlichkeiten zu Jini und SLP, durch welche die Entwicklung von UPnP beeinflusst wurde [McGrath 2000]. Zum Beispiel das periodische Anbieten und Suchen von Geräte- oder Dienstbeschreibungen mittels Multicast-Nachrichten ähnelt den anderen Protokollen. Allerdings ist der Inhalt der Nachrichten ein anderer, da UPnP hier auf XML-basierte Beschreibungen setzt, die über einen zweistufigen Prozess zugreifbar sind (zuerst eine allgemeine Beschreibung in der Nachricht, dann eine detaillierte Beschreibung, die über eine URL erreichbar ist). Weiterhin ist festzustellen, dass UPnP, genau wie SLP und Jini, auch ohne ein zentrales Diensteregister auskommt. Bei SLP und Jini ist dieses Register (die *Directory Agents* bzw. der *Lookup Service*) jedoch optional, wohingegen UPnP die Möglichkeit gar nicht vorsieht. Das Suchen und Anbieten von Diensten geschieht ausschließlich dezentral über Multicast-Nachrichten [Bettstetter und Renner 2000, Frank u. a. 2004].

2.3.5 Salutation

Entwickelt von einem Industriekonsortium, dem *Salutation Konsortium*, erlauben die Salutation-Architektur und die Salutation-Protokolle die spontane Konfiguration sowie das Suchen und Auffinden von Geräten und Diensten in einem Netzwerk. Die Architektur definiert ein abstraktes Modell bestehend aus drei Komponenten: 1) *Clients*, die einen Dienst in Anspruch nehmen möchten, 2) *Server*, die Dienste anbieten und 3) *Salutation Manager*, die als Diensteverzeichnis fungieren und die Kommunikation zwischen Clients und Server vermitteln [Bettstetter und Renner 2000, McGrath 2000, Wiklander 2001, Elenius 2003].

Ein Salutation Netzwerk besteht aus einer Vielzahl an Salutation Managern, welche lokal auf den Geräten der Clients und Server laufen. Im Normalfall hat jedes Gerät seinen eigenen Salutation Manager, dieser kann jedoch alternativ auch von mehreren Clients und Servern gemeinsam genutzt werden (siehe Abbildung 2.8). Salutation Manager können sich untereinander auch in einer hierarchischen oder anderweitigen Graphenstruktur organisieren und tauschen in diesem Fall die Beschreibungen ihrer jeweils registrierten Dienste aus. Die so replizierten Informationen sorgen für einen schnelleren und effizienteren Zugriff auf die Daten und bieten eine höhere Ausfallsicherheit [McGrath 2000].

Über sogenannte *Transport Manager* abstrahieren Salutation Manager von der zugrunde liegenden Kommunikationsinfrastruktur und können untereinander über verschiedene Transportmedien und -protokolle kommunizieren. Außerdem ist es für Clients und Server möglich, die gesamte Kommunikation über einen Salutation Manager als Vermittlungsstelle laufen zu lassen, wodurch sich die Kommunikation ebenfalls über verschiedenartige Nachrichtenkanäle transportunabhängig bewerkstelligen lässt. So existieren zum Beispiel Transport Manager, die auch Bluetooth und IrDA unterstützen [Bettstetter und Renner 2000, McGrath 2000].

Da das Salutation-Protokoll auf *SunRPC* (Remote Procedure Call, entwickelt von der Firma Sun Microsystems Inc.) [Coulouris und Dollimore 1994] basiert, einer Technik für den entfernten Prozeduraufruf, können Clients und Server lediglich Broad- anstatt Multicast verwenden um Salutation Manager im lokalen Netzwerk zu finden [McGrath 2000]. Hat ein Server einen Salutation Manager gefunden, kann er seine Dienste bei diesem registrieren. Zur Beschreibung von Diensten definiert Salutation ein spezifisches (erweiterbares) Format. Dieses Format enthält den Diensttyp sowie weitere Dienstattribute. Bei einer Suche nach bestimmten Diensten können die Clients entweder nach einem solchen Diensttyp oder nach dessen Dienstattributen suchen, wobei neben einem einfachen String-Vergleich auch eigene Vergleichsverfahren registriert werden können. Sobald der Salutation Manager auf Anfrage eines Clients einen passenden Dienst ge-

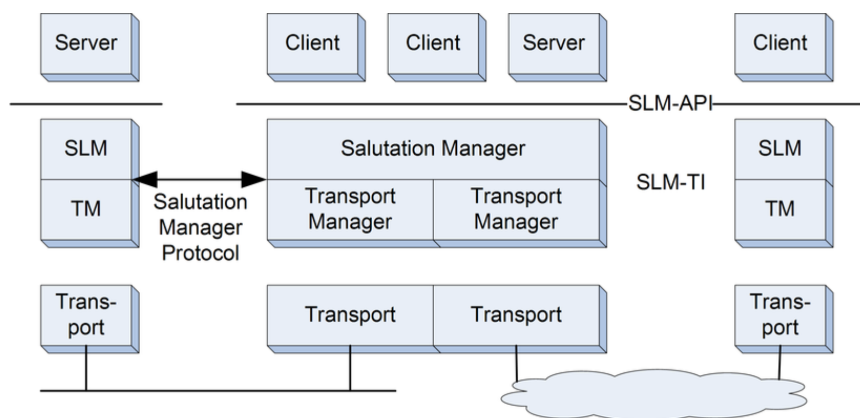

Abbildung 2.8: Salutation: Übersicht über die Salutation-Architektur (nach [Wiklander 2001])

funden hat, übermittelt er dessen Adresse sowie ein sogenanntes *Personality Profile*. Dieses Profil beinhaltet eine detaillierte Beschreibung des Dienstes sowie seiner Schnittstellen [McGrath 2000]. Clients können sich beim Salutation Manager registrieren um informiert zu werden, sobald sich ein neuer Dienst registriert oder ein bereits bekannter Dienst nicht mehr verfügbar ist. In diesem Fall prüft der Salutation Manager in regelmäßigen Abständen aktiv (anders als zum Beispiel Jini's Lease Management) die Verfügbarkeit und sendet einen Statusbericht an den anfragenden Client [Wiklander 2001].

Genau wie SLP und Jini kann auch Salutation ohne ein zentrales Diensteregister auskommen. In diesem Fall geschieht das Anbieten und Suchen von Diensten komplett über Broadcasts im lokalen Netzwerk [McGrath 2000]. Der Einsatz ohne ein zentrales Register wird wegen des geringen Konfigurationsaufwands aber dafür höheren Datenaufkommens ausschließlich für kleine, lokale Netzwerke empfohlen.

2.3.6 JXTA

JXTA ist eine *Peer to Peer*-Infrastruktur, welche seit dem Jahr 2001 von *Sun Microsystems Inc.* als Open Source-Projekt entwickelt wird um die Interoperabilität, Plattformunabhängigkeit und Allgegenwärtigkeit von Geräten und Diensten in einem Netzwerk zu ermöglichen [Gong 2001, Traversat u. a. 2003, Elenius 2003]. Die Grundidee von JXTA ist der Aufbau eines virtuellen Netzwerkes auf einem physikalischen Netzwerk, welches es Teilnehmern (*Peers*) erlaubt direkt miteinander zu interagieren und sich selbst zu organisieren, unabhängig von ihrer Art der Anbindung an das Netzwerk. Peers können auf diese Weise Nachrichten austauschen ohne die komplexe und sich ändernde Topologie des zugrunde liegenden physikalischen Netzwerkes berücksichtigen zu müssen. Dies ermöglicht auch die transparente Kommunikation mit mobilen Peers, die sich von Ort zu Ort bewegen.

Mehrere virtuelle Subnetze (*PeerGroups*) können von jedem Teilnehmer des JXTA-Netzwerkes spontan erstellt und dynamisch auf die physikalische Infrastruktur abgebildet werden (siehe Abbildung 2.9). In diesen virtuellen Domänen befinden sich in der Regel Peers mit gemeinsamen Interessen, ähnlichen Dienstangeboten etc., die sich auf gemeinschaftliche Richtlinien (z.B. Mitgliedschaft, Austausch von Daten etc.) geeinigt haben. PeerGroups stellen nicht nur eine sichere Umgebung für den Datenaustausch zur Verfügung, sie grenzen auch logische Räume innerhalb des virtuellen Netzwerkes ab, sodass die Suche und das Anbieten von Diensten in fest definierten, nach Interessen abgegrenzten Gruppen geschieht und somit das Gesamtnetzwerk entlastet.

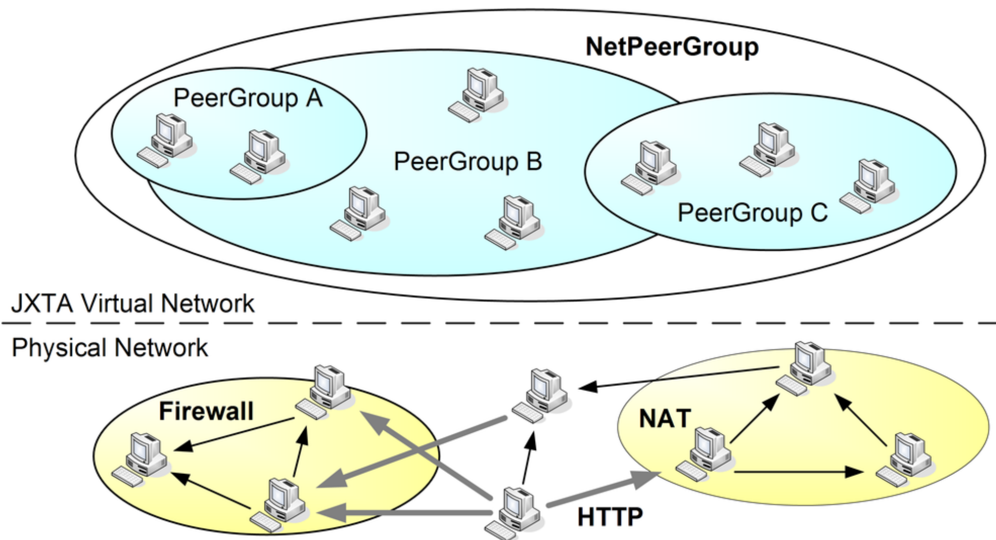

Abbildung 2.9: JXTA: Virtuelles JXTA-Netzwerk (nach [Traversat u. a. 2003])

Eine Besonderheit des JXTA-Konzeptes ist die Repräsentation von Ressourcen. Im Netzwerk werden Ressourcen anhand von sogenannten XML-*Advertisements* beschrieben. Es gibt eine Reihe vordefinierter Advertisements für bestimmte Ressourcen, wie zum Beispiel Peers, PeerGroups, Dienste, Kommunikationskanäle etc. Diese Standard-Advertisements können nach Belieben erweitert werden um so jede Art von zusätzlicher Information unterzubringen. Im Wesentlichen zeigen Advertisements die Existenz einer Ressource an und beschreiben, je nach Art der Ressource, deren Eigenschaften. Diese Advertisements sind mit einer Lebensdauer versehen (also einer Art Lease-Zeit) und werden im Netzwerk verteilt. JXTA spezifiziert nicht, wie die Verteilung bzw. Suche nach diesen Advertisements auszusehen hat, sondern definiert stattdessen ein allgemeines Protokoll, welches von Entwicklern ausdrücklich angepasst oder überschrieben werden darf [Traversat u. a. 2003].

Grundlage dieses Protokolls stellen selbst ernannte *Rendezvous Super Peers* dar. Ein Rendezvous Peer führt einen Advertisement-Index und wird von anderen Peers kontaktiert, sobald diese eigene Advertisements anbieten oder nach Advertisements suchen möchten. Neben diesem Advertisement-Index haben die Rendezvous Peers ein weiteres Verzeichnis, welches ausschließlich die Adressen weiterer Rendezvous Peers beinhaltet. Dieses Verzeichnis wird periodisch an eine zufällige Auswahl bekannter Rendezvous Peers zum Abgleich geschickt. So soll sichergestellt werden, dass Rendezvous Peers untereinander möglichst eng vernetzt sind und die logische Infrastruktur des Netzwerkes intakt bleibt. Darüber hinaus versenden die Rendezvous Peers zusätzlich sogenannte *Heartbeats* an ihnen bekannte Rendezvous Peers. Bleibt ein erwarteter Heartbeat aus, so wird angenommen, dass die Verbindung nicht weiter existiert [Traversat u. a. 2003].

Sucht ein Peer eine bestimmte Ressource, so versendet er eine Anfrage an alle Rendezvous Peers mit denen er logisch verbunden ist. Diese versuchen daraufhin passende Einträge durch String-Vergleiche von XML-Tags und -Attributen in ihrem Advertisement-Index zu finden und leiten die Anfrage ggf. an andere Rendezvous Peers zur Bearbeitung weiter. Ist ein entsprechender Eintrag im Index verzeichnet, so wird die Anfrage an den anbietenden Peer weitergeleitet, der das Advertisement ursprünglich hat indizieren lassen. Erst dieser Peer beantwortet die Anfrage indem er eine aktuelle Version des Advertisements an den anfragenden Peer zurückschickt.

Möchte ein Gerät dem JXTA-Netzwerk beitreten, so gilt es in erster Linie einen Peer zu finden, der bereits Teil des Netzwerkes ist und den neuen Peer weitervermitteln kann. Die Suche kann vollständig dezentral, zentral über ein dediziertes Verzeichnis oder in einem hybriden Verfahren erfolgen. Hierzu sieht JXTA eine Reihe von Mechanismen vor [Gong 2001]:

- *Multicast-Suche*. Über eine Multicast-Nachricht wird eine Anfrage über das lokale (physikalische) Netzwerk verschickt. Andere Peers, die bereits an dem JXTA-Netzwerk teilnehmen, können auf diese Anfrage antworten.
- *Einladung*. Peers können Einladungen (engl. *invitation*) an andere Peers verschicken. Die Informationen, die in der Einladung verpackt werden, können von dem eingeladenen Peer benutzt werden, um Zugang zu dem JXTA-Netzwerk oder einer bestimmten PeerGroup zu erhalten.
- *Kaskade*. Wenn ein Peer einen zweiten Peer findet, kann der erste Peer - mit Erlaubnis des zweiten - die diesem bekannten Peers, PeerGroups, Dienste etc. erfragen.
- *Rendezvous-Punkte*. Ein Rendezvous Peer ist ein spezieller Peer, der Informationen über ihm bekannte Peers vorhält und ggf. mit anderen bekannten Rendezvous Peers austauscht. Ein Peer, der einen Rendezvous Peer kennt, kann über diesen weitere Peers finden.

Dadurch, dass JXTA zum Auffinden von Ressourcen bzw. anderen Teilnehmern nicht zwangsläufig auf eine Multicast-Suche angewiesen ist, eignet es sich, im Gegensatz zu den anderen bisher vorgestellten Verfahren, auch für den Einsatz in größeren Netzwerken und Weitverkehrsnetzen. Außerdem unterstützt es auch Geräte, die durch *NAT-Router* oder *Firewalls* von größeren Netzwerken abgeschirmt werden, und versucht, diese in das JXTA-Netzwerk mit einzubeziehen [Gong 2001]. JXTA spezifiziert nicht, welche Programmiersprache, welches Betriebssystem und welche Netzwerkprotokolle im JXTA-Netzwerk verwendet werden müssen. Die Referenzimplementierung wurde in Java geschrieben und stützt sich auf die TCP/IP-Protokolle, so auch die meisten Anwendungen. Die JXTA-Gemeinde hat allerdings weitere Implementierungen für C/C++, Java Micro Edition etc. sowie die Unterstützung zusätzlicher Transportprotokolle (z.B. Bluetooth) hervorgebracht.

2.3.7 Diskussion

Der Wunsch, dass Geräte einem Netzwerk ohne weiteren Konfigurationsaufwand beitreten und von sich aus benötigte Dienste auffinden können, hat die Entwicklung einer Reihe von *Service Discovery*-Verfahren motiviert. Die Vielzahl an Verfahren resultiert aus den unterschiedlichen Kontexten, in denen diese Verfahren eingesetzt werden und es wird davon ausgegangen, dass in einer allgegenwärtigen digitalen Umwelt mehrere dieser Verfahren nebeneinander zum Einsatz kommen um der Heterogenität der Dienste, Geräte und Netzwerke gerecht zu werden [Friday u. a. 2004].

Es ist daher kaum möglich eine Gesamtbewertung einzelner Verfahren vorzunehmen. Stattdessen soll dieser Abschnitt die vorgestellten Verfahren anhand einer Auswahl allgemeiner Anforderungen und Merkmale, die bereits in Abschnitt 2.3.1 erarbeitet wurden, miteinander vergleichen. Zuerst sollen jedoch die grundlegenden Gemeinsamkeiten der Verfahren nochmals kurz beschrieben werden, dann werden die Verfahren hinsichtlich der gegebenen Kriterien in Beziehung gesetzt und schließlich wird ihre Eignung für den Einsatz in mobilen ad-hoc Netzwerken diskutiert.

Grundlegende Gemeinsamkeiten

Aufgrund der ähnlichen Anforderungen an all diese Verfahren bieten diese auch ähnliche grundlegende Protokolle. Hierzu gehören Protokolle mittels derer Dienste im Netzwerk angeboten werden können, Protokolle mittels derer Dienste im Netzwerk nachgefragt bzw. gesucht werden können und ein Mechanismus über den ein Dienst in Anspruch genommen bzw. ausgeführt werden kann. Analog zu den Ähnlichkeiten bei der Unterscheidung der einzelnen Protokolle weisen die Verfahren einzelnen Geräten im Netzwerk auch ähnliche Rollen zu. So wird grundlegend unterschieden zwischen einem Dienstanbieter, einem Dienstsuchenden und einem Verzeichnisdienst, der Anbieter und Suchende vermittelt [McGrath 2000, Marin-Perianu u. a. 2005]. In der konkreten Ausprägung dieser Protokolle und Rollen sowie den technischen Anforderungen unterscheiden sich die Verfahren jedoch stark, wie im Folgenden näher erläutert.

Übersicht der Unterschiede

In Tabelle 2.1 ist eine Bewertung der in den vorigen Abschnitten vorgestellten Verfahren hinsichtlich der in Abschnitt 2.3.1 erarbeiteten Kriterien und Merkmale dargestellt. Die Bewertungsskala reicht von - - (ungenügend oder nicht vorhanden) und - (schlecht) über *O* (moderat) bis hin zu + (gut) und ++ (hervorragend). Im Folgenden werden die einzelnen Bewertungen zusammenfassend noch einmal aufgeführt und kurz erklärt, weitere Details finden sich dann in den Abschnitten, in denen die Verfahren im Detail erläutert wurden.

	Dezentrale Operabilität	*Inter-operabilität*	*Awareness*	*Lease-Mechanis.*	*Ressourcen-bedarf*	*Reichweite*	*Dienstbeschr./ -Filterung*
Jini	O	O	+	++	-	O	O
SLP	O	+	- -	+	O	-	+
UPnP	+	+	+	- -	-	-	+
Salutation	++	O	O	- -	O	O	+
JXTA	++	++	- -	+	O	++	+

Tabelle 2.1: Bewertung der Discovery-Verfahren für Infrastrukturnetzwerke

Dezentrale Operabilität Jini beruht auf einem zentralen Verzeichnis, welches ggf. auf mehreren Knoten repliziert und föderiert werden kann. Über den Versand von Multicast-Nachrichten ist jedoch auch ein dezentrales Anbieten und Suchen möglich. Das Gleiche gilt für SLP, wohingegen dieses Verfahren die Föderation von Verzeichnissen nicht unterstützt. UPnP sieht keine zentrale Komponente im Netzwerk vor, stattdessen werden alle Anfragen per Multicast an alle Teilnehmer des Netzes verschickt, Antworten hingegen meist per Unicast. Auch bei Salutation gibt es keine verpflichtend zentralen Komponenten, die Salutation Manager jedes Gerätes dienen als verteiltes Diensteverzeichnis, koordinieren sich untereinander und können von mehreren Geräten gemeinsam genutzt werden. Auch JXTA ist für die dezentrale Operation ausgelegt, verwendet aber eine Menge von selbsternannten Rendezvous Peers um das Rückgrat des logischen Netzwerkes aufzuspannen. Suchanfragen und Dienstangebote werden von jedem Teilnehmer an ein oder mehrere dieser Rendezvous Peers geschickt und dort beantwortet, weitergeleitet oder indiziert.

Interoperabilität Jini basiert auf der plattformunabhängigen Sprache Java und bietet keine direkte Unterstützung für andere Sprachen, da der Nachrichtenversand auf serialisierten Java-Objekten beruht und der Aufruf von Diensten über Java's RMI Mechanismus abgewickelt wird. Für SLP gibt es eine Reihe verschiedener Implementationen für unterschiedliche Plattformen, außerdem werden diverse Standards, zum Beispiel DHCP, LDAP, IPv6 etc. berücksichtigt. Zusätzlich gibt es bereits verschiedene Brücken um mit Jini- und Salutation-Netzwerken zu kommunizieren. UPnP basiert auf einer Reihe standardisierter Netzwerkprotokolle (UDP, TCP/IP, HTTP) und verwendet für die Repräsentation von Ressourcen und Nachrichten das XML-Format. Salutation ist durch die Transport Manager unabhängig von den verwendeten Transportprotokollen, bietet jedoch keine standardisierten Schnittstellen um mit anderen Verfahren zu interoperieren. Die Referenzimplementation von JXTA ist zwar in Java geschrieben, jedoch existieren weitere Implementationen für zum Beispiel C/C++ und J2ME. Die Quellen von JXTA sind frei verfügbar, daher existieren eine Reihe von Adaptern für weitere Transportprotokolle. Die Ressourcen werden, ebenso wie bei UPnP, im XML-Format repräsentiert.

Awareness Jini und Salutation bieten Klienten die Möglichkeit sich bei dem Diensteverzeichnis zu registrieren, um anschließend über Ereignisse im Netzwerk, also das An- oder Abmelden von Diensten, benachrichtigt zu werden. Zusätzlich kann Jini einen Dienstanbieter informieren, sobald eine auf den Anbieter passende Suchanfrage empfangen wurde. SLP und JXTA unterstützen keine die Awareness betreffenden Funktionen. UPNP erlaubt die Zustandsüberwachung von Diensten über die Generic Event Notification Architecture.

Lease-Mechanismen Jini, SLP und JXTA verlangen, dass die Registrierung der Dienstbeschreibungen in regelmäßigen Intervallen bei dem Verzeichnisdienst aufgefrischt wird. Zusätzlich werden bei Jini auch die RMI-Stubs, die als Antwort auf eine Suchanfrage verschickt werden, mit Lease-Zeiten versehen, sodass der Aufruf eines Dienstes nur innerhalb eines begrenzten Zeitraums möglich ist. Salutation und UPnP verwendet keinerlei Lease-Mechanismen.

Ressourcenbedarf Jini hat einen recht hohen Ressourcenbedarf, einerseits weil es auf Java basiert, dessen virtuelle Maschine relativ hohe Speicher- und Prozessoranforderungen stellt, und andererseits da periodisch Multicast-Nachrichten über das Netzwerk versandt werden. Bei SLP, UPnP und Salutation ist der Bedarf abhängig von der jeweiligen Implementation. SLP verwendet zur Verringerung des Datenübertragungsvolumens im Netzwerk den Multicast Convergence-Algorithmus. UPnP hingegen versendet sehr viele Multicast-Nachrichten in regelmäßigen Intervallen und mehrfachen Ausführungen. Salutation verwendet Broad- anstatt Multicasts und fragt periodisch Statusinformationen von Diensten ab, anstatt mit Lease-Mechanismen zu arbeiten. Genau wie Jini basiert auch JXTA auf Java und hat daher ebenfalls relativ hohe Ansprüche an die Hardware. Im Netzwerk werden vor allem die Rendezvous Peers zu Gunsten der anderen Teilnehmer durch ein höheres Nachrichtenaufkommen belastet.

Reichweite Jini, SLP und UPnP verwenden Multicast-Nachrichten und sind daher prinzipiell auf lokale Netzwerke begrenzt. Durch statische Vorgabe von Adressen kann Jini jedoch auch Netzwerke überbrücken. Auch der Einsatz von Salutation ist durch die Verwendung von Broadcasts lokal begrenzt, kann aber durch den Einsatz spezieller Transport Manager eventuell auch in Weitverkehrsnetzen operieren. JXTA ist ausdrücklich für die Verwendung sowohl in lokalen

Netzwerken als auch in Weitverkehrsnetzen vorgesehen, es berücksichtigt sogar Geräte, die hinter Barrieren wie NAT-Router oder Firewalls versteckt sind.

Dienstbeschreibung und -filterung Dienste im Jini-Netzwerk müssen sich mit einem RMI-Stub bei dem Diensteverzeichnis registrieren, zusätzlich können weitere Attribute zur Beschreibung des Dienstes mit angegeben werden. Suchanfragen werden durch einen einfach String-Vergleich mit den Attributen des registrierten Dienstes beantwortet. SLP-Dienstbeschreibungen enthalten neben einer Reihe von Attributen noch einen Diensttyp, Suchanfragen können zudem LDAPv3-Prädikate enthalten. Auch Salutation beschreibt Dienste über einen Typ sowie optional weitere Attribute. Als Filter für Suchanfragen kommt ein einfacher String-Vergleich zum Einsatz, welcher jedoch durch benutzerdefinierte Vergleichsverfahren ersetzt werden kann. Sowohl im UPnP- als auch im JXTA-Netzwerk werden Dienste in Form der XML-Syntax beschrieben. Sie können beliebige (binäre) Informationen enthalten. Bei der Suche wird jedoch nur ein einfacher String-Vergleich über die XML-Tags und Attribute angewandt.

Eignung für mobile ad-hoc Netzwerke

Nach [Frank u. a. 2004, Friday u. a. 2004, Tyan und Mahmoud 2005, Campo 2002] eignet sich keines der besprochenen Verfahren für die Verwendung in mobilen ad-hoc Netzwerken. Die wesentlichen Gründe hierfür sind, dass 1) einige der Verfahren Informationen in zentralen Verzeichnissen ansammeln und 2) Nachrichten häufig per Multicast versandt werden.

Zentrale Verzeichnisse verwalten Dienstangebote und beantworten Suchanfragen. Protokolle, die zentrale Verzeichnisse benutzen, skalieren unter Umständen besser, da sie ein Wachstum an Anfragen und Angeboten einfach zum Beispiel durch Einrichten zusätzlicher Verzeichnisse bedienen können (siehe Jini, SLP, Salutation). Allerdings stellen sie auch einen Single Point of Failure dar, denn ist ein Verzeichnis im Netzwerk nicht mehr verfügbar, leidet darunter die Funktionalität des Verfahrens oder der Austausch von Informationen kommt gar ganz zum Erliegen.

Zentrale Verzeichnisse widersprechen allerdings dem Konzept von mobilen ad-hoc Netzwerken, da diese Netzwerke sich unter anderem dadurch auszeichnen, dass sie auf keiner festen Infrastruktur beruhen und Teilnehmer jederzeit und unvorhersehbar aus dem Netzwerk ausscheiden können [Barbeau 2000]. Außerdem beruht der Nachrichtenversand auf *Multi Hop Routing*, das heißt Nachrichten passieren unter Umständen mehrere Zwischenstationen bevor sie von einem Endpunkt zum anderen gelangen, was nicht unbedingt dem Grundgedanken von zentralen Verzeichnissen entspricht [Tyan und Mahmoud 2005]. Ein weiterer Nachteil ist, dass in den betrachteten Verfahren die Verzeichnisse statisch ernannt werden, also ein Administrator ein oder mehrere Netzwerkteilnehmer entsprechend konfiguriert. Hat man es jedoch mit einer dynamischen Netzwerktopologie zu tun, so müssten die Verzeichnisse dynamisch zugewiesen und ausgewählt werden, was zu einem hohen Verwaltungsmehraufwand führt [Tyan und Mahmoud 2005].

Verzichtet ein Verfahren auf zentrale Verzeichnisse, so gibt es hier im Wesentlichen drei verschiedene Ansätze um Anfragen und Angebote im Netzwerk zu verteilen: 1) Flooding, 2) Hash-basierte Ansätze und 3) semantisches Routing [Tyan und Mahmoud 2005]. UPnP greift hierbei auf das Flooding, also das Überschwemmen des Netzwerkes mit Nachrichten, zurück und JXTA bietet neben (de)zentralen Verzeichnissen außerdem einen Hash-basierten Ansatz zum Auffinden von Advertisements.

Der Versand von Multicast-Nachrichten bietet sich in mobilen ad-hoc Netzwerken nicht an, da zum einen viele Routing-Protokolle für ad-hoc Netzwerke kein Multicast Routing unterstützen, zum anderen der Versand von Multicast-Nachrichten ein signifikantes, netzweites Nachrichtenaufkommen erzeugt, welches unter Umständen - gerade bei Netzwerkverbindungen mobiler Geräte - unerwünscht ist [Friday u. a. 2004].

Das Verfahren, was unter allen Verfahren am besten für den Einsatz in ad-hoc Netzwerken geeignet erscheint, ist JXTA. Dieses Verfahren kommt ohne zentrale Instanzen aus. Ein Gerät was am JXTA-Netzwerk teilnehmen will, muss lediglich ein beliebiges anderes Gerät als Einstiegspunkt in das Netzwerk finden und ist hierfür nicht zwangsläufig auf Multicast-Anfragen angewiesen. Wurde ein Gerät gefunden, bekommt der neue Teilnehmer von diesem eine Liste mit Rendezvous Peers, die er kontaktieren und fortan für die Suche und das Anbieten von Diensten nutzen kann. Ist in einem Netzwerk noch keine logische JXTA-Infrastruktur vorhanden, zum Beispiel wenn zwei Geräte nur unter sich ein ad-hoc Netzwerk bilden, können Dienstangebote und -anfragen auch ohne diese Rendezvous Peers direkt miteinander ausgetauscht werden. Außerdem kann ein Gerät zu jedem Zeitpunkt entscheiden, selbst ein Rendezvous Peer zu werden und dies den anderen Geräten des Netzwerkes mitteilen.

Auch hinsichtlich der Interoperabilität scheint JXTA geeignet zu sein, da zum einen der Quelltext öffentlich zur Verfügung steht und somit auch die Protokolle und Mechanismen offen liegen und beliebig angepasst werden können, zum anderen aber auch weil JXTA nicht nur in der ohnehin plattformunabhängigen Programmiersprache Java implementiert ist, sondern bereits auch in andere Sprachen, zum Beispiel C/C++, Java 2 Micro Edition etc., portiert wurde. Die Nachrichten, die im JXTA-Netzwerk verschickt werden, sind darüber hinaus alle im XML-Format codiert, so dass sie prinzipiell auch von anderen Verfahren adaptiert werden können. Die Unterstützung von Lease-Mechanismen ist ein weiterer positiver Beitrag hinsichtlich des Einsatzes in mobilen ad-hoc Netzwerken. Auch wenn diese Mechanismen sehr simpel gehalten sind, erlauben sie dennoch eine Selbstheilung des Systems über die Zeit, da die Beschreibungen von Ressourcen, die nicht mehr verfügbar sind, im Laufe der Zeit aus den Verzeichnissen gelöscht werden. Einziger Nachteil hierbei ist, dass die Lease-Zeiten sehr großzügig ausgelegt sind und in der Standardversion von JXTA keine Möglichkeit besteht, die Lease-Zeiten manuell anzupassen, sodass die Aktualität der Informationen in einem ad-hoc Netzwerk darunter leiden könnte [Bisignano u. a. 2003].

Ein weiterer Nachteil von JXTA ist der Umgang mit Hardware-Ressourcen. Die Protokolle sind primär für den Einsatz in größeren Netzwerken ausgelegt und erzeugen zum Teil ein signifikantes Datenaufkommen. Dieses Problem wurde seitens der JXTA-Gemeinde bereits erkannt und mit der Einführung der Rendezvous Super Peers wurde versucht, durch Errichtung eines Rückgrats die Geräte in der Peripherie des Netzwerkes zu entlasten. Hinzu kommt, dass JXTA nicht mehr als eine Netzwerkschnittstelle und nicht mehr als eine Adresse pro Netzwerkschnittstelle zur gleichen Zeit unterstützt. Das bedeutet, dass ein Peer zum Beispiel nicht als Brücke zwischen zwei unterschiedlich angebundenen Geräten (z.B. ein Gerät im IP-basierten Netzwerk und ein Bluetooth-Gerät) vermitteln und nur Teilnehmer in einem Netzwerk zur Zeit sein kann. Außerdem unterstützt JXTA keine Änderung der Adressen zur Laufzeit, was bei mobilen ad-hoc Netzwerken auf Grund instabiler Netzwerkverbindungen durchaus häufiger vorkommen kann [Bisignano u. a. 2003].

Diese Unzulänglichkeiten der Verfahren motivierten Entwickler dazu, Änderungen oder Erweiterungen an den Verfahren vorzunehmen bzw. neue Verfahren zu entwickeln, die spezifisch auf die Anforderungen in mobilen ad-hoc Netzwerken zugeschnitten sind. Eine Reihe solcher Verfahren werden im folgenden Abschnitt 2.4 vorgestellt und diskutiert.

2.4 Discovery in mobilen ad-hoc Netzwerken

Wie in der Diskussion des vorangehenden Abschnitts erläutert wurde, müssen Discovery-Protokolle für mobile ad-hoc Netzwerke weit größeren Anforderungen genügen, als die Protokolle für Infrastrukturnetzwerke. Hierzu gehören insbesondere der sparsame Umgang mit Ressourcen wie Speicher-, Rechen- und Übertragungskapazitäten sowie mit Energie, aber auch das Berücksichtigen der speziellen ad-hoc Charakteristiken, wie der dynamischen Netzwerktopologie oder den Besonderheiten der Routing-Protokolle.

Eine Vielzahl grundverschiedener Lösungsansätze sind aus diesen Anforderungen erwachsen und wesentlich größer ist die Zahl der konkret entworfenen Protokolle, die die zugrunde liegenden Ideen und Konzepte nahezu beliebig kombinieren. Auch adressieren die Verfahren teils jeweils unterschiedliche Arten von ad-hoc Netzwerken und bieten unterschiedliche Funktionalitäten. Im Folgenden werden eine Reihe von Discovery-Verfahren für ad-hoc Netzwerke vorgestellt, deren Auswahl sich lediglich nach den jeweiligen Ansätzen richtet und einen Überblick über die Konzepte geben soll. Im Anschluss daran folgt eine Diskussion der Protokolle. Der objektive Vergleich ist weniger einfach als bei den Verfahren für Infrastrukturnetzwerke, da der Fokus der einzelnen Verfahren auf teils unterschiedlichen Ausprägungen von ad-hoc Netzwerken liegt. Dennoch soll mit der Diskussion letztendlich die Grundlage für eine Bewertung der Verfahren geschaffen werden.

2.4.1 Nutzung von Routing-Informationen

Die hohe Dynamik und fehlende Infrastruktur, welche ad-hoc Netzwerke charakterisieren, erfordern spezielle Routing-Algorithmen (z.B. *AMRoute* [Xie u. a. 2002], *ODMRP* [Lee u. a. 2002], *DSR* [Johnson und Maltz 1996], *AODV* [Chakeres und Belding-Royer 2004]), um Nachrichtenpakete von einem Netzwerkknoten zu einem oder mehreren anderen zu senden. Für gewöhnlich agiert jeder Netzwerkknoten eines solchen Netzwerkes als *Router* und führt entsprechend eigene *Routing-Tabellen*, die bei Änderung der Netzwerktopologie dynamisch angepasst werden müssen [Marin-Perianu u. a. 2005]. Die Anpassung der Routing-Tabellen erfolgt durch den Versand von Nachrichten mit entsprechenden Kontrollinformationen.

Einige Service Discovery-Verfahren hängen an diese Kontrollpakete zusätzliche Informationen über Dienste an, mittels derer Dienste angeboten und gesucht werden können. Auf diese Weise kann die Menge der zu übertragenden Informationen gering gehalten werden, da die Routing-Kontrollpakete ohnehin versandt werden müssen. Allerdings ist es hierzu notwendig, dass die Metainformationen der versendeten und empfangenen Netzwerkpakete (die sog. Paket-*Header*) eingesehen bzw. verändert werden können. Daher bezeichnet man derartige Verfahren auch als *Cross-Layer Service Discovery*, da sie auf verschiedenen Schichten des ISO/OSI-Modells arbeiten [Marin-Perianu u. a. 2005].

Verfahren von Wu und Zitterbart

Das Verfahren von Wu und Zitterbart [Wu und Zitterbart 2001] nutzt die Kontrollpakete des *Dynamic Source Routing* Verfahrens (DSR) [8] [Johnson und Maltz 1996] um sowohl proaktiv als auch reaktiv Dienstinformationen im Netzwerk zu verbreiten. Tritt ein Knoten dem Netzwerk das erste Mal bei oder bietet er einen neuen Dienst an, so sendet er eine komplette Dienstbeschreibung als eigenständige Nachricht an seine direkten Nachbarn. Diese speichern die Beschreibung in

[8] die Autoren weisen allerdings daraufhin, dass auch andere Routing Verfahren eingesetzt werden können

einem lokalen Verzeichnis und leiten sie an ihre direkten Nachbarn weiter, welche bis zu einer vordefinierten Anzahl an Weiterleitungen genauso verfahren. Sucht ein Knoten einen bestimmten Dienst, so sendet er eine Suchanfrage als eigenständige Nachricht an seine Nachbarknoten. Diese überprüfen, ob in ihrem lokalen Diensteverzeichnis ein entsprechender Dienst verzeichnet ist. Falls dem so ist, antworten sie mit der passenden Dienstbeschreibung, andernfalls leiten sie die Anfrage wiederum an ihre direkten Nachbarn weiter. Wurde schließlich ein Dienst gefunden, auf den die Anfrage passt, so sendet der entsprechende Knoten die Beschreibung des Dienstes zurück an den Anfragenden, wobei jeder Knoten, der diese Antwort auf dem Weg zum Anfragenden weiterleitet, einen entsprechenden Eintrag in seinem Diensteverzeichnis vornimmt.

Um den Status eines Dienstes über das Netzwerk hinweg auf allen Knoten zu aktualisieren, werden die Routing-Kontrollpakete verwendet. Dies kann auf drei verschiedene Arten geschehen: 1) implizite Dienstbestätigung, 2) Empfangen einer *Service Poll*-Antwort oder 3) Empfangen einer *No Service*-Anzeige.

Implizite Dienstbestätigung basiert darauf, jedes Nachrichtenpaket, welches einen Knoten passiert, auf eine enthaltene Dienstanbieter-Adresse hin zu überprüfen. Ist eine solche Adresse enthalten, wird implizit davon ausgegangen, dass der Dienst weiterhin zu erreichen ist. Dieses Verfahren schließt auch die Routing-Kontrollnachrichten mit ein. Enthält zum Beispiel eine Kontrollnachricht die Information, dass der Pfad zu einem Knoten unterbrochen ist, so werden auch alle Dienstbeschreibungen, die den Knoten als (ehemaligen) Anbieter ausweisen, entsprechend aktualisiert.

Empfangen einer *Service Poll*-Antwort Einer solchen Antwort geht eine *Service Poll*-Anfrage voraus. Mittels einer solchen Anfrage kann ein Knoten den aktuellen Zustand eines Dienstes erfragen. Für diese Anfrage werden die Metainformationen von IP-Datenpaketen verwendet. Hierzu wird bei einem zufällig ausgewählten IP-Paket ein entsprechendes Optionsfeld gesetzt, welches von dem Empfänger des Paketes ausgewertet wird. Der Empfänger versendet daraufhin seine aktuellen Dienstinformationen in einem sogenannten *Service Awareness*-Paket, in welchem wiederum ein entsprechendes Optionsfeld anzeigt, dass es sich bei diesem Paket um eine *Service Poll*-Antwort handelt. Jeder Knoten, der dieses Paket auf dem Weg zum eigentlichen Adressaten weiterleitet, wertet dieses Optionsfeld aus und aktualisiert sein lokales Verzeichnis. Außerdem durchsucht er sein Verzeichnis, ob er andere Knoten kennt, die den gleichen Dienst anbieten, und hängt deren Adressen gegebenenfalls an die Service Awareness-Nachricht dran.

Empfangen einer No Service-Anzeige Empfängt ein Knoten eine Nachricht, die für einen von ihm angebotenen Dienst bestimmt ist und ist dieser Dienst nicht mehr länger verfügbar, so versendet er an den anfragenden Knoten eine entsprechende Nachricht.

Dieses Verfahren entkoppelt die Dienstmobilität von der Mobilität der Netzwerkknoten. Die Verantwortung für den Umgang mit der Mobilität der Netzwerkknoten wird bei entsprechenden Routing-Protokollen belassen. Die ausgetauschten Routing-Nachrichten werden hingegen leicht modifiziert, um Statusinformationen eines Dienstes in den Diensteverzeichnissen der Knoten zu aktualisieren. Dies erfordert jedoch, dass die Metainformationen der Netzwerkpakete gelesen und geändert werden können [Wu und Zitterbart 2001]. Entsprechende Zugriffsmechanismen unterscheiden sich jedoch zwischen verschiedenen Betriebssystemen und nicht jede Programmiersprache (z.B. Java) erlaubt den Zugriff auf diese Informationen.

Verfahren von Cheng

Wie bereits in Abschnitt 2.3 festgestellt wurde, basieren Service Discovery-Verfahren in lokalen Infrastrukturnetzwerken vielfach auf dem Versand von Multicast-Nachrichten. Erwähnt wurde außerdem, dass sich derartige Verfahren nicht für mobile ad-hoc Netzwerke eignen, da zum einen nur wenige ad-hoc Netzwerke - abhängig vom verwendeten Routing-Protokoll - überhaupt den Versand von Multicast-Nachrichten unterstützen und zum anderen das Überfluten eines Netzwerkes mit einer Nachricht von einigen Netzwerkknoten, insbesondere Knoten mit geringer Bandbreite, unter Umständen nicht erwünscht ist [Friday u. a. 2004]. Dennoch gibt es ad-hoc Netzwerke, in denen der Versand von Multicast-Nachrichten unterstützt und erwünscht ist (z.B. für das Verteilen von Multimediainhalten) und entsprechend wurden hierfür auch spezielle Routing-Protokolle entwickelt (z.B. das *On-Demand Multicast Routing Protocol* oder *AMRoute*).

Cheng stellt in [Cheng 2002, Cheng und Marsic 2000] ein Verfahren vor, welches mit Hilfe von bestimmten Multicast Routing-Paketen das Auffinden und Anbieten von Diensten in mobilen ad-hoc Netzwerken unterstützt. Anders als bei Wu und Zitterbart, welche auf allgemeinen Routing-Protokollen aufsetzen, verlangt dieses Verfahren spezielle Multicast Routing-Protokolle. Aus Gründen der Effektivität und Effizienz wurde das *On-Demand Multicast Routing Protocol* (ODMRP) unter einigen anderen ausgewählt.

ODMRP verlangt das periodische Senden bestimmter Kontrollpakete sobald ein Gerät einer Multicast-Gruppe beitritt [Lee u. a. 2002]. Mit Hilfe dieser Kontrollpakete machen die Geräte ihre Mitgliedschaft in einer Gruppe kenntlich und aktualisieren die Routing-Pfade untereinander. Bietet ein Gerät einen oder mehrere Dienste an, so versendet es eine kurze Dienstbeschreibung als Anhang eines dieser Multicast-Kontrollpakete, dem sogenannten *QUERY JOIN*, an die gesamte Multicast-Gruppe. Um kenntlich zu machen, dass das Kontrollpaket weitere Informationen mit sich trägt, wird ein speziell reserviertes Feld in den Metainformationen des Paketes gesetzt und von jedem Empfänger der Multicast-Gruppe ausgewertet.

ODMRP sieht als Antwort auf jedes empfangene QUERY JOIN-Paket ein *JOIN REPLY*-Paket vor, welches an den ursprünglichen Absender des QUERY JOIN-Pakets zurückgesandt wird. Enthält ein QUERY JOIN-Paket nun eine Dienstbeschreibung, so speichert der Empfänger diese in seinem lokalen Diensteverzeichnis und präpariert ein JOIN REPLY-Paket, welches er durch Setzen eines Feldes in den Metainformationen außerdem als *Service Awareness*-Antwort auf das Dienstangebot kennzeichnet.

Sobald mindestens ein Knoten mit einem Service Awareness-Paket geantwortet hat, werden fortan - bei Änderung der Dienstbeschreibung - Statusinformationen des Dienstes zusammen mit QUERY JOIN-Paketen an die Multicast-Gruppe geschickt. Andernfalls wartet der anbietende Knoten auf explizite Dienstanfragen von Mitgliedern der Gruppe. Eine Dienstanfrage wird, genau wie ein Dienstangebot, an ein QUERY JOIN-Paket angehängt. Die Anfrage enthält Schlüsselwörter oder eindeutige Dienst-IDs, die von jedem Empfänger der Multicast-Gruppe ausgewertet und im Diensteverzeichnis gesucht werden. Ist eine Suche erfolgreich, so wartet der Knoten einen zufälligen Zeitraum auf Antworten anderer Knoten. Ist in diesem Zeitraum keine Antwort empfangen worden, so verschickt er selbst eine Dienstbeschreibung an die gesamte Multicast-Gruppe, sodass alle Mitglieder diese erhalten.

Dieses Verfahren ist, verglichen mit den Discovery-Verfahren für Infrastrukturnetzwerke, ein sehr simples Verfahren. Es unterstützt keine zentralen Verzeichnisse, keine Lease-Konzepte, beschränkt die Reichweite von Multicast-Nachrichten nicht und stützt sich bei der Beschreibung von Diensten auf einfache Dienst-IDs und Schlüsselwörter. Aber es zeigt, genau wie das Ver-

fahren von Wu und Zitterbart, die grundlegenden Möglichkeiten der weitergehenden Nutzung von Kontrollpaketen, insbesondere in ad-hoc Netzwerken, in denen ohnehin seitens der Routing Protokolle der Versand von Multicast-Nachrichten unterstützt wird.

2.4.2 Bluetooth SDP

Die Bluetooth-Spezifikation [BluetoothGroup 2006] beschreibt, wie Bluetooth-Geräte einander in ihrer unmittelbaren Umgebung finden (engl. *Device Discovery*) und spezifiziert darüber hinaus ein Protokoll für das Auffinden spezifischer Dienste, die von anderen Geräten angeboten werden. Aufgrund der Besonderheiten des Bluetooth-Funkstandards haben Bluetooth-Netzwerke (sogenannte *PicoNets*) eine sehr einfache Struktur (siehe Abbildung 2.10). Ein Gerät des Netzwerkes fungiert als *Master* und bis zu 7 sogenannte *Slaves* können aktive Verbindungen zu diesem Master aufbauen. Insgesamt können jedoch bis zu 260 Geräte an dem Netzwerk teilnehmen, wobei sich die übrigen Geräte, die nicht aktiv mit dem Master verbunden sind, in einem passiven Wartezustand befinden und bei Bedarf aktiviert werden können. Es ist auch möglich, dass ein Gerät an mehreren Piconetzen gleichzeitig teilnimmt, in diesem Fall bezeichnet man die Gesamtheit der untereinander verbundenen Piconetze als *ScatterNet*.

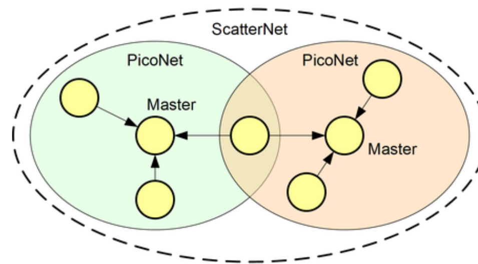

Abbildung 2.10: Bluetooth: Topologie eines Bluetooth ScatterNets (nach [Alterovitz 2001])

Sämtliche Kommunikation zwischen den Geräten eines Piconetzes läuft über den Master als Verbindungsglied. Dieser empfängt Nachrichten und leitet diejenigen, die an andere Geräte adressiert sind, entsprechend weiter. Die Bluetooth-Spezifikation sieht allerdings kein spezielles Nachrichten-Routing zwischen verschiedenen Piconetzen (also innerhalb eines ScatterNets) vor. Hierfür müssen andere Verfahren (z.B. *Routing Vector Method*, RVM [Bhagwat und Segall 1999]) verwendet werden.

Für die Suchen nach Diensten wurde das *Service Discovery Protocol* (SDP) spezifiziert. Dies ist ein sehr einfaches Protokoll, welches speziell für Piconetze und ad-hoc Kommunikation entwickelt wurde. Es unterstützt nur das *Pull*-Modell bei der Suche nach Diensten, nicht hingegen das aktive Anbieten von Diensten. Möchte ein Gerät einen bestimmten Dienst in Anspruch nehmen, sendet es eine Dienstanfrage an das andere Gerät. Diese Dienstanfrage enthält einen Service-Typ und eine Reihe von Service-Attributen mit der ein oder mehrere Dienste passend identifiziert werden können. Empfängt ein Gerät eine solche Anfrage, so durchsucht es seine sogenannten *Service Records* im lokalen Diensteverzeichnis nach passenden Einträgen und antwortet gegebenenfalls mit einem entsprechenden Datensatz.

Die besonderen Vor- und Nachteile des Service Discovery-Protokolls liegen in seiner Einfachheit. Service Records sind einfach zu erstellen und lassen sich darüber hinaus auch mit weiteren beschreibenden Attributen versehen. Außerdem ist das Suchen von Service Records leicht zu implementieren. Der Nachteil hingegen ist, dass das SDP nicht für größere Netzwerke mit mehreren

Teilnehmern gedacht ist, da bei der Suche nach einem bestimmten Dienst das Diensteverzeichnis jedes einzelnen Gerätes durchsucht werden muss [Frank u. a. 2004]. Außerdem ist es nicht möglich, komplexe Anfragen an das Diensteverzeichnis zu stellen, so dass bei der Suche nach mehreren Diensten auch mehrere Anfragen gestellt werden müssen. In [Avancha u. a. 2002] werden Erweiterungen für das SDP beschrieben, die das Beschreiben von Diensten und damit auch die Suche mächtiger machen sollen.

2.4.3 Konark

Konark ist ein Service Discovery-Protokoll, welches speziell für den Einsatz in mobilen ad-hoc Netzwerken entwickelt wurde. Es basiert auf einem Peer to Peer-Modell, in dem jedes teilnehmende Gerät die Fähigkeit besitzt, anderen Geräten lokale Dienste anzubieten sowie das Netzwerk nach angebotenen Diensten zu durchsuchen und diese in Anspruch zu nehmen [Helal u. a. 2003]. Die Anforderungen an das Protokoll erwuchsen aus den speziellen Eigenheiten mobiler ad-hoc Netzwerke. So wurden unter anderem folgende Punkte beim Protokollentwurf berücksichtigt: 1) die Dynamik mobiler ad-hoc Netzwerke, 2) die Frage wo und wie Dienstbeschreibungen gespeichert werden sollen, 3) das Format von Dienstbeschreibungen, 4) die Verteilung bzw. das Anbieten von Dienstbeschreibungen, 5) die Art und Weise wie Dienste in Anspruch genommen werden und 6) die Interaktion mit dem Benutzer.

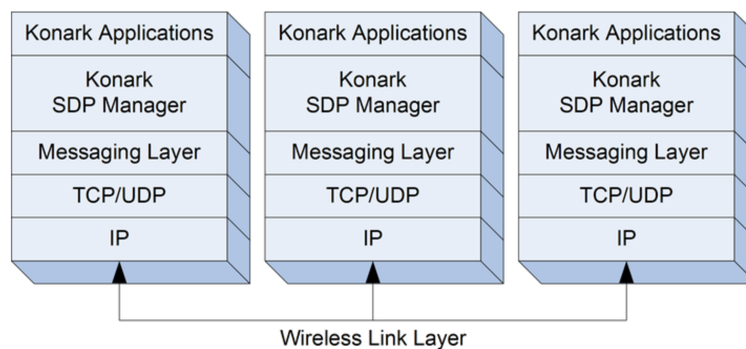

Abbildung 2.11: Konark: Der Konark Discovery Stack (nach [Helal u. a. 2003])

In der Architektur des Konark-Netzwerkes (dargestellt in Abbildung 2.11) spielen die sogenannten *SDP Manager* die Hauptrolle. Jedes Gerät verfügt über einen solchen Manager. Er ist für das Auffinden benötigter Dienste im Netzwerk zuständig, verwaltet ein lokales Diensteverzeichnis und kann Dienste im Netzwerk anbieten. In der Konark-Architektur sitzt der SDP Manager zwischen der Anwendungs- und der Nachrichtentransportschicht. Die Anwendungsschicht enthält - gegebenenfalls neben weiteren Konark-Anwendungen - eine Applikation mittels derer ein Benutzer mit dem SDP Manager direkt interagieren kann. Die Interaktionsmöglichkeiten umfassen das Initiieren und Kontrollieren des Anbietens, Suchens und Ausführens von Diensten.

Das Diensteverzeichnis des SDP Managers ist entsprechend einer Baumstruktur aufgebaut. Diese ist in Abbildung 2.12 dargestellt. Ellipsen kennzeichnen allgemeine Diensttypen, Rechtecke repräsentieren konkrete Dienste. In dieser Baumstruktur werden bekannte Geräte und Dienste anhand von Pfadbeschreibungen gespeichert. Eine Pfadbeschreibung ist eine individuell vergebene Klassifikation eines Dienstes. Für einen Drucker zum Beispiel könnte ein Pfad folgendermaßen Aussehen: *Dienst->Computer->Drucker->Farbdrucker*. Die Verwendung einer Baumstruktur hat zum einen den Vorteil, dass ein Benutzer der oben erwähnten Konark-Applikation intuitiv die

registrierten Dienste anhand der Struktur durchsuchen kann, zum anderen lassen sich über eine derartige Baumstruktur allgemein gehaltene Dienst-Anfragen an das Netzwerk stellen. So kann zum Beispiel einfach nach allen angebotenen Diensten gesucht werden (wenn bei einer Anfrage nur der *Dienst*-Pfad angegeben wurde) oder nach allen vorhandenen Druckern (wenn bei einer Anfrage der *Dienst->Computer->Drucker*-Pfad angegeben wurde).

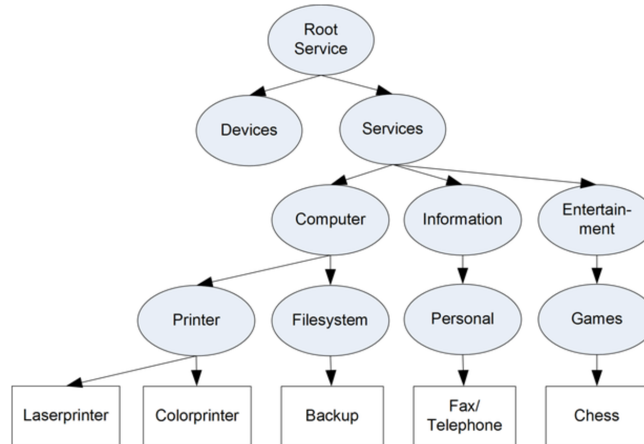

Abbildung 2.12: Konark: Baumstruktur des Diensteverzeichnis (nach [Helal u. a. 2003])

Um Dienste im Netzwerk aufzufinden bzw. anzubieten, unterstützt Konark zwei verschiedene Mechanismen: 1) *active pull* und 2) *passive push*. Bei der Verwendung von *active pull* verschickt ein Client, der einen Dienst sucht, eine Multicast-Anfrage an das Netzwerk. Diese Anfrage enthält ein oder mehrere Schlüsselwörter oder Pfadangaben, die den gewünschten Dienst beschreiben. SDP-Manager, die diese Anfrage erhalten, durchsuchen ihr lokales Diensteverzeichnis entsprechend den angegebenen Pfaden bzw. vergleichen die enthaltenen Dienstbeschreibungen mit den angegebenen Schlüsselwörtern. Bei erfolgreicher Suche antworten sie mit einer passenden Dienstbeschreibung, welche den Dienstnamen, den Diensttyp, eine URL und einen *Time To Live*-Zähler enthält. Die URL zeigt hierbei auf eine detailliertere Dienstbeschreibung (ähnlich wie bei UPnP) und der TTL-Zähler gibt die zu erwartende Bereitschaft des Dienstes in Minuten an.

Auch bei der Verwendung des *passive push*-Mechanismus kann ein Benutzer eine Reihe von Pfadangaben vorgeben. Die Dienstbeschreibungen aller Dienste, die über diese Pfadangaben im lokalen Diensteverzeichnis erreichbar sind, werden dann im Netzwerk angeboten. Hierzu versendet der SDP-Manager die Dienstbeschreibungen der jeweiligen Dienste periodisch mittels Multicast-Nachrichten an andere Geräte im Netzwerk. Empfängt ein Client eine solche Dienstbeschreibung, so fügt er diese entsprechend dem angegebenen Pfad in sein lokales Diensteverzeichnis ein. Möchte ein Client einen Dienst in Anspruch nehmen oder einfach genaue Informationen über einen Dienst erhalten, wird der SDP Manager beauftragt, eine detaillierte Beschreibung des Dienstes anzufordern. Dies entspricht dem Verfahren, welches auch bei UPnP zum Einsatz kommt (vgl. Abschnitt 2.3.4). Über die URL, welche in der Dienstbeschreibung angegeben ist, holt sich der SDP Manager ein XML-Dokument, welches die genauen Eigenschaften und Funktionen eines Dienstes enthält. Die Beschreibungssprache, welche in dem XML-Dokument Verwendung findet, ist an die *Web Service Description Language* [9] (WSDL) angelehnt. Über das *Simple Object Access Protocol* (SOAP, [W3C 2003]) kann ein Benutzer oder eine Konark-Anwendung schließlich den gewünschten Dienst aufrufen.

[9] für weitere Informationen siehe http://www.w3.org/TR/2006/CR-wsdl20-20060327/

Die Konark-Architektur ist unabhängig von einem bestimmten Betriebssystem oder einer bestimmten Programmiersprache, da sie lediglich ein Rahmenwerk liefert, welches Dienste sowie deren Ausführung in IP-basierten Netzwerken beschreibt. Für diese Beschreibungen greift Konark auf das XML-Format und die WSDL- und SOAP-Standards zurück, was die Interoperabilität mit anderen Verfahren prinzipiell ermöglicht [Helal u. a. 2003].

2.4.4 DEAPspace

DEAPspace ist ein Rahmenwerk, welches von *IBM Research, Zurich Lab.* entwickelt wurde [Nidd 2001, Hermann u. a. 2000]. Es besteht aus einem Discovery-Algorithmus und einem Beschreibungsmodell für Dienste. Diese beiden Komponenten sorgen dafür, dass Geräte mit Dienstbeschreibungen von anderen Geräten in ihrer unmittelbaren Umgebung versorgt werden. Die Umgebung bezieht sich hierbei auf die maximale Funkreichweite eines Gerätes. Versendete Nachrichten sollen nicht von anderen Geräten weitergeleitet werden, sodass kein Routing-Verfahren benötigt wird (man spricht auch von *Single Hop*-Nachrichtenversand).

Der Discovery-Algorithmus arbeitet dezentral und basiert auf dem Versand von Broadcast-Nachrichten. In dem von diesem Algorithmus anvisierten Netzwerk ist das Versenden von Broadcast-Nachrichten an sich kein Problem, da es sich um ein sehr kleines Netzwerk mit wenigen Teilnehmern handelt und die Broadcast-Nachrichten ohnehin von den Knoten nicht weitergeleitet werden. Das besondere Problem, welches zu berücksichtigen ist, ist der gleichzeitige Versand von Broadcast-Nachrichten durch zwei oder mehr Geräte, da dies zu Störungen in der Nachrichtenübertragung führen kann.

Die Suche nach Diensten soll - gemäß den Anforderungen des Protokolls - sehr schnell von Statten gehen. Daher haben sich die Entwickler gegen eine *pull*-orientierte Lösung entschieden, bei der Dienste durch Versenden einer Anfrage an das Netzwerk gesucht werden. Stattdessen kommt ein *push*-Modell zum Einsatz, bei welchem die Geräte aktiv Beschreibungen aller ihnen bekannten Dienste (eigene und von anderen angebotene) an alle anderen Geräte versenden.

Um Energie zu sparen und sich bei dem gleichzeitigen Versand von Nachrichten nicht gegenseitig zu stören, konkurrieren die Geräte um dedizierte, fest vordefinierte Versendeintervalle (engl. *advertisement slot*). Innerhalb dieses Zeitraums darf lediglich ein Gerät eine Broadcast-Nachricht mit allen bekannten Dienstbeschreibungen versenden. Das Auflösen der Konkurrenzsituation besteht hierbei lediglich aus dem Warten einer zufälligen Zeitspanne, bevor eine eigene Nachricht versendet werden darf. Wird während dieser Zeitspanne eine Nachricht empfangen, so gilt der Konkurrenzkampf als verloren und beginnt mit dem erneuten Warten wieder von vorne.

Dieser Ansatz erhöht zwar die Menge der zu übertragenden Daten in jeder Nachricht, verringert aber die Häufigkeit des Sendens von Nachrichten. In einer Gruppe von n Geräten lässt sich somit die Übertragungsfrequenz um den Faktor n verringern, indem die Größe der Nachricht um denselben Faktor erhöht wird.

Die Informationen einer empfangenen Nachricht werden von jedem Gerät ausgewertet, mit der eigenen Sicht auf die Umwelt verglichen und gegebenenfalls die eigene Sicht aktualisiert. Stellt ein Gerät dabei fest, dass die empfangene Weltsicht falsche oder nicht mehr aktuelle Informationen über das Gerät selbst hat, so darf das Gerät bei dem nächsten Wettstreit um einen freien Advertisement Slot ein bisschen weniger lang warten, um die Chance zu erhöhen als Gewinner die eigene, aktualisierte Weltsicht verbreiten zu dürfen. Kommt ein Gerät zum ersten Mal in die Sendereichweite eines ad-hoc Netzwerkes, so darf es als Neuankömmling die eigenen Dienste im nächsten Slot sofort bewerben.

2.4.5 Scalable Service Discovery for MANET

Das *Scalable Service Discovery Protocol*, entwickelt von [Sailhan und Issarny 2005], adressiert, anders als die bisher vorgestellten Protokolle, mobile ad-hoc Netzwerke mit hundert und mehr Teilnehmern. Außerdem berücksichtigt es die Möglichkeit des Überbrückens (engl. *bridging*) von Netzwerken, seien es ad-hoc oder Infrastrukturnetzwerke. Entgegen der weitläufigen Meinung, zentrale Verzeichnisse widersprächen dem Konzept von mobilen ad-hoc Netzwerken [Nidd 2001, Barbeau 2000], sieht die Architektur dieses Protokolls zentrale Verzeichnisse als Schlüsselkomponenten vor, da diese - wie bereits in Abschnitt 2.3.7 erwähnt - grundlegend für eine gute Skalierbarkeit sind.

Das Protokoll beruht auf der dynamischen und homogenen Verteilung dieser zentralen kooperierenden Diensteverzeichnisse innerhalb des Netzwerkes. Jedes dieser Verzeichnisse zeichnet sich für eine Menge von Knoten verantwortlich, die in geringer Nachrichtendistanz (gemäß dem sogenannten *Hop Count*, welcher von der Dichteverteilung der Knoten abhängig ist) zu erreichen sind. Kern des Netzwerkes ist - ähnlich wie bei dem in Abschnitt 2.3.6 besprochenen JXTA-Protokoll - ein Rückgrat (engl. *backbone*), welches die Diensteverzeichnisse untereinander verbindet, sodass durch diese Verknüpfung letztlich ein global verteiltes Gesamtverzeichnis entsteht. Ist einem Knoten kein Verzeichnis zugewiesen und hat er in einem zeitlichen Rahmen keine Nachricht über das Vorhandensein eines für ihn zuständigen Verzeichnisses erhalten, so initiiert er in einem kleinen Umkreis (zwei bis drei Hops) ein Auswahl- oder Bewerbungsverfahren, um ein neues Verzeichnis zu bestimmen. Durch dieses freiwillige Auswahlverfahren werden bevorzugt leistungsstarke und gut verbundene Knoten als Verzeichnis bestimmt. Ausschließlich diese Verzeichnisse halten Dienstbeschreibungen vor; andere Knoten brauchen diese daher nicht in lokalen Verzeichnissen zwischenzuspeichern.

Untereinander tauschen die Verzeichnisse Profile aus. Diese Profile beschreiben zum einen die Leistung des Knoten, zum anderen enthalten sie aber auch eine kurze, charakteristische Zusammenfassung des Verzeichnisinhaltes. Diese auf *Hashing* basierende Zusammenfassung wird durch sogenannte *Bloom Filter* [Bloom 1970] erzeugt. Kann ein Verzeichnis eine Suchanfrage eines Knotens nicht beantworten, so leitet es diese Anfrage an eine Menge von Verzeichnissen weiter, deren Bloom-Zusammenfassung darauf schließen lässt, dass diese eventuell die gesuchte Dienstbeschreibung vorhalten. Neben der Verwendung von Bloom-Filtern zur Reduktion des Übertragungsvolumens zwischen einzelnen Verzeichnissen, sieht das Protokoll einen weiteren Mechanismus - das *2 Hop Bordercasting* - vor, um die Skalierbarkeit zu erhöhen. Bordercasting wird analog zum Broad- oder Multicasting in Infrastrukturnetzwerken eingesetzt, um Diensteverzeichnisse im Netzwerk bekannt zu machen. Durch geschickte Auswahl der Empfänger wird hierbei von Station zu Station versucht, eine Nachricht flächendeckend zu verteilen, ohne dass ein Empfänger dieselbe Nachricht mehr als einmal erhält.

Ein Großteil der ad-hoc Netzwerke heutzutage sind hybride Netzwerke [Sailhan und Issarny 2005]. Dies bedeutet, dass ein Teil des Netzwerkes ad-hoc Charakteristiken aufweist, der andere Teil hingegen ein Infrastrukturnetzwerk ist. Um diesem Umstand gerecht zu werden, sieht das Protokoll sogenannte *Gateway*-Verzeichnisse vor, die zwischen den unterschiedlichen Teilen des Netzwerkes vermitteln. Dabei ist es durchaus vorstellbar, dass auf der infrastrukturbasierten Seite des Netzwerkes andere als das vorgestellte Scalable Service Discovery Protocol Verwendung finden.

2.4.6 CARD

Die *Contact-based Architecture for Resource Discovery* (CARD) ist ein Verfahren, welches noch größere ad-hoc Netzwerke (mit zehntausenden Teilnehmern) adressiert als das im vorangegangenen Abschnitt erläuterte Scalable Service Discovery-Verfahren [Helmy u. a. 2005, Nahata u. a. 2002]. Die Motivation für dieses Verfahren liegt in der Tatsache, dass andere Discovery-Protokolle für mobile ad-hoc Netzwerke entweder auf globalem Multi- oder Broadcasting (engl. *global flooding*) beruhen oder auf einer komplexen Hierarchie basieren. Beide Ansätze sind - den Autoren zufolge - ungeeignet für derartig große ad-hoc Netzwerke, da das globale Überfluten mit Nachrichten nicht skaliert und der Aufbau und Erhalt einer Hierarchie komplexe Koordination zwischen den Knoten voraussetzt, welche aufgrund der Mobilität der Knoten nur mit großem Nachrichtenaufwand zu bewältigen ist.

Die CARD-Architektur basiert auf der *Small World*-Hypothese [10] und versucht über sogenannte *Contacts* den durchschnittlichen Grad der Separation zwischen Teilen des Netzwerkes zu reduzieren. Diese Reduktion soll letztendlich eine globale Suche nach Ressourcen ermöglichen, aber mit weniger Nachrichtenaufwand als zum Beispiel bei einer Multi- oder Broadcast-Suche. Jeder Knoten hat eine unmittelbare Nachbarschaft (typischerweise 3-5 *Hops*). Diese Nachbarschaft vergrößert sich jedoch durch Hinzunahme von Kontakten (typischerweise 5-15). Ein Kontakt ist ein Knoten außerhalb der lokalen Nachbarschaft, der sich bereit erklärt hat, bei der Suche nach Ressourcen zu helfen. Durch diese Kontakte wird die Sicht eines Knotens auf das gesamte Netzwerk vergrößert (siehe Abbildung 2.13).

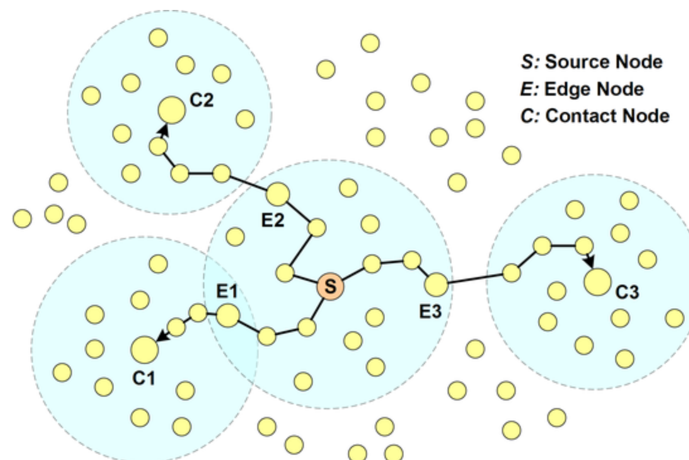

Abbildung 2.13: CARD: Netzwerksicht eines mit Kontakten verbundenen Knotens (nach [Helmy u. a. 2005])

Des größtmöglichen Nutzens wegen sollten sich die Nachbarschaften der Kontakte eines Knotens nicht überlappen, daher verwendet das CARD-Verfahren einen besonderen Algorithmus (veranschaulicht in Abbildung 2.14), der dieses Problem umgehen soll. Hierbei sendet ein Knoten *S* eine Kontaktauswahl-Anfrage zu einem Knoten *a*, der am Rande seiner Nachbarschaft (also 3-5 Hops entfernt) liegt. Diese Anfrage enthält den eindeutigen Bezeichner von *S* (*id*), einen Hop-Zähler (*h*), eine Liste mit den Bezeichnern aller Randknoten (*e-list*) und eine Liste mit den Bezeichnern aller bisherigen Kontakte (*c-list*) von *S*. Der Knoten *a* wählt zufällig einen Nachbarn von sich aus (der nicht in *e-list* enthalten ist) und leitet die Anfrage von *S* an diesen Knoten (*e*)

[10] Die Hypothese von Stanley Milgram besagt, dass alle Menschen auf der Welt über eine kurze Kette an Bekanntschaften miteinander in Beziehung stehen.

weiter. Der Knoten *e* verfährt genauso und das Weiterleiten der Anfrage geschieht solange, bis die maximale Hop-Distanz *h* erreicht wurde oder bis ein Empfänger der Anfrage feststellt, dass kein Knoten seiner Nachbarschaft in der *e-list* oder *c-list* enthalten ist. Dieser Knoten entscheidet dann, ob er als Kontakt für *S* fungieren möchte und antwortet gegebenenfalls.

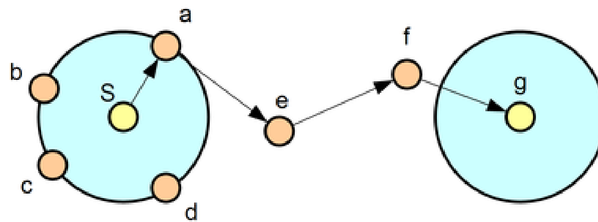

Abbildung 2.14: CARD: Kontaktauswahl-Algorithmus (nach [Nahata u. a. 2002, Poster])

Um die Erreichbarkeit der Kontakte fortlaufend zu überprüfen, werden von jedem Knoten periodisch Validierungs-Anfragen an die Kontakte verschickt. Diese Anfragen enthalten neben dem Bezeichner des Empfängers auch die Bezeichner aller Knoten auf dem Pfad zu dem Empfänger. Jeder Knoten auf diesem Pfad überprüft beim Empfang einer solchen Validierungs-Nachricht, ob der nächste Pfadknoten ein direkter Nachbar ist. Falls dem nicht so ist, muss der Pfad aktualisiert werden, indem versucht wird, die Folgeknoten auf der Pfadliste zu erreichen. Scheitert auch dies, gilt der Kontakt als verloren. Andernfalls wird die Validierungs-Nachricht weitergeleitet bis sie bei dem Kontakt selbst ankommt und bestätigt wird. Das zusätzliche Nachrichtenaufkommen, was durch diese Validierungs-Anfragen entsteht, soll im Laufe der Zeit abnehmen, wie eine Evaluation gezeigt hat, da sich vermehrt Kontakte zwischen *stabilen* Knoten bilden.

Wenn ein Knoten nach einer bestimmten Ressource sucht, stellt er zuerst eine Anfrage an die Knoten in seiner lokalen Nachbarschaft. Ist diese Suche nicht erfolgreich, so verschickt er die Anfrage an seine Kontakte, welche wiederum in ihrer lokalen Nachbarschaft suchen und gegebenenfalls die Anfrage wieder weiter zu ihren Kontakten leiten, solange, bis eine vorgegebene Anfragetiefe erreicht ist. Auf diese Weise wird ein Anfrage-Baum erstellt, welcher die Suche nach Ressourcen skalieren lässt. Das proaktive Anbieten von Ressourcen, nach dem *Push*-Modell, wird von diesem Verfahren nicht unterstützt, da es ein zu hohes Nachrichtenaufkommen verursacht.

2.4.7 Diskussion

Wie bereits bei der Diskussion von Service Discovery-Protokollen für Infrastrukturnetzwerke erläutert (vgl. Abschnitt 2.3.7), werden Discovery-Protokolle für mobile ad-hoc Netzwerke vor eine Reihe weiterer Anforderungen gestellt. So sollen es diese Protokolle ermöglichen, dass ressourcenbeschränkte und kabellos verbundene Geräte dynamisch Dienste in einem spontanen Netzwerk auffinden bzw. anbieten können, wobei einerseits die zu übertragene Datenmenge, die Reaktionszeit, der Energieverbrauch etc. minimiert werden sollen und andererseits berücksichtigt werden muss, dass Geräte unter Umständen nur sporadisch im Netzwerk verfügbar sind. Diese Anforderungen verschärfen sich noch, wenn man zusätzlich größere MANETs mit vielen hundert bis tausend Teilnehmern in Betracht zieht, da in solchen Netzen die Skalierbarkeit der Protokolle von besonderer Bedeutung ist. Außerdem muss bedacht werden, dass die Discovery-Protokolle nach Möglichkeit eine Brücke zwischen mobilen ad-hoc und Infrastrukturnetzwerken schlagen können sollten, um beide Welten zusammenzuführen [Sailhan und Issarny 2005].

Noch weniger als bei den Discovery-Protokollen für Infrastrukturnetzwerke lassen sich die besten und effektivsten Protokolle für mobile ad-hoc Netzwerke bestimmen, da hier je nach Charakteristiken des anvisierten ad-hoc Netzwerkes unterschiedliche Bewertungskriterien angelegt werden. In den vorangegangenen Abschnitten wurde daher versucht, eine Auswahl an Discovery-Verfahren zu beleuchten, die auf unterschiedliche Arten von MANETs zugeschnitten sind. Im Folgenden soll ein Versuch unternommen werden, diese Verfahren hinsichtlich allgemeiner Anforderungen miteinander zu vergleichen.

Einige Vergleichskriterien, die bei den Discovery-Protokollen für Infrastrukturnetzwerke Verwendung fanden, erübrigen sich bei dieser Diskussion. Zum Beispiel die Verwendung von Lease-Konzepten bei Dienstbeschreibungen und -registrierungen. Aufgrund der Dynamik von mobilen ad-hoc Netzwerken muss ein Discovery-Protokoll zwangsläufig derartige Mechanismen vorhalten, andernfalls würde das Netzwerk im Laufe der Zeit aufgrund vieler veralteter Einträge zusammenbrechen. Daher sehen alle oben angesprochenen Verfahren (mit Ausnahme des Bluetooth SDP) entsprechende Mechanismen vor. Unterschiede liegen lediglich im Detail. So verlangen Konark und DEAPspace zum Beispiel das regelmäßige Aktualisieren von Dienstbeschreibungen seitens der Anbieter. CARD hingegen lässt die Knoten (lediglich die Nachfrager) periodisch Nachrichten an ihre Kontakte verschicken, um die Verfügbarkeit zu überprüfen und um die Routen durch das Netzwerk zu aktualisieren.

Ein weiterer Punkt, der bei der Diskussion nicht weiter berücksichtigt werden soll, ist die Awareness-Unterstützung, also das Benachrichtigen bei Eintreten bestimmter Ereignisse. Keines der untersuchen Verfahren bietet ein ausgefeiltes System der zeitnahen Ereignisbenachrichtigung, zum Beispiel beim Registrieren eines neuen Dienstes oder der Nichtverfügbarkeit eines ehemals vorhandenen Dienstes. Derartige Informationen verbreiten sich zwar innerhalb des ad-hoc Netzwerkes, aber kein System sieht einen speziellen *publish/subscribe*-Mechanismus wie etwa bei Jini oder UPnP's Generic Event Notification Architecture vor. Der Grund hierfür könnte darin liegen, dass das Erscheinen oder Austreten eines Knotens aus dem Netzwerk ohnehin von den Protokollen berücksichtigt werden muss und sich diese Information im Laufe der Zeit über andere Mechanismen (z.B. Auswerten der Routing-Information) im Netzwerk verbreitet. Eine explizite, zeitnahe Benachrichtigung würde die Anzahl zu übertragender Nachrichten weiter erhöhen und widerspräche dem Grundprinzip der Discovery-Verfahren für mobile ad-hoc Netzwerke.

Grundlegende Methoden

Die vorgestellten Verfahren versuchen das Anbieten und Auffinden von Diensten über teils grundverschiedene Ansätze anzugehen. Die Verfahren von Wu und Zitterbart sowie von Cheng verwenden die Informationen innerhalb von Routing-Kontrollpaketen für das Anbieten, Suchen und Aktualisieren von Dienstbeschreibungen. Diese Methode zeichnet sich zwar durch ein geringeres Datenübertragungsvolumen allein für das Discovery-Protokoll aus [Wu und Zitterbart 2001, Cheng 2002], dies wird allerdings von der Menge zu übertragender Kontroll- und Steuerinformationen seitens der Routing-Protokolle teils wieder egalisiert. In Netzwerken, in denen hingegen ohnehin Multi Hop oder Multicast Routing-Protokolle eingesetzt werden, bietet sich diese Methode durchaus an.

Eine weitere Methode ist das Etablieren von sogenannten *Overlay Networks*. Diese virtuellen Netzwerke basieren auf dem eigentlichen physikalischen Netzwerk und stellen einzelnen Geräten virtuelle Verbindungen zu anderen, weiter entfernten Geräten, zur Verfügung. Die so miteinander verbundenen Geräte fungieren in einem begrenzten Umkreis als zentrale Verzeichnis-

se und tauschen untereinander Dienstbeschreibungen aus, um Informationen in unterschiedlichen Teilen des Netzwerkes zugänglich zu machen. Das Scalable Service Discovery-Protokoll sowie CARD sind zwei Vertreter dieser Art. Auf diese Weise lässt sich das Überfluten des Netzwerkes mit Discovery-Paketen vermeiden und doch ist es möglich global, ggf. sogar netzwerkübergreifend (mittels *Gateways*), nach Diensten zu suchen. Der Nachteil ist, dass neben den eigentlichen Dienstbeschreibungen und Suchanfragen auch noch eine Reihe von Kontroll- und Steuerinformationen für die komplexe Koordination zwischen den Geräten ausgetauscht werden müssen, damit das Overlay-Netzwerk auch bei Ausfall mehrerer Knoten intakt bleibt [Nahata u. a. 2002, Helmy u. a. 2005].

Das Konark-Netzwerk versucht nicht, wie andere Verfahren, eine Hierarchie oder spezielle virtuelle Topologie aufzubauen und aufrecht zu erhalten, stattdessen beruht es auf einem reinen Peer to Peer-Ansatz. Jedes Gerät hat die Möglichkeit, mittels Multicast-Nachrichten Dienste zu suchen und anzubieten. Mitglieder bestimmter Multicast-Gruppen speichern einmal aufgefundene Beschreibungen und stellen sie bei Bedarf anderen Mitgliedern zur Verfügung [Helal u. a. 2003].

Das Service Discovery-Protokoll des Bluetooth-Standards hingegen unterstützt lediglich das passive Anbieten von Diensten. Interessenten müssen die Dienstbeschreibungen der gewünschten Dienste bei jedem Gerät innerhalb des Single Hop-Piconetzwerkes aktiv erfragen. Es gibt kein zentrales Diensteverzeichnis, welches alle im Netzwerk angebotenen Dienstbeschreibungen zwischenspeichert [BluetoothGroup 2006]. Ähnlich verhält es sich auch im DEAPspace-Netzwerk. Auch hier gibt es kein zentrales Verzeichnis, stattdessen speichert jedes Gerät eine komplette Sicht auf alle verfügbaren Dienste im Netzwerk und versendet diese Weltsicht zur Aktualisierung in zufälligen dedizierten Zeitintervallen an alle in Übertragungsreichweite liegenden Geräte. Zwar ist hierbei das Übertragungsvolumen recht hoch, dafür aber die Häufigkeit der Übertragungen gering. Dies erlaubt es den Geräten zwischenzeitlich die Sendeleistung zu reduzieren um Energie zu sparen [Nidd 2001].

Anforderungen

Mit den unterschiedlichen Herangehensweisen bei der Suche und dem Anbieten von Diensten gehen auch unterschiedliche Anforderungen an die zugrunde liegende Netzwerkinfrastruktur sowie die Gerätesoftware einher. Bei der Entscheidung für oder gegen ein bestimmtes Verfahren stellen diese Anforderungen harte Ausschlusskriterien dar. Einige Verfahren, unter anderem die von Wu und Zitterbart sowie Cheng, sind auf bestimmte Routing-Protokolle festgelegt, in deren Kontrollpakete sie weiterführende Informationen zu Diensten unterbringen. Neben der Tatsache, dass selbst einfache Routing-Protokolle in ad-hoc Netzwerken nicht unbedingt selbstverständlich sind (z.B. in Bluetooth- oder DEAPspace-Netzen), ist auch die Forderung nach einem speziellen Routing-Protokoll eine starke Einschränkung der Interoperabilität[11], sowohl mit anderen Geräten als auch mit anderen Netzwerken. Zusätzlich müssen die Programmiersprachen, in denen die Protokolle implementiert werden, den Zugriff auf die Metainformationen der Nachrichtenpakete (engl. *packet header*) von der Anwendungsschicht aus erlauben, was zum Beispiel den Einsatz der Java-Programmiersprache nahezu ausschließt.

Auch die Forderung eines Multicast Routing-Protokolls, wie es im Protokoll von Cheng sowie bei Konark verlangt wird, stellt eine nicht unbedeutende Einschränkung dar, da eine derartige Versandart nur von wenigen Protokollen überhaupt unterstützt wird. Wenn es unterstützt

[11] Wikipedia listet zum Beispiel über siebzig verschiedene ad-hoc Routing-Verfahren, unterteilt in neun verschiedene Kategorien (Stand September 2006) [Wikipedia 2006a]

wird, bedeutet dies oft ein erhebliches Nachrichtenaufkommen, einerseits durch die Routing-Kontrollpakete, anderseits durch die Multicast-Nachrichten an sich, was die ohnehin schon ressourcenbeschränkten Geräte noch weiter belastet [Nidd 2001].

Einige Discovery-Verfahren, die in dieser Arbeit allerdings nicht weiter untersucht wurden (z.B. das *Hexell Cluster-Netzwerk* [Tyan und Mahmoud 2005]), verlangen das Vorhandensein von Geoinformationen, die den aktuellen Standort der Geräte in der natürlichen Welt beschreiben. Diese Daten werden häufig sowohl für das Nachrichten-Routing als auch für den Aufbau eines Overlay-Netzwerkes verwendet [Sailhan und Issarny 2005]. Die geringsten Anforderungen unter allen vorgestellten Protokollen stellen das Bluetooth SDP sowie DEAPspace. Beide arbeiten in Single Hop-Netzwerken und sind daher nicht auf besondere Routing-Protokolle angewiesen. Sie benötigen weder zentrale Verzeichnisse noch ein besonderes Overlay-Netzwerk. Allerdings sind beide Verfahren auch nur für sehr kleine Netzwerke mit wenigen Teilnehmern und geografischer Begrenztheit entwickelt worden.

Skalierbarkeit

Die Skalierbarkeit von Discovery-Protokollen für ad-hoc Netzwerke ist heutzutage lediglich von peripherem Interesse, da die Verbreitung der entsprechenden Technologien noch nicht so weit fortgeschritten ist, dass bereits jetzt eine allgegenwärtige Vernetzung möglich wäre. Dennoch bedarf dieser Punkt einer genaueren Betrachtung, da der Mobiltechnologie und den ad-hoc Diensten eine erfolgreiche Zukunft vorausgesagt wird und eines Tages die spontane Vernetzung einer Vielzahl von Geräten unseren Alltag begleiten könnte.

Die bisher am weitesten verbreitete Technologie ist Bluetooth. Eine Vielzahl von Geräten des täglichen Bedarfs, seien es Mobiltelefone, Kopfhörer oder Peripheriegeräte des Computers, sind bereits mit der Funkschnittstelle ausgestattet. Die Bluetooth-Architektur ist allerdings auf eine maximale Netzwerkgröße von ca. 250 Teilnehmern ausgelegt, von denen lediglich acht Teilnehmer gleichzeitig aktiv sein können. Hinzu kommt, dass das Netzwerk eine sternförmige Topologie aufweist und somit das Master-Gerät in der Mitte des Sterns einen Flaschenhals und einen Single Point of Failure darstellt. Aus diesen Gründen ist festzustellen, dass das SDP nur sehr schlecht skaliert und darüber hinaus technologiebedingt eine obere Teilnehmerschranke besitzt.

DEAPspace ist, genau wie Bluetooth, auf einen geografisch kleinen Raum, nämlich die Funkreichweite von Geräten, begrenzt und stellt daher auch keine Ansprüche gut zu skalieren. Es lässt sich lediglich feststellen, dass mit zunehmender Teilnehmerzahl jedes Gerät größere Datenmengen in größeren Abständen verschicken muss. Die insgesamt zu versendende Datenmenge bleibt über die Zeit also konstant, jedoch wächst die zu empfangene und zu verarbeitende Datenmenge linear mit der Anzahl der Teilnehmer. Die Skalierbarkeit der Verfahren von Wu und Zitterbart sowie Cheng dürften wesentlich von der Skalierbarkeit des verwendeten Routing-Protokolls abhängig sein, wobei Cheng's Verfahren aufgrund der Verwendung von Multicast-Nachrichten weitaus schlechter skalieren dürfte. Gleiches gilt auch für das Konark-Verfahren, welches zwar unabhängig vom verwendeten Routing-Protokoll arbeitet, aber auch intensiven Gebrauch von Multicast-Nachrichten macht.

Das Scalable Service Discovery-Protokoll sollte, wie der Name bereits sagt, gut skalieren. Es ist mit seiner hierarchischen Architektur für ein Netzwerk mit mehreren hundert Teilnehmern entwickelt worden. Eine Vielzahl weiterer Konzepte, unter anderem die dynamische Verzeichniszuweisung, das Bordercasting, die Directory Profiles etc. sorgen dafür, dass das erzeugte Nachrichtenaufkommen, sowohl zur Strukturerhaltung als auch für das Suchen und Anbieten von

Diensten, möglichst gering gehalten wird. CARD adressiert sowohl kleine als auch große Netzwerke mit mehreren tausend Teilnehmern. Ähnlich wie das Scalable Service Discovery-Protokoll basiert auch CARD auf einem virtuellen Netzwerk und einem Rückgrat bestehend aus wenigen Knoten, die größere Distanzen (mehrere Hops) zwischen Teilen des Netzwerkes überbrücken und so eine globale Suche nach Diensten ermöglichen. Auch dieses Verfahren sollte gut skalieren können.

Abschließend muss festgestellt werde, dass die Skalierbarkeit der Verfahren lediglich das Potenzial ausdrückt, eine Vielzahl von Geräten in einem Dienste-Netzwerk zu vereinen. Keines der Verfahren wurde in der Praxis in großen ad-hoc Netzwerken bestehend aus hunderten oder tausenden Geräten eingesetzt. Die Aussagen über die Skalierbarkeit rühren entweder von dem Wissen über die Skalierbarkeit allgemeiner Mechanismen (z.B. skaliert Multicasting in der Regel schlecht) oder von Simulationen, die die Entwickler der Verfahren mittels Netzwerksimulatoren unter verschiedenen Parameterkonfigurationen durchgeführt haben (für Simulationsergebnisse siehe unter anderem [Helmy u. a. 2005, Sailhan und Issarny 2005, Nidd 2001, Wu und Zitterbart 2001]).

Reichweite der Entdeckung

Dieser Punkt knüpft eng an den Punkt der Skalierbarkeit an. Grundsätzlich lässt sich sagen, je besser ein Verfahren skaliert, desto größer ist die vorgesehene Reichweite. Da ad-hoc Netzwerke meistens über Funkverbindungen hergestellt werden, ist die Funkreichweite des Senders die wichtigste Beschränkung. Es ist aber natürlich möglich durch Weiterleiten von Nachrichten über eine oder mehrere Zwischenstationen die Reichweite zu vergrößern. Das Bluetooth SDP sowie DEAPspace sind beide nur für Single Hop-Netzwerke ausgelegt, entsprechend ist ihre Reichweite nur auf wenige Geräte in ihrer unmittelbaren Umgebung ausgelegt. Andere Verfahren wie CARD, Konark oder das Scalable Service Discovery-Protokoll hingegen unterstützen bzw. verlangen Multi Hop-Netzwerke, ihre Reichweite ist daher im Prinzip unbestimmt. Aber lediglich das Scalable Service Discovery-Protokoll sieht explizit eine Unterstützung von Gateways zu anderen Arten von Netzwerken vor, sodass dieses Protokoll prinzipiell auch in Weitverkehrsnetzen (z.B. dem Internet) das Suchen und Anbieten von Diensten unterstützt.

Zusammenfassung

Service Discovery-Protokolle für mobile ad-hoc Netzwerke sind vor gänzlich andere Anforderungen gestellt als Protokolle für Infrastrukturnetzwerke. Einerseits müssen sie die Dynamik eines solchen Netzwerkes, in dem Geräte unvorhersehbar beitreten und wieder ausscheiden können, berücksichtigen, andererseits müssen sie äußerst sparsam mit den Ressourcen (Übertragungs-, Rechen- und Speicherkapazität sowie Energie) der Geräte umgehen. Diese Anforderungen führten zu einer Vielzahl verschiedener Herangehensweisen und daraus resultierender Protokolle. Sieben Verfahren wurden insgesamt vorgestellt und näher beleuchtet. Jedes dieser Verfahren folgt einem anderen Ansatz und ist auf eine bestimmte Art von ad-hoc Netzwerken ausgerichtet. Aufgrund der hohen Diversifizierung lässt sich nicht abschließend ein optimales oder bestes Protokoll ausmachen. Vielmehr gilt es für bestimmte Anwendungszwecke deren Anforderungen und die herrschenden Gegebenheiten genau zu analysieren, um sich dann für ein passendes Verfahren zu entscheiden oder der Interoperabilität wegen eine abstrakte Architektur zu modellieren, die eine Vielzahl unterschiedlicher Verfahren unterstützt [Marin-Perianu u. a. 2005].

2.5 Repräsentation von Ressourcen

Wie bereits beschrieben, müssen die Entitäten, um ein Bewusstsein für ihre Umwelt zu erlangen, Informationen über ihnen bekannte Ressourcen untereinander austauschen um so eine möglichst globale Sicht zu erlangen. In den Abschnitten 2.3 und 2.4 wurden hierfür verschiedene Verfahren vorgestellt, mittels derer die Suche und der Austausch von Informationen, zum einen in Infrastruktur- zum anderen in mobilen ad-hoc Netzwerken, ermöglicht wird. Dieser Austausch setzt voraus, dass die Informationen in einer Form vorliegen, die einerseits von den beteiligten Parteien verstanden wird, und andererseits derart kodiert werden kann, dass man sie über ein Netzwerk verschicken kann. Einige dieser Verfahren bedienen sich hierfür sehr simpler, proprietärer Repräsentationen, andere wiederum setzen auf ausdrucksstarke, standardisierte Beschreibungssprachen. In [Chen u. a. 2001] werden folgende allgemeine Mängel bzw. Probleme der von vielen Service Discovery-Verfahren (u.a. SLP, Jini, UPnP, Salutation etc.) verwendeten Repräsentationen für Dienste bzw. Ressourcen aufgeführt:

Mangel an reichhaltigen Repräsentationen Dienste unterscheiden sich in den von ihnen definierten Funktionalitäten und Fähigkeiten. Anhand der Beschreibungen dieser Funktionalitäten und Fähigkeiten werden passende Dienste gesucht, ausgewählt und in Anspruch genommen. Hierbei mangelt es allerdings vielfach an ausdrucksstarken Beschreibungssprachen, die ausreichend mächtig sind, eine Vielzahl unterschiedlicher Dienste einheitlich zu beschreiben und Schlussfolgerungen aus den Beschreibungen zu ziehen.

Mangel an Beschränkungsangaben und ungenaues Matching Die meisten Verfahren führen einen Abgleich (engl. *matching*) von Suchanfragen und angebotenen Dienstbeschreibungen lediglich auf syntaktischer Ebene durch. Hierbei werden exakte Übereinstimmungen zwischen den Angaben in einer Suchanfrage und den Beschreibungen der Dienste gesucht. Die Suche nach einem S/W-Drucker zum Beispiel würde keinen Farbdrucker als Ergebnis liefern. Auch ist es häufig nicht möglich Beschränkungsangaben (engl. *constraints*) in den Suchanfragen anzugeben. Zum Beispiel könnte man bei einem Auktionsdienst nach Auktionen suchen wollen, die mindestens 3 und maximal 10 Angebote haben.

Mangel an Unterstützung von Ontologien Menschen und in höherem Maße auch Maschinen müssen sich beim Austausch von Informationen auf eine einheitliche Semantik einigen, damit sich diesen der Sinn von Aussagen erschließt. Wohldefinierte Ontologien können zu einem gemeinsamen Verständnis von Aussagen führen. Dies ist umso wichtiger, sobald Informationen nicht nur über ein Verfahren ausgetauscht, sondern mittels verschiedener Verfahren in heterogenen Umgebungen verbreitet werden.

Unter anderem aus diesen Unzulänglichkeiten lassen sich eine Reihe von Anforderungen ableiten, die an Beschreibungssprachen für Ressourcen gestellt werden. Allen voran sollen die Beschreibungen möglichst sowohl von Menschen als auch von Maschinen erstellt, gelesen und manipuliert werden können. Da diese Beschreibungen zwischen Maschinen in einem Netzwerk ausgetauscht werden sollen, müssen sie darüber hinaus serialisierbar sein. Die Beschreibungen sollten so mächtig wie nötig sein, also möglichst viele notwendige Details erfassen können, ggf. auch implizite Informationen aus expliziten Informationen mittels eines Inferenzmechanismus ableiten können, dabei aber auch gerade so simpel sein, dass sie einfach in kurzer Zeit ohne besondere Kenntnisse (gegebenenfalls unter Zuhilfenahme von Werkzeugen) von Menschen erstellt werden können.

Eine zusätzliche Anforderung wäre neben der reinen syntaktischen Beschreibung auch eine semantische Beschreibung. Dies ist vor allem für Inferenzen notwendig, um aus richtigen Aussagen keine falschen Schlüsse zu ziehen. Das oben genannte Beispiel der Druckersuche könnte zum Beispiel zum Erfolg gebracht werden, wenn Ontologien eingesetzt würden, die das Konzept eines Farbdruckers mit der zusätzlichen Eigenschaft schwarz drucken zu können anreichern. Nebenbei führt der Austausch von semantisch beschriebenen Informationen auch zu besserer Interoperabilität mit anderen Verfahren, da automatische Übersetzungen zwischen verschiedenen Beschreibungssprachen möglich werden.

Im Folgenden werden nun verschiedene Möglichkeiten der Repräsentation vorgestellt, unter anderem werden hierbei die ausdrucksstarken Beschreibungssprachen RDF und RDF-Schema, DAML+OIL, OWL sowie WSDL betrachtet. Am Ende des Abschnittes findet sich eine Zusammenfassung, die den einzelnen Verfahren die oben erwähnten Anforderungen gegenüberstellt.

2.5.1 Resource Description Framework

Das *Resource Description Framework* (RDF) wurde vom *World Wide Web Consortium* (W3C) entwickelt, um Informationen über Ressourcen und Beziehungen zwischen Ressourcen in einem standardisierten Format zu repräsentieren. Als Ressourcen werden hierbei Objekte verstanden, die jedwede Art von Information enthalten können. Daher bezeichnet man die mittels RDF kodierten Informationen auch als *Metadaten*, da sie Informationen über Informationen beschreiben [W3C 2004e, W3C 2004d, Tauberer 2006, Schön 1998].

Der Begriff des Rahmenwerkes (engl. *framework*) deutet schon an, dass es sich nicht um ein starres Format zur Beschreibung handelt, sondern um ein flexibles und offenes Modell, welches nach Belieben erweitert werden kann. Damit der Austausch von Informationen zwischen unterschiedlichen Anwendungen funktioniert, liegt diesem offenen Modell das RDF-Datenmodell zugrunde, welches unabhängig von einer bestimmten Serialisierungssyntax ist [Schön 1998].

Das RDF-Datenmodell versucht Entitäten sowie Abhängigkeiten zwischen Entitäten, so wie sie gemeinhin durch Graphen beschrieben werden, in eine einfachere, sowohl für Menschen als auch für Maschinen lesbare Form zu bringen. Hierzu werden Informationen über die Entitäten als sogenannte *Triplets* (auch als *Statements* bezeichnet) dargestellt. Diese Triplets beschreiben eine Subjekt-Prädikat-Objekt-Beziehung, z.B. (Winter, ist_eine, Jahreszeit). Sie stellen also eine Beziehung (das Prädikat) zwischen einem Subjekt und einem Objekt dar, wobei das Objekt selbst wiederum ein solches Triplet sein kann und somit eine Aussage über eine Aussage getroffen werden kann (auch als *Reifikation* bezeichnet). Eine Menge von Triplets bilden einen gerichteten RDF-Graphen, der komplexe Zusammenhänge zwischen den einzelnen Triplets beschreibt. Ein solcher Graph bildet somit eine logische Konjunktion über alle Aussagen aller in ihm enthaltenen Triplets [W3C 2004d].

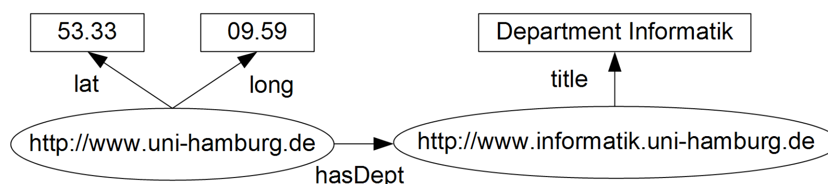

Abbildung 2.15: RDF-Graph gehörend zum Beispiel aus Listing 2.1 (eigene Darstellung)

Bezeichnet werden Subjekte, Prädikate und Objekte in RDF über sogenannte *Uniform Resource Identifier* (URI) und Literale, wobei Literale lediglich Objekten zugewiesen werden dürfen. Die URIs stellen global eindeutige Bezeichner dar, die es erlauben, die Gesamtheit aller möglichen Namen in definierte Einheiten zu unterteilen, so wie die *Uniform Resource Locator* (URL) im Internet. Enthalten verschiedene Dokumente dieselben URIs, so kann man davon ausgehen, dass sie sich ebenfalls auf dieselbe Ressource beziehen. Auf diese Weise lassen sich verteilte Wissensbeschreibungen von Ressourcen einfach zusammenführen. Da URIs unter Umständen recht lang sein können, werden sie üblicherweise unter Verwendung des Konzeptes der Namensräume (engl. *namespaces*) abgekürzt [W3C 2004d, Tauberer 2006].

Um RDF-Beschreibungen sowohl für Maschinen als auch für Menschen lesbar zu machen, bedarf es einer festgelegten Syntax. Zwar schreibt das W3C keine Syntax vor, empfiehlt aber das XML-Format zum Austausch von Ressourceninformationen[12]. In Listing 2.1 ist ein Beispiel eines RDF/XML-Dokumentes gegeben, Abbildung 2.15 zeigt den zugehörigen RDF-Graphen und Tabelle 2.2 listet die jeweiligen Triplets zur besseren Veranschaulichung auf.

```
1  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns"
2           xmlns:dc="http://purl.org/dc/elements/1.1/"
3           xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos"
4           xmlns:edu="http://www.example.org/">
5
6     <rdf:Description rdf:about="http://www.uni-hamburg.de">
7         <geo:lat>53.33</geo:lat>
8         <geo:long>09.59</geo:long>
9         <edu:hasDept rdf:resource="http://www.informatik.uni-hamburg.de"
10                     dc:title="Department␣Informatik"/>
11    </rdf:Description>
12
13 </rdf:RDF>
```

Listing 2.1: RDF/XML Beispiel-Dokument (nach [Tauberer 2006])

In diesem Dokument wird die Universität Hamburg sowie deren Department Informatik beschrieben. Natürlichsprachlich spiegelt dieses Dokument folgende Aussage wider: *Die Universität Hamburg liegt auf 53.33 Grad nördlicher Breite und liegt auf 9.59 Grad östlicher Länge und hat ein Department mit dem Titel „Department Informatik“*. Kodiert sind diese Aussagen in zwei verschiedenen Arten von Knoten (Zeilen 6 - 11): *Ressourcenknoten* und *Eigenschaftsknoten*.

Ressourcenknoten stellen die Subjekte und Objekte eines Statements dar. Das Attribut `rdf:about` bezeichnet die URI, also den global eindeutigen Namen der Ressource, welche diese repräsentiert. In Zeile 6 wird ein Ressourcenknoten definiert, der das Subjekt *http://www.uni-hamburg.de* benennt. In den Zeilen 7-10 werden diesem Subjekt Prädikate bzw. Eigenschaftsknoten zugeordnet. In diesem Beispiel gibt es drei Eigenschaftsknoten: `geo:lat`, `geo:long` und `edu:hasDept`. Ihre Werte stellen die Objekte dar. Objekte können entweder Literale, wie zum Beispiel *53.33* oder *Department Informatik*, sein oder eine Referenz darstellen. Referenzen verweisen auf andere Ressourcen als Objekt, kenntlich gemacht durch das Attribut `rdf:resource` (vgl. Zeile 9) [Tauberer 2006].

[12] *Notation 3* (N3) [W3C 1998] ist ein weiteres verbreitetes Format, welches allerdings in dieser Arbeit nicht weiter beschrieben wird

Subjekt	*Prädikat*	*Objekt*
<http://www.uni-hamburg.de>	edu:hasDept	<http://www.informatik.[...].de>
<http://www.uni-hamburg.de>	geo:lat	„53.33“
<http://www.uni-hamburg.de>	geo:long	„09.59“
<http://www.informatik.[...].de>	dc:title	„Department Informatik“

Tabelle 2.2: Subjekte, Prädikate und Objekte des RDF-Beispieldokumentes (nach [Tauberer 2006])

RDF Schema

Das beschriebene Beispiel zeigt, wie man einfache Aussagen über Ressourcen mittels benannter Eigenschaften und Werte verfasst. Es fehlt aber die Möglichkeit, diesen Aussagen ein Vokabular zuzuordnen, damit Aussagen über Ressourcen gleichen Typs mit bestimmten Eigenschaften auch gleiche Bezeichner tragen. Hierfür hat das W3C die *RDF Vocabulary Description Language: RDF Schema* (RDFS) entworfen. Mittels RDF Schema ist es möglich, anwendungsspezifische Klassen und Eigenschaften zu beschreiben. Es bietet allerdings keine vordefinierten Vokabulare sondern unterstützt lediglich beim Erstellen dieser [W3C 2004e, W3C 2004f].

Das von RDF Schema eingeführte Typsystem lässt sich mit dem Typsystem objektorientierter Programmiersprachen (z.B. Java) vergleichen. Zum Beispiel ist es möglich, Ressourcen als Instanzen einer oder mehrerer Klassen auszuweisen. Somit ist definiert, welche Eigenschaftstypen die Ressourcen haben. Außerdem können Klassen in einer Hierarchie organisiert sein, sodass eine Klasse Unterklasse einer anderen Klasse sein kann und deren Eigenschaftstypen erbt.

Alles, was RDF Schema mit sich bringt, wird in Form eines RDF Vokabulars bereitgestellt. Das bedeutet RDF Schema ist eine spezialisierte Menge vordefinierter RDF Ressourcen mit ihrer eigenen besonderen Bedeutung. Die beschriebenen Vokabulare sind ihrerseits gültige RDF-Graphen. Somit kann eine Software, die nicht speziell die Verarbeitung von RDF Schema unterstützt, trotzdem einen gültigen RDF-Graphen erzeugen, wird aber die zusätzliche Semantik von RDF Schema nicht verstehen.

RDF Abfragesprachen

Bisher wurde beschrieben, wie Ressourceninformationen in ein Datenmodell überführt werden und wie dieses Modell zum Austausch zwischen verschiedenen Parteien serialisiert werden kann. Von Interesse ist allerdings auch, wie eine Anwendung auf die Informationen, die im Datenmodell enthalten sind, zugreifen kann. Hierfür verwendet man sogenannte *RDF Abfragesprachen* (engl. *query languages*).

Eine Abfragesprache sollte die wesentlichen Konzepte von RDF berücksichtigen und unterstützen. Hierzu gehören das abstrakte Datenmodell, die formale Semantik und die Möglichkeit der Inferenz (das Ableiten impliziter Informationen aus expliziten Informationen), die Unterstützung von XML-Schema Datentypen sowie die Unterstützung der freien Beschreibung von Ressourcen (wenn jeder Ressourcen beschreiben darf, müssen unvollständige und widersprüchliche Aussagen berücksichtigt werden) [Haase u. a. 2004]. Eine ganze Reihe verschiedener Abfragesprachen wurden seit der Veröffentlichung von RDF entwickelt. Unter ihnen sind z.B. *RQL* [Karvounarakis u. a. 2002], *SeRQL* [Broekstra und Kampman 2004], *RDQL* [Seaborne 2004] und seit April 2006 als Empfehlung des W3C auch *SPARQL* [Prud'hommeaux und Seaborne 2006]. Eine Bewertung der einzelnen Sprachen soll an dieser Stelle nicht vorgenommen werden, hierfür sei auf [Haase u. a. 2004] verwiesen.

```
SELECT
 ?title

WHERE
 (?res, <edu:hasDept>, ?pub),

AND
 ?res eq 'http://www.uni-hamburg.de'

USING
  rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns>,
  dc  for <http://purl.org/dc/elements/1.1/>,
  edu for <http://www.example.org/>
```

Listing 2.2: RDQL-Abfrage

Zur Veranschaulichung der grundlegenden Funktion einer solchen Abfragesprache sei eine RDQL-Abfrage in Listing 2.2 gegeben, die zu dem oben beschriebenen Beispiel die Titel aller Departments der Universität Hamburg aus dem Datenmodell abfragt.

2.5.2 DAML+OIL

Wie bereits erwähnt, ist das Resource Description Framework ein offenes Rahmenwerk, welches nach Belieben erweitert werden kann. Genau dies haben zwei Forschergruppen (anfangs) unabhängig voneinander gemacht. Auf amerikanischer Seite ist dabei die *DARPA Agent Markup Language Ontology* (DAML-ONT) entstanden, die Europäer entwickelten *Ontology Inference Layer* (OIL) [Fensel u. a. 2000]. Die Motivation beider Projekte bestand darin, das relativ einfache und beschränkte RDF Schema durch semantische Beschreibungen (*Ontologien*) noch ausdrucksstärker zu machen. RDF Schema ist zwar schon in gewisser Weise als Ontologiesprache erkennbar: es verwendet Klassen und Eigenschaften, Bereichs- und Domänenbeschränkungen sowie Unterklassen- und Untereigenschaften-Relationen, aber auch nicht mehr, RDF Schema bleibt eine relativ primitive Sprache. Hinzu kommt die Anforderung, automatisch Schlussfolgerungen ziehen zu können, das heißt, die semantischen Beziehungen zwischen syntaktisch unterschiedlichen Termen zu bestimmen.

Um dies zu leisten, haben sich die beiden Projekte DAML-ONT und OIL zu *DAML+OIL* zusammengeschlossen [Horrocks 2002b, Horrocks 2002a, Horrocks u. a. 2002, Ouellet und Ogbuji 2002]. DAML+OIL ist eine Ontologiesprache, die schließlich entworfen wurde, um die Struktur einer Domäne zu beschreiben. Genau wie bei RDF Schema wird ein objektorientierter Ansatz verfolgt, in dem die Struktur durch Klassen und Eigenschaften - sowie bei DAML+OIL außerdem durch Vereinigung, Durchschnitt und Inversion von diesen - beschrieben werden kann. Außerdem gibt es eine Menge von *Axiomen*, mittels derer die Charakteristiken dieser Klassen und Eigenschaften näher bestimmt werden können. Damit ist DAML+OIL gleichwertig mit einer ausdrucksstarken Beschreibungslogik.

Obwohl DAML+OIL bereits in Discovery-Protokollen für mobile ad-hoc Netzwerke (z.B. beim *GSD Discovery Protocol* [Chakraborty u. a. 2002]) eingesetzt wird, soll es an dieser Stelle nicht weiter vertieft werden, da die Weiterentwicklung der Sprache zugunsten der im Folgenden vorgestellten OWL Web Ontology Language eingestellt wurde.

2.5.3 OWL Web Ontology Language

Die *OWL Web Ontology Language* wurde, genau wie auch RDF und RDF Schema, vom *World Wide Web Consortium* (W3C) im Rahmen der *Semantic Web*-Initiative erstellt. Ihr Zweck ist die Erstellung von Ontologien zur Beschreibung von Ressourcen und zum Ableiten neuen Wissens aus vorhandenem Wissen. Im Gegensatz zu Szenarien in denen Informationen dem Menschen lediglich präsentiert werden sollen, wie es im heutigen Internet der Fall ist, soll OWL vor allem dort eingesetzt werden, wo Informationen von Anwendungen bearbeitet werden und die Anwendungen hierfür mit der Semantik der Informationen umgehen müssen.

OWL entstand aus der im vorigen Abschnitt vorgestellten Beschreibungssprache DAML+OIL und ist eng an deren Semantik und Syntax angelehnt. Die Unterschiede zwischen diesen beiden Sprachen sind nur minimal und in [W3C 2004b] kurz aufgeführt. Genau wie DAML+OIL basiert OWL auch auf RDF und RDF Schema und kann als eine Ergänzung dieses Rahmenwerkes angesehen werden, welches neue Möglichkeiten bietet. Das W3C hat in [W3C 2004c] eine Reihe von Fallstudien (engl. *use cases*) entworfen, in denen der Einsatz von RDF und RDF Schema durch deren Beschränktheit nicht möglich bzw. sinnvoll ist. Zu diesen Beschränkungen gehören vor allem [Antoniou und van Harmelen 2003]:

Lokaler Geltungsbereich von Eigenschaften In RDF Schema ist es nicht möglich, den Bereich von Klassen, für die bestimmte Eigenschaften definiert sind, einzugrenzen. Zum Beispiel lässt sich nicht direkt ausdrücken, dass Kühe Pflanzen fressen, andere Tiere aber auch Fleisch.

Disjunktheit von Klassen In RDF Schema gibt es keine Möglichkeit auszudrücken, dass Klassen keine gemeinsamen Elemente besitzen, zum Beispiel die Klassen Frauen und Männer. Man kann lediglich Unterklassenbeziehungen beschreiben, zum Beispiel dass Frauen eine Unterklasse von Personen bilden.

Boolesche Kombinationen von Klassen Es ist häufig einfacher, neue Klassen durch Kombination existierender Klassen unter Zuhilfenahme von Vereinigung, Durchschnitt und Komplement zu beschreiben. So ließe sich zum Beispiel die Klasse Person einfach als die Vereinigung der Klassen Frauen und Männer beschreiben. Dies wird von RDF Schema nicht unterstützt.

Beschränkungen der Kardinalität In manchen Fällen kann es sinnvoll sein, die Anzahl verschiedener Werte, die eine Eigenschaft annehmen kann oder muss, zu beschränken. Zum Beispiel lässt sich in RDF Schema nicht ausdrücken, dass eine Person genau zwei Elternteile hat oder dass ein Kurs von mindestens einem Veranstalter gehalten werden muss.

Besondere Charakteristiken von Eigenschaften Manchmal kann es nützlich sein, eine Eigenschaft zum Beispiel als *transitiv*, *eindeutig* oder *invers* zu einer anderen Eigenschaft auszuweisen. Auch dies wird von RDF nicht unterstützt.

Unter anderem aus diesen Anforderungen hat das W3C mit der Web Ontology Language eine Beschreibungssprache entwickelt, die zwar an RDF Schema angelehnt, aber weitaus mächtiger ist und alle obigen Anforderungen erfüllt. Dabei sollte ein Kompromiss zwischen der Ausdrucksstärke der Sprache und der Forderung nach einem Mechanismus für Schlussfolgerungen (engl. *reasoning*), zur Ableitung neuen Wissens aus bestehendem Wissen, gefunden werden. Dieser Kompromiss resultiert in drei verschiedenen OWL-Sprachebenen, die unterschiedlich ausdrucksstark und deren Schlussfolgerungsmechanismen unterschiedlich effizient sind. Die höchste Ebene

stellt hierbei *OWL Full* dar. Diese Ebene ist sowohl syntaktisch als auch semantisch kompatibel zu RDF und RDF Schema, nutzt alle Sprachprimitive von OWL, bietet dafür aber keine Garantie der Entscheidbarkeit von Schlussfolgerungsprozeduren. Um die Entscheidbarkeit und vor allem die Effizienz der Schlussfolgerungsmechanismen wieder zu erlangen, werden in der nächst tieferen Ebene bei *OWL DL* (DL steht für *Description Logic*, zu deutsch Beschreibungslogik[13]) einige Einschränkungen bezüglich des Sprachgebrauchs von OWL gemacht. Der Nachteil dieser Einschränkungen ist unter anderem der Verlust der vollständigen Kompatibilität zu RDF und RDF Schema. Auf der untersten Ebene steht schließlich *OWL Lite*, welche noch weiter beschränkt ist und zum Beispiel Disjunktionen verbietet und die Kardinalitätsgrenzen auf 0 und 1 setzt. Der große Vorteil dieser Sprachebene liegt in der einfacheren Verständlichkeit für den Benutzer sowie der einfacheren Implementierbarkeit für den Entwickler [Antoniou und van Harmelen 2003, W3C 2004a].

Beispiel

Anstatt ein vollständiges Beispiel einer OWL-Beschreibung zu geben, wird im Folgenden lediglich ein Auszug einer Beschreibung betrachtet, der einerseits die enge Verbindung zu RDF und RDF Schema verdeutlichen und andererseits den erweiterten Sprachumfang von OWL beleuchten soll. Die vollständige Beschreibung dieses Beispiels findet sich in [Antoniou und van Harmelen 2003]. Das Listing 2.3 zeigt einen Auszug der Beschreibung einer Ontologie für Pflanzen. Es werden drei Klassen mit ihren Eigenschaften beschrieben (*plant*, *tree* und *branch*) und eine weitere Klasse (*animals*), die in diesem Auszug nicht aufgeführt ist, referenziert.

```
1  <owl:Class rdf:ID="plant">
2    <owl:disjointWith="#animal"/>
3  </owl:Class>
4
5  <owl:Class rdf:ID="tree">
6    <rdfs:subClassOf rdf:resource="#plant"/>
7  </owl:Class>
8
9  <owl:Class rdf:ID="branch">
10   <rdfs:subClassOf>
11     <owl:Restriction>
12       <owl:onProperty rdf:resource="#is-part-of"/>
13       <owl:allValuesFrom rdf:resource="#tree"/>
14     </owl:Restriction>
15   </rdfs:subClassOf>
16 </owl:Class>
17
18 <owl:TransitiveProperty rdf:ID="is-part-of"/>
```

Listing 2.3: OWL: Auszug einer OWL-Beschreibung (nach [Antoniou und van Harmelen 2003])

[13] Beschreibungslogiken bilden eine Familie logischer Sprachen, die in ihrer Ausdrucksstärke zwischen der Aussagenlogik und der Prädikatenlogik angesiedelt sind

In den Zeilen 1-3 wird die Klasse der `plants` einfach durch eine Disjunktion mit der Klasse der `animals` beschrieben. Das bedeutet, dass die Klasse der Pflanzen und die Klasse der Tiere keine gemeinsamen Elemente haben. Beschreibt man nun eine Klasse Lebewesen als Oberklasse der Pflanzen und Tiere und hat ein Element, welches kein Tier ist, so könnte über einen Schlussfolgerungsmechanismus die Aussage, dass dieses Element in die Klasse der Pflanzen gehöre, abgeleitet werden. Die Zeilen 4-6 beschreiben mit der Klasse `tree` die Klasse der Bäume als Unterklasse der Pflanzen. Zur Beschreibung der Unterklassenrelation wird ein RDF Schema-Ausdruck auf eine RDF-Ressource angewendet. Hier wird die enge Verwandtschaft von OWL zu diesen Beschreibungssprachen besonders deutlich.

Schließlich wird in den Zeilen 7-14 die Klasse `branch` als Unterklasse der Bäume eingeführt, wobei die Zeilen 9-12 an dieser Stelle von besonderer Bedeutung sind. In diesen Zeilen wird eine Einschränkung auf die Eigenschaft `is-part-of` (welche in diesem Auszug nicht näher definiert ist) bestimmt. Durch `owl:allValuesFrom` wird die Klasse der möglichen Werte, die die Eigenschaft is-part-of annehmen kann, spezifiziert oder anders ausgedrückt, alle Werte müssen aus dieser Klasse kommen. Durch die zusätzliche Charakteristik der Transitivität, definiert in Zeile 16, ist in diesem Beispiel also alles, was in einer is-part-of Beziehung zu Bäumen steht, Element der Klasse Äste [Antoniou und van Harmelen 2003].

2.5.4 Web Services Description Language

Das W3C hat im Jahre 2001 die *Web Services Description Language Version 1.1* (WSDL) vorgestellt, eine Beschreibungssprache für sogenannte *Web Services*[14] [W3C 2001]. WSDL definiert eine plattform-, programmiersprachen- und protokollunabhängige XML-Spezifikation zur Beschreibung von Web Services und zum Austausch von Nachrichten zur Interaktion mit diesen. Die Beschreibung umfasst im Wesentlichen Informationen über die von einem Dienst angebotenen Funktionen samt ihrer Eingabe- und Rückgabewerte, die verwendeten Datentypen, den Inhalt der zu kommunizierenden Nachrichten sowie die Adressen und Protokolle mittels derer ein Dienst angesprochen werden kann, zusammengefasst also *was* ein Dienst bietet, *wo* er angeboten wird und *wie* er aufgerufen werden kann.

Sprachelemente

Die Beschreibung basiert auf XML und XML Schema um sowohl von Menschen als auch von Maschinen erzeugt, gelesen und manipuliert werden zu können. Das hat auch zur weiten Akzeptanz von WSDL geführt. Durch ihre Flexibilität und Erweiterbarkeit empfiehlt sie sich - neben der Beschreibung von Web Services - außerdem für weitergehende Einsatzmöglichkeiten. So verwenden zum Beispiel das Discovery-Protokoll von Tyan und Mahmoud [Tyan und Mahmoud 2005], das Scalable Service Discovery-Protokoll [Sailhan und Issarny 2005] und Konark [Helal u. a. 2003] eine WSDL-ähnliche Beschreibungssprache für Dienste. Um einen Dienst mittels WSDL zu beschreiben, muss der Anbieter des Dienstes Angaben zu folgenden Komponenten in einer speziellen, öffentlich zugreifbaren Datei machen [Burden 2005]:

Datentypen Mit Hilfe der Datentypen werden die Daten beschrieben, die die Anwendung für die Ausführung verlangt und die schließlich als Ergebnis zurückgeliefert werden. Dieses können primitive Typen wie Ganzzahlen oder Zeichenketten, aber auch komplexe benutzerdefinierte Typen sein.

[14] Ein Web Service ist eine sich selbst beschreibende, eigenständige, modulare Applikation, die über das Internet zugreifbar ist [Tsalgatidou und Pilioura 2002].

Operationen Über die Operationen werden die konkreten Methoden beschrieben, die der Dienst an seiner öffentlichen Schnittstelle den Klienten anbietet. Die abstrakte Beschreibung der Methoden wird einerseits verwendet, damit die Klienten wissen, über welche Nachrichten sie den Dienst oder einzelne Teildienste aufrufen und Ergebnisse empfangen können und andererseits, um die angebotenen Operationen auf die konkrete Implementation auf Seite des Anbieters abzubilden.

Protokolle WSDL ist zwar eine protokollunabhängige Beschreibungssprache, aber für den Zugriff auf einen Dienst müssen die von diesem Dienst verstandenen Protokolle aufgeführt werden. Eine Protokollbeschreibung enthält hierfür die Art des Protokolls (z.B. *SOAP*, *HTTP-GET* etc.), die Netzwerkadresse des Dienstes, an welche die Nachrichten verschickt werden müssen, die Art des Austauschmechanismus (z.B. One Way, Request/Response etc.) und die Intention der Nachrichten (z.B. *Remote Procedure Call*, Dokumentanforderung etc.).

Bindungen Da eine WSDL-Datei durchaus mehr als einen Dienst beschreiben kann und für einen Dienst verschiedene Operationen, die über unterschiedliche Protokolle aufgerufen werden können, spezifiziert sein können, müssen alle diese Angaben miteinander verknüpft werden. Erst über diese Bindungen lassen sich alle Details eines Dienstes unzweifelhaft feststellen.

Um die konkrete Anwendung einer Beschreibungen zu demonstrieren, soll im nachfolgenden Abschnitt ein kurzes Beispiel gegeben werden. Auf eine detailliertere Beschreibung von WSDL wird verzichtet, da diese Sprache nicht im Fokus der Arbeit steht. Hierfür sei auf weiterführende Literatur verwiesen, zum Beispiel [W3C 2001, Skonnard 2003, Provost 2003, Tsalgatidou und Pilioura 2002].

Beispiel

Im Folgenden soll als Beispiel die Beschreibung eines Web Services entwickelt werden, welcher auf Anfrage die Temperatur an einem vorgegebenen Ort zurückliefert. Wie bereits erwähnt ist WSDL unabhängig von der Programmiersprache, zur Veranschaulichung soll jedoch angenommen werden, der Dienst wäre in der Java-Programmiersprache geschrieben und entsprechend des Listings 2.4 implementiert.

```
1 public float getTemperature(String zipCode, boolean celsius)
2 {
3   [...]
4 }
```

Listing 2.4: WSDL: Java-Quelltext der Beispielmethode (nach [Rotenstreich 2006])

Um nun diese Methode als Web Service anzubieten, muss man eine entsprechende WSDL-Beschreibung erstellen. In Listing 2.5 ist der Rumpf einer WSDL-Datei dargestellt. Das `definitions`-Element (Zeile 1) ist das Wurzelelement der Beschreibung. An dieser Stelle werden die einzelnen Namensräume definiert, die innerhalb der Beschreibung verwendet werden können[15].

[15] In diesem Beispiel wurde nur der WSDL-Namensraum als Standard vorgegeben, alle anderen Namensräume, die im Beispiel Verwendung finden (z.B. der *SOAP*-Namensraum) wurden der Einfachheit halber ausgelassen

Die ersten drei Elemente (*types*, *message* und *porttype*) sind abstrakte Definitionen der Schnittstelle des Web Services. Sie stellen die Verbindung zu der Schnittstelle der Implementation des Web Services dar, spezifizieren also zum Beispiel die Parameter und Rückgabewerte. Die anderen beiden Elemente (*binding* und *service*) beschreiben die konkreten Details der Abbildung der abstrakten Schnittstelle auf die zu versendenden Nachrichten. Die Reihenfolge der Angaben ist unerheblich und kann beliebig gewählt werden [Skonnard 2003].

```
1  <definitions name='weatherservice' xmlns='http://schemas.xmlsoap.org/wsdl/'>
2    <!-- abstract definitions -->
3    <types> ...
4    <message> ...
5    <porttype> ...
6
7    <!-- concrete definitions -->
8    <binding> ...
9    <service name='WeatherService' > ...
10 </definitions>
```

Listing 2.5: WSDL: Rumpf der WSDL-Beschreibung (nach [Skonnard 2003])

Zwischen den `types`-Knoten lassen sich eigene Datentypen in Form von XML Schema-Definitionen angeben, was in diesem Beispiel nicht notwendig ist, da die primitiven Datentypen bereits ausreichen. Anschließend werden die Nachrichten (engl. *messages*) definiert (siehe Listing 2.6), die zwischen Klienten und Anbieter ausgetauscht werden. Eine Nachricht wird durch einen Namen identifiziert und besteht aus einem oder mehreren Teilen (engl. *parts*), welche die Variablennamen und den jeweiligen Datentyp angegeben.

```
1 <message name='Weather.GetTemperature'>
2   <part name='zipCode' type='xsd:string'/>
3   <part name='celsius' type='xsd:boolean'/>
4 </message>
5 <message name='Weather.GetTemperatureResponse'>
6   <part name='Result' type='xsd:float'/>
7 </message>
```

Listing 2.6: WSDL: Definition von Nachrichten (nach [Rotenstreich 2006])

Die Nachrichtendefinitionen alleine enthalten noch keine Angaben darüber, ob es sich um eine Anfragenachricht oder eine Ergebnisnachricht handelt. Für jede öffentlich angebotene Methode oder Funktion muss daher eine sogenannte `operation` definiert werden (siehe Listing 2.7, Zeilen 2-5), die festlegt, welche Nachricht als Anfrage erwartet wird, welche Nachricht schließlich als Antwort zurückgeschickt wird und im Falle mehrfachen Nachrichtenaustausches, in welcher Reihenfolge die Nachrichten ausgetauscht werden sollen. Die Menge aller Operationen, die ein Dienst an einem bestimmten Port anbietet, wird als *Port-Typ*[16] bezeichnet, sodass die Beschreibung der einzelnen Operationen als Unterknoten eines `porttype`-Knotens (Zeile 1) deklariert werden müssen.

```
<porttype name='WeatherSoapPort'>
  <operation name='GetTemperature'>
    <input message='wsdlns:Weather.GetTemperature' />
    <output message='wsdlns:Weather.GetTemperatureResponse' />
  </operation>
</porttype>
```

Listing 2.7: WSDL: Definition von Port-Typen (nach [Rotenstreich 2006])

Im nächsten Schritt wird durch Angabe eines *Bindings* der Übergang von den abstrakten Datentypen, Nachrichten und Operationen hin zu den konkreten Details der Nutzung eines Port-Typen mit einem bestimmten Protokoll geschaffen. Das `binding`-Element ist generisch und muss durch spezielle protokollabhängige Elemente ausgestaltet werden. Wie bereits erwähnt, stützt sich dieses Beispiel auf den Aufruf des Dienstes über SOAP, sodass die folgenden Angaben in Listing 2.8 spezifisch für dieses Protokoll sind. Zum Verständnis der spezifischen Details sei auf weiterführende Literatur (z.B. [W3C 2001, Skonnard 2003]) verwiesen, an dieser Stelle soll nur kurz angedeutet werden, was die Angaben zu bedeuten haben.

```
<binding name='WeatherSoapBinding' type='wsdlns:WeatherSoapPort'>
  <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='GetTemperature'>
    <soap:operation soapAction='http://xyz.org/Weather.GetTemperature'/>
    <input>
      <soap:body use='encoded' namespace='http://tempuri.org/message/'
                 encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </input>
    <output>
      <soap:body use='encoded' namespace='http://tempuri.org/message/'
                 encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </output>
  </operation>
</binding>
```

Listing 2.8: WSDL: Definition von Bindungen (nach [Rotenstreich 2006])

Da SOAP-Nachrichten über verschiedene Protokolle versendet werden können, muss in Zeile 2 ein Transportprotokoll für die Übermittlung der Nachrichten angegeben werden. Für jede WSDL-Operation, die im Vorwege definiert wurde, müssen dann Angaben für bestimmte Felder in den Nachrichtenköpfen gemacht werden (z.B. die Angabe der Adresse über *SOAPAction*) und schließlich für die einzelnen Nachrichten die Kodierung spezifiziert werden, sodass zum Beispiel die Bedeutung der Zeichenketten *true* oder *-1* als boolescher Wert für den formalen Parameter *celsius* der Java-Beispielmethode interpretiert werden kann. Mit der Angabe des `service`-Elementes wird die Beschreibung des Dienstes schließlich komplettiert. Dieses Element enthält eine Menge von `port`-Knoten (auch als *Endpunkte* bezeichnet), die sich auf ein bestimmtes Binding beziehen und die Details der Adressierung für dieses Binding aufführen.

[16] seit der Version WSDL 1.2 wurde der Begriff Port-Typ durch *Interface* ersetzt

```
1 <service ...>
2   <port name='WeatherSoapPort' binding='wsdlns:WeatherSoapBinding'>
3     <soap:address location='http://localhost/weatherservice.asp'/>
4   </port>
5 </service>
```

Listing 2.9: WSDL: Definition von Ports und Nachrichten (nach [Rotenstreich 2006])

Hiermit ist die WSDL-Beschreibung des Web Services abgeschlossen und die Datei, welche die Beschreibung enthält, kann in SOAP Clients für den Zugriff auf den Dienst verwendet werden.

2.5.5 Zusammenfassung

In den vorangegangenen Abschnitten wurden die Unzulänglichkeiten der von Discovery-Verfahren verwendeten Repräsentationen beschrieben sowie ein Katalog von Anforderungen für die Beschreibung von Ressourcen erstellt. Darauf folgend wurde eine Auswahl an gebräuchlichen Beschreibungssprachen für Ressourcen vorgestellt: das Resource Description Framework mit der Erweiterung RDF Schema, DAML+OIL, die OWL Web Ontology Language sowie die Web Services Description Language.

Diesen Möglichkeiten der Beschreibung von Ressourcen bzw. Diensten ist ihre Ausdrucksstärke, verglichen mit einfacheren Beschreibungen, zum Beispiel mittels Schlüssel-Wert-Paaren, gemein. Alle Sprachen basieren auf dem XML-Standard, sind somit serialisierbar und lassen sich flexibel erweitern. Sie sind sowohl von Menschen als auch von Maschinen zu verstehen und zu bearbeiten und alle erlauben eine Anreicherung der Informationen mit zusätzlicher Semantik.

Im Gegensatz zu RDF, DAML+OIL und OWL, welche sich für die allgemeine Beschreibung von Ressourcen aller Art anbieten, ist WSDL jedoch speziell auf die Domäne von Diensten zugeschnitten. Trotz seiner Flexibilität und Erweiterbarkeit bietet WSDL sich nicht für die Beschreibung von allgemeinen Ressourcen an, was zwar durchaus möglich wäre, aber nicht der ursprünglichen Intention entspricht und daher besser von anderen, entsprechend geeigneten Beschreibungssprachen, zum Beispiel RDF oder OWL, unterstützt wird.

2.6 Zusammenfassung

Der Begriff der Awareness bezeichnet allgemein das Gewahrsein einer Entität über Vorgänge und andere Entitäten in ihrer Umwelt. Im Gegensatz zu den Begriffen des Lookup und der Discovery versteht man unter Awareness einen Prozess, der nicht nur in einem bestimmten Zeitraum mit definiertem Ende, sondern kontinuierlich abläuft, sodass das interne Umweltmodell einer Entität stetig aktualisiert und an neue Umweltzustände angepasst wird.

Es existieren eine Reihe verschiedener Arten von Umweltmodellen, beispielsweise die Modelle der natürlichen, der logischen und der sozialen Umwelt. Die einzelnen Modelle unterscheiden sich zum einen vornehmlich durch die Prinzipien und Prozesse, die in ihnen wirken, zum anderen auch durch die Entitäten, die ein Modell umfasst. Allgemein differenziert man hierbei zwischen ziel- und funktionsorientierten Entitäten, wobei erstere im Kontext dieser Arbeit die Agenten darstellen und letztere die Ressourcen (z.B. Dienste) repräsentieren.

Wenn eine Entität ihrer Umwelt gewahr sein möchte, so muss sie über die Vorgänge in ihrer Umwelt bzw. deren Auswirkungen informiert werden, damit sie einen neuen Umweltzustand in ihr internes Umweltmodell integrieren kann. Die Änderung des Umweltzustandes bezeichnet

man auch als Umweltereignis und es wurden unterschiedliche Arten von Ereignissen entsprechend ihren Urhebern identifiziert.

In einer logischen Umwelt müssen Informationen über Ereignisse explizit unter den einzelnen Entitäten in Form von Nachrichten ausgetauscht werden. Hierfür werden Discovery-Verfahren eingesetzt, welche über verschiedenartige Mechanismen den Austausch von Informationen zwischen den Entitäten einer logischen Umwelt regeln und so das Suchen, Auffinden und Überwachen des Zustandes entfernter Ressourcen erlauben. Je nach Entwurfsziel werden diese Verfahren entweder in Infrastruktur- oder in mobilen ad-hoc Netzwerken eingesetzt, wobei jedes dieser Verfahren wiederum bestimmte Aspekte der Informationsübermittlung in den Netzwerken adressiert.

Der letzte Abschnitt dieses Kapitels befasste sich schließlich mit der Repräsentation von Informationen über Entitäten und beleuchtete verschiedene Beschreibungssprachen. Die Entscheidung für oder gegen eine bestimmte Beschreibungssprache hängt insbesondere von deren Einsatzzweck ab, wobei man unter anderem besondere Anforderungen an die allgemeine Verständlichkeit, die Ausdrucksstärke sowie die Verbreitung einer Sprache stellt, damit sichergestellt werden kann, dass sich verschiedenartige Entitäten in heterogenen Umgebungen untereinander verständigen können.

3 Grundlagen der Migration in mobilen Systemen

Binnen der letzten Jahre begannen die Grenzen zwischen Intra- und Inter-Netzwerken zu verschwimmen. Die Netze überspannen sich gegenseitig und damit einher geht ein Wachstum der Größe des Netzwerkes. Ein Nebeneffekt dieses Wachstums ist ein Ansteigen des Datenverkehrs in den Netzen, welcher wiederum eine Leistungssteigerung der Kommunikationsinfrastruktur verlangt. Mit anwachsender Größe und höherer Leistung geht auch eine stärkere Durchdringung (engl. *pervasion*) und Allgegenwärtigkeit (engl. *ubiquity*) der Netze in unserem Alltag einher, was durch die hohe Verbreitung und Akzeptanz mobiler Kommunikationsgeräte noch weiter verstärkt wird.

Die Technologien, Architekturen und Methodologien, die man für den Entwurf und die Entwicklung von verteilten Anwendungen verwendet, gelangen hier an ihre Grenzen. Insbesondere wird nicht der gewünschte Grad an Skalierbarkeit, Konfigurierbarkeit, Anpassbarkeit und Fehlertoleranz erreicht [Fuggetta u. a. 1998, Cabri u. a. 1998, Shiao 2004]. Um diesen Problemen zu begegnen, versucht man etablierte Modelle und Technologien an diese neuen Bedingungen anzupassen, setzt dabei aber weiterhin die alten Architekturen (z.B. Client/Server) voraus. *CORBA* [Object Management Group 1999] ist ein Beispiel hierfür. Einen anderen Ansatz geht das Konzept der Codemobilität, welches für die Fähigkeit steht, die Bindungen zwischen Programmcode und Ausführungsort dynamisch zu ändern [Carzaniga u. a. 1997].

Dieses Kapitel widmet sich der Mobilität, angefangen mit den grundlegenden Abstraktionen, Mechanismen, Technologien und Paradigmen. Einen Schwerpunkt bildet hierbei das Paradigma mobiler Agenten. Es werden die Besonderheiten dieses Entwurfsprinzips mit seinen Vor- und Nachteilen vorgestellt und die Eignung für mobile Systeme untersucht. Insbesondere der umstrittene Punkt der effizienten Migration wird diskutiert und mögliche Optimierungsansätze vorgeschlagen. Schließlich wird mit einer Beschreibung von Mobilitätsmodellen auch das *Kalong-Modell* [Braun und Rossak 2005] vorgestellt, welches das Ergebnis von Analysen der schlechten Migrationsleistung mobiler Agenten darstellt und eine Reihe von Mechanismen vorsieht, um die Migrationseffizienz zu steigern.

3.1 Grundlagen der Mobilität

In den folgenden Abschnitten sollen die Grundlagen der Codemobilität vermittelt werden. Beginnend mit den wesentlichen Abstraktionen, wird eine grundlegende Terminologie aufgebaut. Dann folgt eine Erklärung der Mechanismen, die es einem Programm erlauben, seinen Ausführungsort zu wechseln. Eine kurze Übersicht über ausgewählte Technologien, die Codemobilität bereits ermöglichen, soll die Übertragbarkeit der Konzepte in die Praxis verdeutlichen. Mit den Entwurfsparadigmen werden schließlich mehrere Methodologien für den Entwurf sogenannter Mobile Code-Systeme miteinander verglichen, bevor am Ende mit der Programmiersprache *Java* eine konkrete Technologie mit ihren Möglichkeiten und Schwierigkeiten hinsichtlich der Codemobilität im Detail vorgestellt wird.

3.1.1 Abstraktionen

Existierende Technologien zur Unterstützung mobilen Codes verwenden auf den ersten Blick eine Reihe unterschiedlicher Konzepte und Primitive. Bevor einzelne Technologien näher beleuchtet werden können und ein paar konkrete Beispiele folgen, sollen in dem folgenden Abschnitt daher zuerst einige Abstraktionen eingeführt werden, damit darauf aufbauend die verschiedenen Mechanismen zum Verschieben von Code und Zustand eines Programms betrachtet werden können.

Architekturen verteilter Systeme

Herkömmliche verteilte Systeme weisen eine Struktur auf, bei der den einzelnen Komponenten eine weitgehend transparente Sicht auf das Netzwerk geboten wird (dargestellt in der oberen Hälfte der Abbildung 3.1) [Fuggetta u. a. 1998].

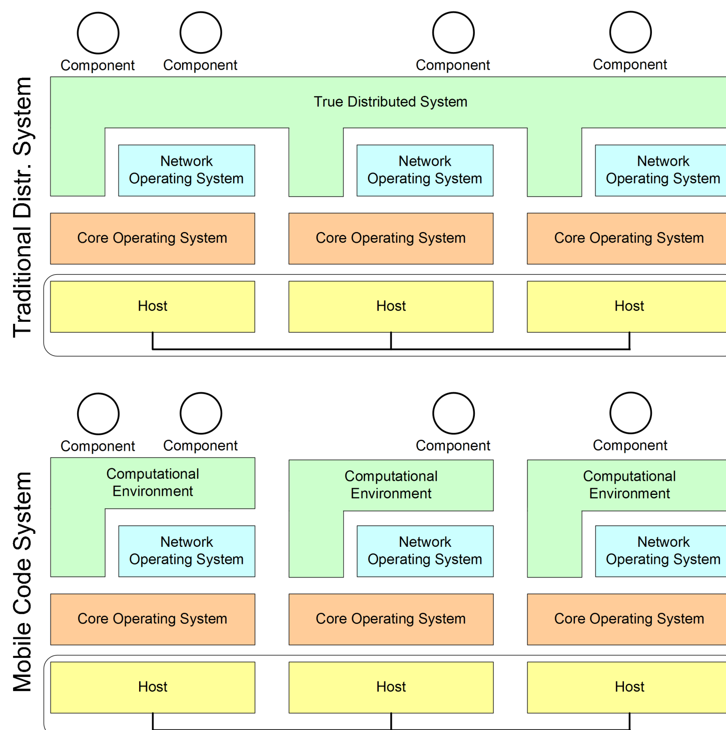

Abbildung 3.1: Architektur herkömmlicher verteilter Systeme (oben) versus Mobile Code Systems (unten) (nach [Fuggetta u. a. 1998])

Das System insgesamt besteht aus einer Menge von Geräten (engl. *hosts*), welche untereinander über ein Netzwerk verbunden sind. Auf jedem Gerät läuft ein sogenanntes *Core Operating System* (COS), also die Kernkomponente für alle grundlegenden Funktionen, wie dem Zugriff auf das Dateisystem, die Speicherverwaltung, Prozessunterstützung etc. Auf dieser Ebene wird noch keine Unterstützung für Kommunikation oder Verteilung geboten. Diese Funktionen stellt das *Network Operating System* (NOS) zur Verfügung, welches auf dem COS aufsetzt. Anwendungen, welche die Funktionen des NOS nutzen wollen, müssen die Dienste, die entfernt in Anspruch genommen werden sollen, explizit adressieren. Das bedeutet, es werden lediglich nicht-transparente Funktionen zur Verfügung gestellt und eine Anwendung muss Informationen darüber besitzen, wie das zugrunde liegende Netzwerk beschaffen ist, um mit einem bestimmten Ziel zu kommunizieren.

Die Netzwerktransparenz wird schließlich durch eine weitere Schicht, dem sogenannten *True Distributed System* (TDS), erreicht. Komponenten, die auf dem TDS aufsetzen, nehmen alle anderen Komponenten, die über das Netzwerk verteilt sind, als lokal vorhanden wahr. Die Mechanismen des Zugriffs auf diese eigentlich entfernten Komponenten und deren angebotene Dienste unterscheiden sich daher nicht von dem Zugriff auf lokal vorliegende Dienste bzw. Komponenten. Die Sicht auf das Netzwerk ist also völlig transparent und eine Anwendung braucht der zugrunde liegenden Netzwerkstruktur nicht gewahr zu sein. Zum Beispiel lassen sich CORBA-Dienste [Object Management Group 1999] als Dienste der TDS-Schicht auffassen. Ein CORBA-Programmierer braucht sich normalerweise der Struktur des Netzwerkes nicht bewusst zu sein und interagiert ausschließlich mit einem einzigen, im Vorwege bekannten *Object Broker*. Dieser Broker vermittelt Anfragen und stellt den Anwendungen auf diese Weise eine transparente Sicht auf das Netzwerk zur Verfügung.

Eine andere Perspektive nehmen Technologien ein, die Codemobilität unterstützen. Anstatt die Struktur des zugrunde liegenden Netzwerkes vor den Anwendungen und dem Programmierer zu verstecken, wird diese explizit offenbart. Die untere Hälfte der Abbildung 3.1 zeigt den Aufbau eines verteilten Systems, welches Codemobilität unterstützt. Man spricht in diesem Fall auch von einem *Mobile Code System* (MCS). Der einzige, aber gewichtige Unterschied zu den herkömmlichen verteilten Systemen liegt in der Schicht über dem COS. Wie bereits beschrieben, erlaubt ein TDS den Anwendungen in einem herkömmlichen System eine transparente Sicht auf das Netzwerk. Als Ersatz für dieses TDS findet man in MCSs eine Schicht, die sich *Computational Environment* (CE) (dt. *Ausführungsumgebung*) nennt. Diese Schicht bewahrt die Identität des Computers, auf dem sie angesiedelt ist, und stellt den Anwendungen diese Information explizit zur Verfügung. Außerdem bietet diese Schicht, unter Verwendung der Kernfunktionen des COS und NOS, eine Menge von Funktionen, mit Hilfe derer eine Anwendung dynamisch ihre Komponenten samt ihres Zustandes von einer CE in eine andere CE umlagern kann.

Ausführungseinheiten und Ressourcen

Die Komponenten einer Anwendung unterteilt man, je nach Art, in *Ausführungseinheiten* (engl. *execution units*) (EU) und Ressourcen (siehe Abbildung 3.2). Ausführungseinheiten repräsentieren sequentielle Befehlsabläufe, zum Beispiel einfache Prozesse. Ressourcen repräsentieren Entitäten, die von mehreren EUs geteilt werden können, zum Beispiel Dateien, von mehreren Prozessen benutzte Objekte oder Variablen des Betriebssystems.

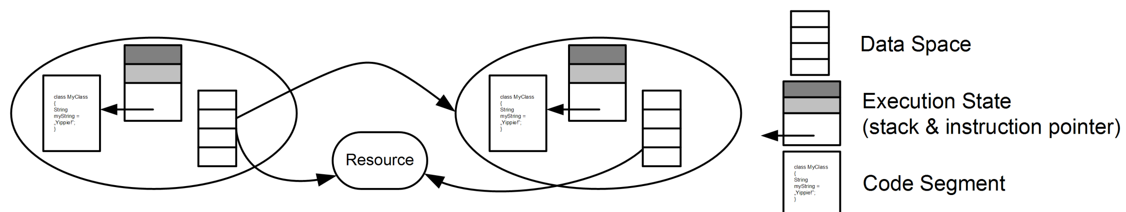

Abbildung 3.2: Interne Struktur zweier Ausführungseinheiten (nach [Fuggetta u. a. 1998])

Eine Ausführungseinheit besteht neben dem Programmcode, also dem sequentiellen Befehlsablauf (engl. *code segment*), noch aus einem Programmzustand, welcher in einen Datenraum (engl. *data space*) und einen Ausführungszustand (engl. *execution state*) unterteilt werden kann. Der Datenraum ist die Menge der Referenzen auf alle Ressourcen, auf welche die Ausführungseinheit zugreifen kann. Der Ausführungszustand enthält die Kontrollinformationen über die aktuelle

Berechnung, zum Beispiel den Aufrufstapel (engl. *call stack*) sowie den Befehlszeiger (engl. *instruction pointer* oder *program counter*) [Fuggetta u. a. 1998].

3.1.2 Mechanismen

Wie im vorangehenden Abschnitt bereits erwähnt, besteht eine Ausführungseinheit (EU) aus dem Programmcode (*code segment*), ihrem Datenraum (*data space*) und dem aktuellen Ausführungszustand (*execution state*). Die Besonderheit von *Mobile Code Systems* (MCS) im Gegensatz zu den herkömmlichen verteilten Systemen, besteht in der Möglichkeit, jeden Teil einer EU jederzeit auf andere Knoten des Netzwerkes umzusiedeln. Der Teil einer Ausführungseinheit, der tatsächlich migriert werden soll, wird zur Laufzeit durch verschiedene Mechanismen bestimmt. Man unterscheidet hierbei Mechanismen, die einerseits die Mobilität von Programmcode und Ausführungszustand unterstützen (engl. *code and execution state management*) und andererseits Mechanismen der Datenraumverwaltung (engl. *data space management*). Um diese Unterscheidung hervorzuheben, werden die einzelnen Mechanismen, beginnend mit der Mobilität von Programmcode und Ausführungszustand, im Folgenden getrennt voneinander behandelt. Eine Klassifikation der einzelnen Mechanismen sowie ihrer Unterklassen findet sich in Abbildung 3.3 [Fuggetta u. a. 1998].

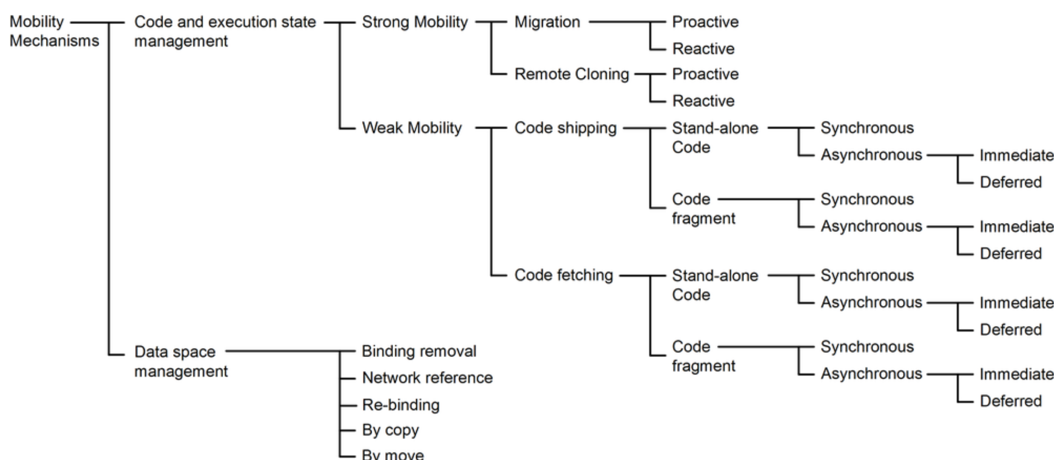

Abbildung 3.3: Klassifikation von Mobilitätsmechanismen (nach [Fuggetta u. a. 1998])

Mobilitätsmechanismen für Programmcode und Ausführungszustand

Bei existierenden MCSs unterscheidet man zwei verschiedene Möglichkeiten der Mobilität. Auf der einen Seite steht die schwache Mobilität (engl. *weak mobility*), auf der anderen Seite die starke Mobilität (engl. *strong mobility*). Die schwache Migration erlaubt es, den Programmcode und ggf. eine Reihe von Initialisierungswerten von einer Ausführungsumgebung (CE) zu einer anderen zu transferieren und dort auszuführen[1]. Die starke Migration erlaubt zusätzlich den Transfer des Ausführungszustandes, also den gesamten Aufrufstapel sowie den Befehlszeiger. Somit ist es möglich, ein laufendes Programm zu jedem Zeitpunkt einzufrieren, in eine andere Ausführungsumgebung zu transferieren und dort an demselben Ausführungspunkt fortzuführen [Fuggetta u. a. 1998, Illmann u. a. 2000].

[1] Illmann u.a. bezeichnen jegliche Art der Migration, die nicht stark ist, als schwache Migration [Illmann u. a. 2000].

Betrachtet man die einzelnen Bestandteile einer Ausführungseinheit, so lassen sich hinsichtlich der Mobilität verschiedene Aspekte unterscheiden. Abbildung 3.4 zeigt eine Klassifikation der verschiedenen Migrationsaspekte nach [Illmann u. a. 2000]. Auf der höchsten Ebene findet sich die *starke Migration* (engl. *strong migration*). Auf der darunterliegenden Ebene besteht die Klassifikation aus zwei orthogonalen Aspekten: der Migration des Programmcodes (engl. *code migration*) und der Migration des Programmzustandes (engl. *state migration*).

Migration des Programmcodes bedeutet, dass sämtlicher Code einer EU von einer CE in eine entfernte Ziel-CE übertragen werden muss. Hierzu gehört nicht nur der Code der EU selbst, sondern auch der Programmcode sämtlicher referenzierter Objekte (ausgenommen Ressourcen). Hierbei ist zu berücksichtigen, dass möglicherweise die Ziel-CE bereits einen Teil des erforderlichen Programmcodes selbst bereitstellt, sodass unter Umständen nur einzelne Codefragmente übertragen werden müssen.

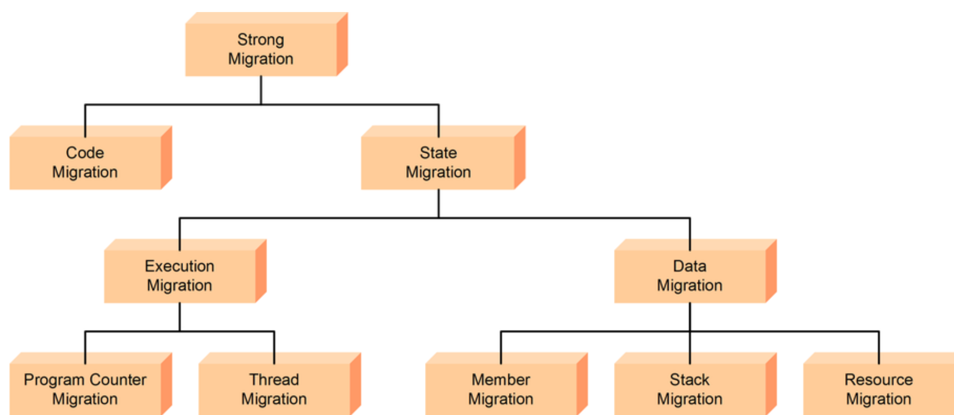

Abbildung 3.4: Klassifikation verschiedener Migrationsaspekte (nach [Illmann u. a. 2000])

Die Migration des Programmzustandes lässt sich in zwei weitere Aspekte untergliedern: die Migration des Ausführungszustandes (engl. *execution migration*) und die Migration des Datenraumes (engl. *data migration*). Die Migration des Ausführungszustandes umfasst die Migration des Befehlszeigers (engl. *program counter migration*), der den nächsten auszuführenden Befehl des Agenten anzeigt, sowie die Migration aller Prozesse (engl. *thread migration*), die von dem Agenten erzeugt wurden. Die Prozessmigration ist besonders anspruchsvoll, da sie rekursiv alle Aspekte der starken Migration umfasst und darüber hinaus die Synchronisationsvariablen der Threads berücksichtigen muss. Zur Migration des Datenraumes gehört die Migration der Zustandsvariablen aller Objekte, die zu einer EU gehören (engl. *member migration*), die Migration des Aufrufstapels (engl. *stack migration*), also aller gerade im Aufruf befindlichen Methoden samt ihrer lokalen Variablen und Operanden sowie ggf. die Migration gebundener Ressourcen (engl. *resource migration*), sofern dies möglich ist [Illmann u. a. 2000].

Bei der starken Mobilität unterscheidet man weitergehend zwei Mechanismen: 1) die Migration und 2) das entfernte Duplizieren (engl. *remote cloning*), siehe Abbildung 3.3. In beiden Fällen wird zwar der Programmcode inklusive des Ausführungszustandes transferiert und an entfernter Stelle der Programmlauf fortgesetzt, jedoch wird bei der Migration die Bindung der EU an ihr aktuelles CE getrennt und an entfernter Stelle eine neue Bindung der EU an die Ausführungsumgebung erstellt. Bei dem entfernten Duplizieren bleibt die ursprüngliche Bindung erhalten. Nur das Duplikat erhält an entfernter Stelle eine neue Bindung. Beide Mechanismen können dar-

über hinaus proaktiv oder reaktiv angewandt werden [2]. Bei der proaktiven Migration bzw. der proaktiven Duplizierung entscheidet die Ausführungseinheit selbständig über den Zielort und den Zeitpunkt des Transfers. Im reaktiven Fall wird diese Entscheidung von einer anderen Ausführungseinheit bestimmt.

Auch bei der schwachen Migration können zwei verschiedene Mechanismen unterschieden werden: 1) der übertragene Programmcode wird dynamisch an eine bereits existierende Ausführungseinheit gebunden oder 2) für den Programmcode wird eine neue Ausführungseinheit erzeugt. Beide Mechanismen können weitergehend in verschiedene Unterklassen eingeteilt werden. Die Einteilung beruht auf der Art des Programmcodes, der Richtung des Codetransfers, der einhergehenden Synchronisation und dem Zeitpunkt an dem der Programmcode wirklich zur Ausführung gelangt.

Die Art des Programmcodes lässt sich unterteilen in Codefragment und vollständiger Code (engl. *stand-alone code*). Bei der Richtung des Codetransfers unterscheidet man grundlegend zwischen Abrufen (engl. *fetch* oder *pull*) und Versenden (engl. *ship* oder *push*) des Programmcodes[3] Wenn eine EU ihre Ausführung unterbricht um auf die Ausführung des gerade transferierten Programmcodes an entfernter Stelle zu warten, spricht man von einem synchronen Mechanismus, andernfalls von einem asynchronen. Bei asynchronen Mechanismen unterscheidet man darüber hinaus zwischen sofortiger (engl. *immediate*) und verzögerter (engl. *deferred*) Ausführung, je nachdem, wann der transferierte Code tatsächlich zur Ausführung gelangt [Fuggetta u. a. 1998].

Mobilitätsmechanismen zur Datenraumverwaltung

Wenn eine EU von einer CE zu einer neuen CE migriert, muss ihr Datenraum, also die Menge aller Ressourcenbindungen, neu arrangiert werden. Eine Ressourcenbindung ist zum Beispiel eine geöffnete Datei, eine Netzwerkverbindung oder eine Objektreferenz. Diese Neuordnung kann zur Folge haben, dass alte Bindungen für ungültig erklärt werden, neue Bindungen hergestellt werden oder sogar einige gebundene Ressourcen zusammen mit der EU in die neue CE migriert werden müssen. Die Wahl eines geeigneten Vorgehens hängt primär von dem Typ der jeweils gebundenen Ressource, der Art der Bindung und den Anforderungen der Anwendung ab.

Der Typ einer Ressource ist durch ihre Informationsstruktur und Schnittstelle bestimmt. Er bestimmt, ob eine Ressource transferierbar (engl. *transferable*) oder nicht-transferierbar ist und somit, ob es überhaupt möglich ist, die Ressource zusammen mit der Ausführungseinheit migrieren zu lassen und dabei die Bindung zu erhalten. Beispielsweise ist eine Ressource des Typs 'Datei' prinzipiell transferierbar, eine Ressource des Typs 'Drucker' nicht. Transferierbare Ressourcen lassen sich weiterhin in freie (engl. *free*) oder feste (engl. *fixed*) Ressourcen unterteilen. Somit lässt sich die prinzipielle Transferierbarkeit von Ressourcen einschränken. Zum Beispiel wäre es im Falle sehr großer Dateien eventuell performanter, wenn die Dateien nicht gleich mit der EU komplett in die neue CE transferiert werden und stattdessen später bei Bedarf ein Zugriff auf die Dateien über das Netzwerk erfolgt.

Bei der Art von Bindung unterscheidet man drei Mechanismen: 1) die Bindung über einen Identifikator (engl. *by identifier*), 2) die Bindung über einen Wert (engl. *by value*) und 3) die Bindung über einen Typ (engl. *by type*)), siehe Tabelle 3.1. Die Bindung über einen Identifikator ist die stärkste Art der Bindung. Die gebundene Ressource wird hierbei eindeutig identifiziert und kann

[2]Cabri u.a. bezeichnen die proaktive Migration auch als *explizite*, die reaktive Migration als *implizit* [Cabri u. a. 1998]

[3]Je nach Art des Programmcodes kann man weitergehend noch in *pull-per-class*, *pull-all-classes*, *push-per-class*, *push-all-classes* und bei Berücksichtigung der Empfänger auch noch in *push-all-to-next*, *push-all-to-all* etc. unterteilen [Braun und Rossak 2005].

nicht gegen eine andere Ressource ausgetauscht werden. Für die Migration der EU bedeutet dies, dass die gebundene Ressource entweder mit verschoben (engl. *by move*) oder nach der Migration über eine Netzwerkverbindung referenziert werden muss (engl. *network reference*).

Bei der Bindung über einen Wert verlangt die EU lediglich den Zugriff auf eine Ressource gleichen Typs und gleichen Wertes. Die Identität der Ressource spielt keine Rolle. Bei der Migration kann also entweder eine Kopie der Ressource zusammen mit der Ausführungseinheit transferiert (engl. *by copy*) oder eine Referenz auf dieselbe Ressource über das Netzwerk hergestellt werden.

Die schwächste Form der Bindung ist die Bindung über einen Typ. Hierbei muss lediglich sichergestellt werden, dass eine EU zu jedem Zeitpunkt an eine Ressource bestimmten Typs gebunden sein muss, gleich welche Identität oder welchen Wert diese Ressource hat. Diese Art der Bindung erfolgt meist an Ressourcen, die von jeder CE innerhalb des Netzwerkes zur Verfügung gestellt werden, zum Beispiel bestimmte Laufzeitbibliotheken oder ein Anzeigegerät. Nach der Migration kann daher eine solche Bindung einfach an dem neuen Ort mit den lokalen Ressourcen wieder hergestellt werden (engl. *by re-binding*).

Die Datenraumverwaltung wird also bei der Migrationsvorbereitung vor die Entscheidung gestellt, Ressourcen entweder zusammen mit der EU zu transferieren (entweder durch Kopieren oder Verschieben) oder die Bindung zu diesen oder gleichwertigen bzw. gleichartigen Ressourcen an dem Zielort wieder herzustellen (entweder über eine Netzwerkverbindung oder durch Bindung an neue Ressourcen). Tabelle 3.1 fasst die Möglichkeiten der Datenraumverwaltung hinsichtlich der Ressourcentypen und Bindungsarten noch einmal zusammen.

	Free Transferable	*Fixed Transferable*	*Fixed Not Transferable*
By Identifier	By move (Network reference)	Network reference	(Network reference)
By Value	By copy (By move, Network reference)	By copy (Network reference)	(Network reference)
By Type	Re-binding (Network reference, By copy, By move)	Re-binding (Network reference, By copy)	Re-binding (Network reference)

Tabelle 3.1: Ressourcen, Bindungen und Datenraumverwaltungsmechanismen (nach [Fuggetta u. a. 1998])

3.1.3 Technologien

Im vorangegangenen Abschnitt wurden eine Reihe von Mobilitätsmechanismen behandelt. Auf der einen Seite standen Mechanismen, die den Programmcode und den Zustand eines Programms in verschiedenartiger Weise zwischen Ausführungsorten transferieren. Auf der anderen Seite wurden Mechanismen zur Verwaltung des Datenraumes eines Prozesses erläutert und es wurden verschiedene Möglichkeiten und Weisen beschrieben, den Datenraum bei Wechsel der Ausführungsumgebung eines Prozesses zu reorganisieren. Im Folgenden sollen nun einige Beispiele von Technologien für mobilen Code vorgestellt werden, wobei ein besonderes Augenmerk unter anderem auf den besprochenen Mobilitätsmechanismen liegt. Die Auswahl der vorgestellten Technologien folgte der subjektiv wahrgenommenen Häufigkeit der Referenzen in wissenschaftlicher Literatur und stellt in diesem Sinne keine Wertung dar. Während Agent TCL, Telescript, Tabriz und Odyssey wichtige (historische) Meilensteine der Mobile Code-Systeme dar-

stellen, zählt die Java-Technologie heute als de-facto Standard in diesem Bereich. Sumatra wurde unter einer Reihe von Technologien mit ähnlichen Schwerpunkten beispielhaft aufgrund der durchdachten Konzepte ausgewählt.

Agent TCL

Die *Tool Command Language* (TCL) ist eine für Unix-Systeme entwickelte und flexibel erweiterbare Skript-Sprache. An der Universität Darthmouth wurde darauf aufbauend *Agent TCL* [Gray 1995] entwickelt. Diese Erweiterung erlaubt es einem Programm, während der Laufzeit den Ausführungsort zu wechseln, wobei sowohl der Programmcode als auch der -zustand migriert werden können. Nach obiger Klassifikation spricht man in diesem Fall von starker Migration [Fuggetta u. a. 1998, Cabri u. a. 1998]. Die EUs (*Agents* genannt) können ausschließlich auf Ressourcen zugreifen, die ihnen von dem Unix-Betriebssystem zur Verfügung gestellt werden. Da diese Ressourcen als nicht-transferierbar angesehen werden, wird in Agent TCL eine Bindung zwischen einer EU und einer Ressource bei der Migration einfach entfernt.

Der Transfer einer EU kann auf drei verschiedene Arten erfolgen. Mittels des *jump*-Kommandos kann eine EU proaktiv die Migration ihres kompletten Prozesses, inklusive des Zustandes, in eine neue CE veranlassen. Über den *fork*-Befehl kann eine EU proaktiv ein Duplikat ihrer selbst in eine andere CE kopieren (*remote cloning*). Und mittels einer *submit*-Anweisung können einzelne Codefragmente an entfernte EUs zur Ausführung übergeben werden [Fuggetta u. a. 1998].

Telescript, Tabriz & Odyssey

Im Gegensatz zu Agent TCL, welches eine reine Skriptsprache darstellt, folgt Telescript [White 1999] einem objektorientierten Ansatz. Die Sprache wurde Anfang der neunziger Jahre von der Firma *General Magic* entwickelt, um agentenorientierte Konzepte für mobile Geräte umzusetzen, wobei insbesondere der Fokus auf Sicherheit und starker Mobilität lag [Fuggetta u. a. 1998, Busse 1999]. Telescript unterscheidet zwei Arten von EUs: Agenten (*agents*) und Plätze (*places*). Ein Platz ist eine Art stationärer Container, welcher andere EUs aufnehmen kann. Im Gegensatz dazu sind Agenten die mobilen, transferierbaren Programmeinheiten. Beide Arten von EUs werden in sogenannten *engines* ausgeführt, welche bei Telescript die CEs repräsentieren.

Ebenso wie bei Agent TCL, können Telescript-Agenten zusammen mit ihrem Ausführungszustand zur Laufzeit proaktiv migrieren, entweder indem sie sich selbst oder eine Kopie in eine andere CE verschieben. Das Übertragen von Codefragmenten hingegen ist nicht vorgesehen. Telescript hat eine durchdachte Datenraumverwaltung, welche sich auf das Zugehörigkeitskonzept (engl. *ownership concept*) stützt. Hierbei wird mit jeder Ressource eine EU assoziiert, welche den Inhaber dieser Ressource darstellt. Wechselt eine EU ihre CE, so werden alle ihr zugehörigen Ressourcen mit verschoben. Die Bindungen anderer EUs an diese Ressourcen werden entfernt [Fuggetta u. a. 1998].

Da Telescript Anfang der Neunziger zwar mobile Geräte, nicht aber das zu der Zeit an Bedeutung gewinnende Internet, im Visier hatte, wurde nachfolgend mit *Tabriz AgentWare* eine neue Technologie von der Firma General Magic auf den Markt gebracht. Diese setzt auf den Telescript-Konzepten auf, ist allerdings nicht mehr auf mobile Geräte, sondern vielmehr auf ans Internet angeschlossene Geräte zugeschnitten. Doch aufgrund der geschlossenen, proprietären Technologie von Tabriz fand auch diese keinen großen Anklang. 1997 wurde schließlich die weitere Entwicklung von Telescript/Tabriz zugunsten eines neuen Systems eingestellt: *Odyssey* [Busse 1999].

Odyssey stützt sich auf die Konzepte von Telescript/Tabriz, ist allerdings eine Erweiterung der *Java*-Sprache, womit sowohl der potenzielle Einsatz auf mobilen Geräten als auch der Einsatz im Internet und schließlich eine offene Programmierschnittstelle einhergehen. Im Gegensatz zu dem nachfolgend vorgestellten *Sumatra* [Acharya u. a. 1997] verlangt Odyssey keine Änderungen an dem nativen Java Interpreter, bietet somit allerdings auch keine starke, sondern lediglich schwache Mobilität [Cabri u. a. 1998].

Java

Java [Sun 2007c], entwickelt von *Sun Microsystems Inc.*, ist eine objektorientierte, plattformunabhängige Programmiersprache, die vielfältig eingesetzt wird. Aufgrund einiger bereits integrierter Mechanismen, die für die Mobilität von Programmen essenziell sind, findet sie häufig bei der Programmierung von MCSs Verwendung [Braun u. a. 2005] (vgl. auch Abschnitt 3.1.5). Bereits übersetzter Java Programmcode (*bytecode*) wird von der sogenannten *Java Virtual Machine* (JVM) interpretiert. In einer virtuellen Maschine können mehrere Java Prozesse (*java threads*) laufen. Diese virtuelle Maschine repräsentiert die CE, die Java Prozesse stellen die EUs dar.

Als *Classloader* bezeichnet Java einen Mechanismus, mit Hilfe dessen die JVM zur Laufzeit Codefragmente (*Java classes*) laden und dynamisch binden kann. Hierbei ist es möglich, sowohl lokal als auch entfernt vorliegende Klassen in eine JVM zu laden. Der Vorgang des Ladens und Bindens findet im Allgemeinen implizit statt. Er wird angestoßen, sobald der in der JVM ausgeführte Programmcode nicht aufgelöste Klassenreferenzen aufweist. In diesem Fall wird der geladene und neu gebundene Programmcode sofort ausgeführt (engl. *immediate execution*). Zusätzlich ist es auch möglich, den Vorgang explizit von Anwendungsseite her anzustoßen. In diesem Fall kann die Ausführung des nachgeladenen Programmcodes auch verzögert (engl. *deferred execution*) geschehen [Fuggetta u. a. 1998].

Aus diesen Gründen unterstützt Java schwache Mobilität, asynchrones Abrufen von Programmcode und sowohl dessen sofortige als auch verzögerte Ausführung. In keinem Fall wird allerdings ein Ausführungszustand oder etwaige Ressourcenbindungen mit übertragen, daher bietet Java auch keinerlei Mechanismen zur Datenraumverwaltung. Unter Zuhilfenahme der bereits als Kernfunktionalität integrierten Serialisierungsmechanismen ist es jedoch möglich, Java-Objekte inklusive ihres internen Zustandes (Variablenbelegungen) zu transferieren.

Sumatra

Sumatra [Acharya u. a. 1997] ist eine Java-Erweiterung. Sie wurde von der Universität Maryland entwickelt, um Java-Programmen die Anpassung an sich verändernde (Hardware-) Ressourcen zu erlauben. Hierfür können die Programme die von Sumatra gebotenen Mobilitätsmechanismen nutzen, um Programmcode und -zustand zwischen CEs zu transferieren [Fuggetta u. a. 1998].

Um diese starke Mobilität bieten zu können, nimmt Sumatra Änderungen an dem Java Interpreter, der Java Virtual Machine, vor. Diese Änderungen sorgen dafür, dass die JVM den Ausführungszustand eines Java Programms schrittweise verfolgt, damit sowohl der Aufrufstapel als auch der Befehlszeiger eines Java Prozesses zugänglich sind und transferiert werden können [Acharya u. a. 1997, Cabri u. a. 1998]. Auch in Sumatra sind Java-Prozesse die EUs, neu eingeführte Ausführungsmaschinen *execution engines* lösen hingegen die herkömmliche JVM als CE ab. Diese Ausführungsmaschinen erweitern die abstrakte Maschine der JVM um proaktive Migrationsmechanismen, proaktives entferntes Duplizieren und Versenden von Codefragmenten mit synchroner und sofortiger Ausführung.

Auch können Prozesse und Codefragmente unabhängig von etwaigen gebundenen oder benötigten Ressourcen transferiert werden. Die Objektgruppen-Abstraktion (engl. *object group abstraction*) bietet hierfür die Möglichkeit, Einheiten von zu migrierenden Objekten zur Laufzeit zusammenzufassen. Die Datenraumverwaltung für eine Objektgruppe verschiebt diese immer als Ganzes, hierbei werden jedoch die Bindungen zu Objekten außerhalb der Objektgruppe durch Netzwerkreferenzen ersetzt [Fuggetta u. a. 1998, Acharya u. a. 1997].

3.1.4 Entwurfsparadigmen

Um verteilte Softwaresysteme zu bauen, genügt es nicht, geeignete Technologien zur Entwicklung eines Systems zu verwenden. Technologie und Methodologie sollten in diesem Prozess eng miteinander in Beziehung gesetzt werden, um eine passende Softwarearchitektur zu entwerfen. Eine Softwarearchitektur, definiert als die Dekomposition eines Softwaresystems in Softwarekomponenten und ihre Interaktionen [Shaw und Garlan 1996], kann hierbei durch Entwurfsparadigmen repräsentiert werden. Diese hängen allerdings nicht notwendigerweise von den verwendeten Technologien ab, sondern sollten, wie nachfolgend geschehen, als konzeptuell eigenständige Einheiten betrachtet werden [Carzaniga u. a. 1997, Fuggetta u. a. 1998].

Die Paradigmen für den Entwurf herkömmlicher verteilter Softwaresysteme (also ohne die Unterstützung mobiler Programmeinheiten) sind für den Aufbau komplexer, verteilter Anwendungen nicht mehr ausreichend. Die Konzepte des Ortes, der Verteilung von Softwarekomponenten auf diese Orte sowie die Migration von Komponenten zwischen den Orten, müssen explizit bereits beim Entwurf des Systems berücksichtigt werden. Der Grund hierfür ist, dass sich lokale Interaktionen von Komponenten hinsichtlich der Latenzzeit, des Zugriffs auf einen Speicher, des partiellen Ausfalls und der Nebenläufigkeit gänzlich anders verhalten, als Interaktionen über Ortsgrenzen hinweg [Kendall u. a. 1994]). Daher ist es wichtig, angemessene Entwurfsparadigmen für verteilte Systeme, die Codemobilität unterstützen, zu identifizieren und in Beziehung zu den Technologien zu setzen, mittels derer sie implementiert werden können [Fuggetta u. a. 1998].

Konzepte

Bevor näher auf die einzelnen Paradigmen eingegangen wird, gilt es einige grundlegende Konzepte zu betrachten, die von den Entitäten (z.B. Dateien, Variablenbelegungen, ausführbarer Programmcode, Prozesse etc.) eines Softwaresystems abstrahieren [Carzaniga u. a. 1997].

Komponenten Ein Softwaresystem ist aus einer Reihe von Komponenten (engl. *components*) aufgebaut. Diese unterteilt man in sogenannte *Codekomponenten* (engl. *code components*), *Berechnungskomponenten* (engl. *computational components*) und *Ressourcenkomponenten* (engl. *resource components*). Codekomponenten kapseln das Wissen, wie eine bestimmte Berechnung auszuführen ist. Sie sind also vergleichbar mit einer Art Rezept. Die Berechnungskomponenten führen aktiv Berechnungen aus, die von den Codekomponenten vorgegeben werden. Ressourcenkomponenten repräsentieren Daten oder Geräte, die von den Berechnungskomponenten während der Ausführung einer Berechnung benötigt werden.

Ausführungsort Den Ort, im Sinne eines Gerätes, an dem eine Berechnung durch eine Berechnungskomponente ausgeführt wird, bezeichnet man als Ausführungsort (engl. *site*). Ein solcher kann mehrere Komponenten beherbergen.

Interaktion Eine Interaktion bezeichnet ein Ereignis, an dem mindestens zwei Komponenten untereinander beteiligt sind. Eine Interaktion zwischen Komponenten desselben Ausführungsortes wird im Allgemeinen als weniger kostspielig angesehen als eine Interaktion über Ortsgrenzen hinweg.

Im Folgenden sollen nun die Entwurfsparadigmen näher beschrieben werden. In [Carzaniga u. a. 1997] wurden drei hauptsächliche Paradigmen identifiziert, die die Mobilität von Code unterstützen. Zu diesen gehören die *Entfernte Auswertung* (engl. *Remote Evaluation*), *Code auf Anforderung* (engl. *Code On Demand*) und *Mobile Agenten* (engl. *Mobile Agents*). Jedes dieser Paradigmen ist charakterisiert 1) durch den Ort der für eine Berechnung notwendigen Komponenten vor und nach der Berechnung, 2) durch die Berechnungskomponente, welche die Dienstleistung erbringt und 3) den Ort, an dem die Berechnung letztendlich ausgeführt wird.

Die Betrachtung der Entwurfsparadigmen stützt sich auf ein Szenario, in dem eine Berechnungskomponente A, welche an einem Ausführungsort S_A liegt, das Ergebnis einer bestimmten Dienstleistung benötigt. An der Erbringung dieser Dienstleistung ist eine weitere Berechnungskomponente B an einem Ausführungsort S_B involviert. Eine Übersicht über die einzelnen Paradigmen findet sich in Abbildung 3.5. In dieser Abbildung sind die beteiligten Ausführungsorte samt ihrer Komponenten, sowohl vor als auch nach der Berechnung, dargestellt. Die Dreiecke repräsentieren die Berechnungskomponenten A bzw. B. In fetter Schrift hervorgehoben, ist jeweils diejenige Komponente, die die Berechnung letztendlich durchführt.

Abbildung 3.5: Verschiedene Entwurfsparadigmen von Mobile Code Systems (nach [Carzaniga u. a. 1997])

Client/Server-Paradigma

Das klassische Client/Server-Paradigma stellt zwar kein Paradigma für die Verwendung mobilen Codes dar, findet aber oft bei dem Entwurf von verteilten Systemen Verwendung. Eine Berechnungskomponente B (der Server) offeriert von einem Ort S_B aus eine Menge von Diensten. Die hierfür benötigten Code- und Ressourcenkomponenten liegen ebenfalls an S_B. Ein Klient A kann von S_A aus mit der Komponente B in Interaktion treten und das Erbringen einer Dienstleistung fordern. Als Antwort darauf führt B, unter Verwendung der lokalen Code- und Ressourcenkomponenten, die Berechnung aus und liefert das Ergebnis durch eine zweite Interaktion zurück an A. Sowohl die Berechnungs-, Ressourcen- und Codekomponenten verbleiben an ihren Orten. Es findet also kein Austausch von Programmcode statt [Carzaniga u. a. 1997, Braun und Rossak 2005].

Remote Evaluation-Paradigma

In diesem Paradigma hat die Berechnungskomponente A, beheimatet an dem Ausführungsort S_A, lokalen Zugriff auf die für eine geforderte Berechnung notwendigen Codekomponenten. Ihr fehlen allerdings die Ressourcen an dem Ausführungsort. A weiß allerdings, dass die Ressourcen an einem Ort S_B zur Verfügung stehen und sendet daher (Kopien der) Codekomponenten an die Berechnungskomponente B. Mittels dieser führt B die Berechnung durch und schickt das Ergebnis wiederum zurück an A [Carzaniga u. a. 1997, Braun und Rossak 2005].

Code On Demand-Paradigma

Im Gegensatz zu der Remote Evaluation hat die Berechnungskomponente A am Ausführungsort S_A Zugriff auf die für eine Berechnung notwendigen Ressourcen, nicht aber auf das Wissen, wie die Berechnung durchzuführen ist. Das Wissen in Form von Codekomponenten liegt an einem Ausführungsort S_B. Folglich interagiert A mit der Berechnungskomponente B und bittet um (eine Kopie) der erforderlichen Codekomponenten. In einer zweiten Interaktion übersendet B die benötigten Komponenten an A, welches nun die Berechnung selbständig am Ort S_A ausführen kann [Carzaniga u. a. 1997, Braun und Rossak 2005].

Mobile Agent-Paradigma

Die Ausgangssituation in diesem Paradigma ist ähnlich wie bei der Remote Evaluation [4]. Eine Berechnungskomponente A kann eine Berechnung am Ausführungsort S_A aufgrund fehlender Ressourcen nicht durchführen. Wiederum hat A lokalen Zugriff auf die benötigten Codekomponenten und weiß, dass an Ausführungsort S_B die erforderlichen Ressourcen bereitstehen. In diesem Paradigma wird allerdings keine Berechnungskomponente B verlangt, die im Auftrag von A mittels dessen Codekomponenten eine Berechnung durchführt. Stattdessen besitzt A die Fähigkeit den Ausführungsort zu wechseln, migriert folglich von S_A nach S_B und nimmt sowohl die Code- als auch ggf. einen Teil der bereits auf S_A vorhandenen Ressourcenkomponenten mit nach S_B, wo A schließlich die Berechnung durchführen kann. Im Gegensatz zu den anderen Paradigmen werden hier nicht nur Codekomponenten bzw. Berechnungsergebnisse ausgetauscht. Stattdessen wird eine komplette Berechnungskomponente samt ihres Zustandes von einem Ort zu einem anderen bewegt [Carzaniga u. a. 1997, Braun und Rossak 2005].

[4]Eine detailliertere Beschreibung dieses Paradigmas findet sich in Abschnitt 3.2

Zusammenfassung

Die hier vorgestellten Paradigmen werden bereits erfolgreich für den Entwurf großer, verteilter Systeme eingesetzt [Carzaniga u. a. 1997]. Die wichtigste Neuerung dieser Paradigmen im Vergleich zu den klassischen Paradigmen (z.B. gegenüber dem Client/Server-Paradigma) ist die explizite Modellierung des Konzeptes eines Ortes. Der Ausführungsort (*site*) wurde als Abstraktion bereits auf der Entwurfsebene eingeführt um den Standort der verschiedenen Komponenten zu verdeutlichen.

Die klassischen Paradigmen sind darüber hinaus als statisch anzusehen, da sowohl der Programmcode allgemein als auch der Ort der einzelnen Komponenten während der Laufzeit unverändert bleiben. Dies hat zur Folge, dass auch die Arten der Interaktion sowie ihre Qualität (lokal oder entfernt) von vornherein festgelegt sind. Mobile Code-Paradigmen überwinden diese Beschränkungen, indem sie die Mobilität von Komponenten berücksichtigen und somit die Art und Qualität der Interaktionen dynamisch anpassen können. Hieraus resultiert wiederum die Möglichkeit Interaktionskosten zu reduzieren, indem interagierende Komponenten in gegenseitige Nähe gebracht werden.

Den Autoren Fuggetta u.a. zufolge ist diese Flexibilität und Dynamik zwar grundsätzlich nützlich, unklar bleibt allerdings, in welchen Fällen die beschriebenen Paradigmen für den Entwurf eingesetzt werden sollen und wie man das richtige Paradigma für einen Anwendungszweck auswählt. Im Rahmen dieser Arbeit soll, wie auch von [Braun u. a. 2005] vorgeschlagen, ausschließlich das Mobile Agent-Paradigma berücksichtigt werden. Eine weitere Einschränkung wird auch hinsichtlich der verwendeten Technologie gemacht. Wie bereits in Abschnitt 3.1.3 bei der Vorstellung verschiedener Technologien erwähnt, ist die Programmiersprache Java bereits ein de-facto Standard für die Programmierung von Mobile Code-Systemen und wird daher im folgenden Abschnitt näher behandelt.

3.1.5 Java als de-facto Standard für MCS

Die Programmiersprache Java [Sun 2007c] zählt im Bereich der mobilen Multiagentensysteme und MCSs bereits als de-facto Standard. Die Gründe hierfür sind - neben der allgemein weiten Verbreitung und hohen Akzeptanz - unter anderem, dass die Sprache plattformunabhängig und objektorientiert ist und eine Reihe nützlicher Mechanismen bereits im Sprachumfang enthalten sind, zum Beispiel dynamisches Laden von Klassen, Serialisieren von Daten, Sandbox-Mechanismen usw. [Illmann u. a. 2000, Bouchenak u. a. 2004, Braun und Rossak 2005].

Java wurde als Beispiel einer Technologie, die mobilen Code unterstützt, im Kontext anderer Technologien bereits in Abschnitt 3.1.3 kurz eingeführt. In diesem Abschnitt soll Java weitergehend behandelt werden, da es im Rahmen dieser Arbeit durchweg Verwendung finden wird. In den folgenden Abschnitten werden daher insbesondere die Möglichkeiten der Mobilität, die Java bietet, näher beleuchtet. Auf der einen Seite werden grundlegende Mechanismen für die schwache Codemobilität betrachtet, auf der anderen Seite sollen die Schwierigkeiten der starken Mobilität sowie mögliche Lösungsansätze vorgestellt werden.

Schwache Migration

Nach [Illmann u. a. 2000] wird jegliche Migration als schwach bezeichnet, die nicht als starke Migration anzusehen ist. In Abschnitt 3.1.2 wurde eine Klassifikation der verschiedenen Migrationsaspekte beschrieben, nach denen die durch Java realisierte Möglichkeit der Migration als schwach

anzusehen ist, da sie ausschließlich den Programmcode sowie die Daten einer Anwendung migrieren lässt. Der Ausführungszustand eines Programms (zum Beispiel Prozesse und Befehlszeiger) hingegen bleibt unberücksichtigt [Bouchenak u. a. 2004].

Programmcode-Migration Für das Laden von Programmcode sieht Java das Konzept des dynamischen Ladens mittels eines sogenannten *Classloaders* vor. Die Aufgabe eines solchen Classloaders besteht darin, eine mit Namen angegebene Java-Klasse zu finden und sie der Laufzeitumgebung zur Verfügung zu stellen. Eine typische Strategie ist zum Beispiel die Suche nach der benannten Klasse in Form einer Datei auf dem lokalen System. Es ist aber ebenso möglich eine benannte Klasse bei Bedarf über eine Netzwerkverbindung von einem entfernten System zu laden. Auf diese Weise lässt sich die Programmcode-Migration realisieren [Illmann u. a. 2000, Fuggetta u. a. 1998, Sun 2007d].

Migration der Zustands- und Klassenvariablen Java erlaubt es, Objekte über den Prozess der *Serialisierung* (engl. *serializing*) in eine flache Repräsentation zu transformieren. Das bedeutet, dass die Variablenbelegungen, die den Zustand eines Objektes ausmachen, in einen serialisierten Bytestrom überführt werden. Ein solcher Bytestrom kann dann zum Beispiel über eine Netzwerkverbindung zu einem entfernten System übertragen werden (*member migration*) [Illmann u. a. 2000, Sun 2007a].

Vorgang der Migration Wenn eine Java-Anwendung von einer Ausführungsumgebung in eine andere Ausführungsumgebung migrieren möchte, so werden, ausgehend von einer Basisklasse, alle referenzierten Objekte, die serialisierbar sind, in einen Bytestrom geschrieben. Dieser Bytestrom kann dann über eine Netzwerkverbindung an die andere Ausführungsumgebung übermittelt werden. Dort angekommen werden die einzelnen Objekte sowie deren Zustände wiederhergestellt. Fehlende Klassenbeschreibungen werden bei Bedarf von der Ursprungsplattform über einen Netzwerk-Classloader nachgeladen. Ist die Wiederherstellung abgeschlossen, wird schließlich die Ausführung des Programmcodes vom Anfang des Programms neu angestoßen [Bouchenak u. a. 2004]. Und hier liegt genau der Unterschied zwischen der schwachen und der starken Migration, welche im folgenden Abschnitt beschrieben wird.

Starke Migration

Starke Migration bedeutet, wie sowohl im vorangegangenen Abschnitt als auch in Abschnitt 3.1.2 beschrieben, dass eine Anwendung zu jedem Zeitpunkt migrieren und dabei sowohl ihren Programmcode und die Zustandsdaten der Objekte als auch ihren aktuellen Ausführungszustand, also alle Prozesse samt den zugehörigen Stacks und Befehlszeigern, von einer Ausführungsumgebung in die andere mit nimmt [Illmann u. a. 2000]. Dies ist allein unter Benutzung des Java-Sprachkerns nicht möglich und bedarf bestimmter Erweiterungen entweder des Java-Interpreters oder der Anwendung. Im Folgenden sollen die dabei auftretenden Schwierigkeiten näher betrachtet und verschiedene Lösungsansätze vorgestellt und diskutiert werden.

Schwierigkeiten Java-Anwendungen sind plattformunabhängig, dass bedeutet, dass sie ohne Änderung auf jedem System laufen, für das eine zu Java kompatible virtuelle Maschine implementiert ist. Um die Plattformunabhängigkeit bieten zu können, liegen die Anwendungen in sogenanntem *Bytecode* vor, welche als eine Art Maschinencode für die abstrakte virtuelle Java

Maschine anzusehen ist. Die JVM ist Teil der Java-Umgebung und definiert eine Menge von Befehlen (den Bytecode), eine Ausführungsmaschine (engl. *execution engine*) als Äquivalent eines Hardwareprozessors und sogenannte Laufzeit-Datenbereiche (engl. *runtime data areas*), welche zum Beispiel für die Speicher- und Prozessverwaltung benutzt werden [Bouchenak u. a. 2004].

Die Schwierigkeit, eine starke Migration in Java zu realisieren, liegt darin, dass eine Anwendung, damit sie auf heterogenen Systemen laufen kann, von der JVM interpretiert werden muss. Die JVM ist dafür zuständig den Bytecode der Anwendung in jeweils auf ein spezielles System angepassten, nativen Maschinencode zu übersetzen. Das bedeutet, dass ein Teil der Java-Umgebung abhängig von der darunterliegenden Plattform ist (siehe Abbildung 3.6).

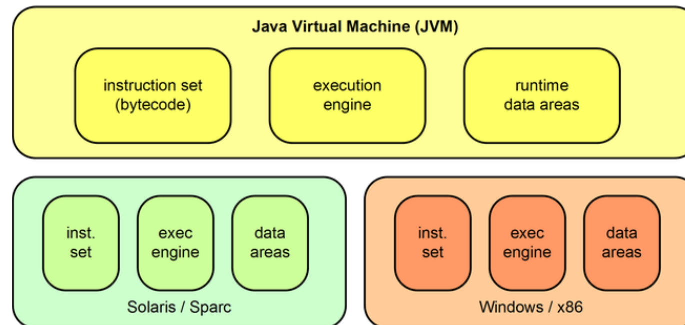

Abbildung 3.6: Architektur der Java-Umgebung (nach [Bouchenak u. a. 2004])

Die JVM gestattet nur äußerst beschränkten Zugriff zu internen und nativen Informationen, sodass es einem Java-Prozess zum Beispiel nicht möglich ist, den Stand des eigenen Befehlszeigers oder den Inhalt des eigenen Aufrufstapels auszulesen. Für eine starke Migration müssen jedoch genau diese Dinge zugreifbar sein, das heißt, man muss deren aktuellen Zustand auslesen, serialisieren und nach einer Migration wieder deserialisieren und den Zustand zur weiteren Ausführung neu setzen können. Diesen Schwierigkeiten kann man auf drei verschiedenen Abstraktionsebenen begegnen: 1) der Ebene des Quelltextes, 2) der des Bytecodes und 3) auf der Ebene der virtuellen Maschine bzw. des Java Interpreters [Illmann u. a. 2000, Bouchenak u. a. 2004]. Alle drei Möglichkeiten sowie ihre individuellen Schwierigkeiten sollen im Folgenden vorgestellt werden, bevor zum Schluss eine Diskussion über deren Vor- und Nachteile folgt.

Quelltext- bzw. Bytecodeänderungen Wie bereits gesagt, ist ein direkter Zugriff auf interne Informationen der JVM bezüglich des Zustandes eines Prozesses nicht möglich. Daher verfolgen manche Projekte den Ansatz, die benötigten Informationen für die starke Migration zur Laufzeit durch die Anwendung selbst mitprotokollieren zu lassen. Hierfür wird entweder der Quelltext oder der Bytecode einer Anwendung zur Übersetzungszeit von einem Präprozessor mit zusätzlichen Anweisungen versehen (engl. *code injection*). Diese zusätzlichen Anweisungen sorgen dafür, dass jeder Prozess alle Zustandsinformationen, die normalerweise auf dem Java-Stapel (engl. *java stack*) verwaltet werden, zusätzlich manuell in einem designierten Java-Objekt protokolliert.

Bei einer Migration kann dieses Java-Objekt über den normalen Serialisierungsprozess mit verschickt werden und in der entfernten Ausführungsumgebung bei Deserialisierung der Objekte gleich die korrekte Aufrufinitialisierung übernehmen. Auf diese Art und Weise können zum Beispiel der Aufrufstapel sowie die lokalen Variablen der aufgerufenen Methoden berücksichtigt werden. Der Befehlszeiger hingegen lässt sich nicht direkt auslesen bzw. setzen [Illmann u. a. 2000, Bouchenak u. a. 2004]. Diese Art und Weise der Prozessmigration wird von einer Reihe von Systemen umgesetzt: *Wasp* [Fuenfrocken 1999] und *Java-*

Go [Sekiguchi u. a. 1999] gebrauchen einen Quelltext-Präprozessor; *Brakes* [Truyen u. a. 2000], *JavaGoX* [Sakamoto u. a. 2000] und *Kalong* [Braun und Rossak 2005] verwenden einen Bytecode-Präprozessor.

Einen Schritt weiter gehen Projekte wie *Javaflow* [Curdt 2007] und *Rife/continuation* [Bevin 2007]. Diese Projekte versuchen den kompletten Zustand eines Prozesses inklusive des Befehlszeigers zu extrahieren um diesen zu einem späteren Zeitpunkt wieder neu setzen zu können. Beide Projekte arbeiten mit sogenannten *Continuations* (dt. Fortsetzung), die unter anderem im Kontext der Webprogrammierung verwendet werden, um einen asynchronen Frage/Antwort-Zyklus, wie er beim HTTP-Protokoll üblich ist, in einen linearen Ablauf umbiegen zu können. Hierzu werden in einem Programm an vorab definierten Stellen Continuation-Objekte erzeugt, die eine Repräsentation des aktuellen Befehlszählers sowie Stack-Informationen enthalten. Zu einem beliebigen späteren Zeitpunkt erlauben diese Continuation-Objekte den Programmlauf an derselben Stelle wieder fortzusetzen.

Da es - wie bereits erwähnt - nicht möglich ist, den Java-Stack und den Befehlszeiger direkt zu manipulieren, reichert zum Beispiel Javaflow den Bytecode eines Programms mit zusätzlichen Informationen derart an, dass bei Wiederaufnahme der Programmausführung über eine Continuation, zuerst die gesamten bis dahin abgelaufenen Programmschritte bis zu der aktuellen Ausführungsposition geschickt übersprungen werden und dann die einzelnen Variablenbelegungen auf dem Java-Stack wieder hergestellt werden [Ortega-Ruiz u. a. 2006].

Erweiterung der JVM Einen anderen Weg geht die Methode der Erweiterung der JVM. Hierbei werden der Java-Umgebung neue Funktionen hinzugefügt, mittels derer sich der komplette Zustand eines Prozesses jederzeit auslesen und in eine serialisierbare Form bringen lässt. Diese serialisierte Form kann nun von einer Ausführungsumgebung zu einer anderen verschickt werden. Dort kann mit symmetrisch arbeitenden Mechanismen ein neuer Thread mit demselben Zustand erzeugt und an derselben Ausführungsposition gestartet werden.

Die Schwierigkeiten bei diesem Ansatz liegen im Detail. Um zumindest die Teile eines Prozesszustandes auszulesen, die innerhalb der JVM verwaltet werden, kann man mittels JNI-basierten Bibliotheken die JVM erweitern und so Zugriff auf Zustandsinformationen erlangen. Zu diesen Zustandsinformationen gehören der Java-*Stack*, der *Heap* und die sogenannte *method area* (siehe Abbildung 3.7). Während Letztere in plattformunabhängigen Formaten vorliegen, ist der Java-Stack meistens eine plattformspezifische Struktur, die in der Programmiersprache C implementiert wurde. Für jede Plattform muss daher ein Übersetzer implementiert werden, der die C-Struktur in ein portables Format überführen und aus diesem auch wieder in eine native Struktur zurückführen kann.

Bei diesem Übersetzungsprozess ergibt sich insbesondere das Problem, dass die Werte in der C-Struktur keinen Typ haben, man also nicht unterscheiden kann, ob ein 4-Byte Wert einen Integer, einen Float oder eine Objektreferenz repräsentiert. Auch hierfür existieren Lösungen, die entweder parallel zum Programmlauf einen zusätzlichen *Typestack* aufbauen (z.B. in *Sumatra* [Acharya u. a. 1997], siehe auch [Qin und Li 2006]) oder an bestimmten Programmpunkten durch Bytecode-Analyse des Kontrollflusses die Werttypen identifizieren (z.B. in *Merpati* [Suezawa 2000] und *JavaThread* [Bouchenak u. a. 2004]).

Komplizierter wird dieses Vorgehen, wenn eine JVM die Ausführungsgeschwindigkeit durch sogenannte *Just-in-Time Kompilierung* (JIT) optimiert. Hierbei werden Java-Methoden, also der Bytecode, dynamisch in nativen Maschinencode übersetzt. Die Ausführung dieser Methoden basiert fortan nicht mehr auf dem Java-Interpreter und dem Java-Stack sondern direkt auf dem

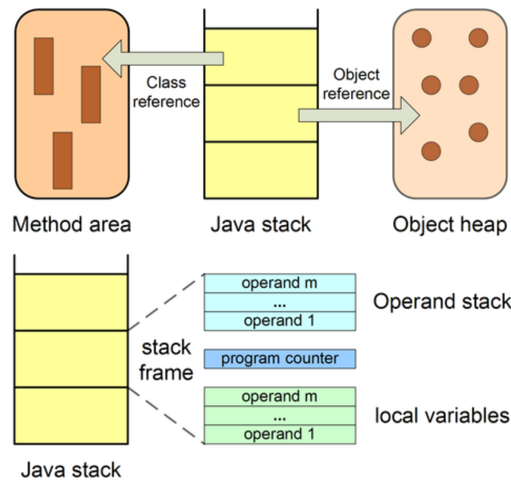

Abbildung 3.7: Java Thread State (nach [Bouchenak u. a. 2004])

Prozessor und einem nativen Stack, wodurch zwar die Ausführung erheblich beschleunigt wird, aber der Zugriff auf den Stack nicht mehr ohne Weiteres möglich ist. In diesem Fall greift man auf dynamische *Deoptimierungstechniken* zurück, welche zur Laufzeit die Informationen auf dem nativen Stack wieder in den Java-Stack integrieren [Bouchenak u. a. 2004].

Diskussion Jede der oben vorgestellten Möglichkeiten hat ihre Vor- und Nachteile. Durch die Erweiterung der JVM kann der komplette Ausführungszustand einer Anwendung oder eines Prozesses gelesen, serialisiert und gesetzt werden. Im Gegensatz dazu kann eine Änderung des Programmcodes durch Hinzufügen weiterer Befehle in den meisten Fällen lediglich einen Teil des Ausführungszustandes berücksichtigen, fällt also eher in den Bereich der schwachen Migration.

Durch eine Erweiterung der JVM kann diese zwar sowohl alle normalen als auch alle mit Prozessserialisierung arbeitenden Programme ausführen, Letztere laufen allerdings nur noch auf speziell erweiterten JVMs und sind nicht mehr allgemein einsetzbar. Durch Änderung des Quelltextes bzw. Bytecodes bleibt die Portabilität hingegen komplett erhalten.

Beiden Ansätzen gemein ist hingegen ein nicht zu vernachlässigender Effizienz-Verlust bei der Programmausführung. Die Anreicherung des Quelltextes bzw. Bytecodes mit weiteren Anweisungen führt zu einer geringeren Ausführungsgeschwindigkeit, da mehr Befehle verarbeitet werden müssen. Bei Erweiterung der JVM hängt es von den jeweils verwendeten Mechanismen ab, ob und wie stark die Effizienz sinkt. Wird zum Beispiel zur Laufzeit ein Typestack verwaltet, müssen auch hier mehr Befehle ausgeführt werden. Gleiches gilt für eine Kontrollflussanalyse, um die Werttypen des Java-Stacks zu ermitteln [Bouchenak u. a. 2004]. Zusammenfassend lassen sich vier Charakteristiken von Prozessserialisierungs-Mechanismen feststellen [Bouchenak u. a. 2004]:

- *Generalität* (engl. *genericity*) der Prozessserialisierung, also die Möglichkeit einen Mechanismus in unterschiedlichen Kontexten (Mobilität, Persistenz etc.) einzusetzen [5]
- *Vollständigkeit* des Prozesszustandes.
- *Portabilität* der Serialisierungsmechanismen über verschiedene Java-Umgebungen hinweg.
- *Effizienz* der Mechanismen, also der Einfluss auf die Leistung der Prozessausführung.

[5] Im Rahmen dieser Arbeit wird die Prozessserialisierung lediglich zum Zwecke der Mobilität eingesetzt. Andere Möglichkeiten werden daher nicht näher behandelt. Siehe hierzu unter anderem [Suezawa 2000, Bouchenak u. a. 2004].

3.1.6 Zusammenfassung

Im Zuge immer größerer, leistungsfähigerer Netzwerke, die nunmehr nicht nur stationär, sondern in Teilen auch mobil, ad-hoc gebildet und ohne feste Infrastruktur sind, wachsen die Anforderungen an die Technologien und Methodologien zur Entwicklung verteilter Anwendungen. Da Zweifel an der Eignung herkömmlicher Techniken aufgeworfen werden, wurden in den vorangegangenen Abschnitten insbesondere Techniken und Paradigmen für den Entwurf von Mobile Code-Systemen betrachtet, die den Anspruch erheben die Probleme der Skalierbarkeit, Konfigurierbarkeit, Anpassbarkeit und Fehlertoleranz verteilter Anwendungen zu lösen.

Es wurde die abstrakte Architektur von Mobile Code-Systemen der Architektur herkömmlicher verteilter Systeme gegenübergestellt sowie unter anderem mit den Begriffen der Ausführungsumgebung und Ausführungseinheit wesentliche Primitive eingeführt. Im Rahmen der Beschreibung der Migration und den damit zusammenhängenden Vorgängen und Entitäten, wurden auch die Mechanismen der Mobilität für Programmcode und Ausführungszustand sowie Mechanismen zur Verwaltung des Datenraumes einer Anwendung erläutert. Die praktische Umsetzung der Konzepte wurde anhand ausgewählter Technologien, unter anderem zum Beispiel Agent TCL und Telescript, betrachtet.

Als Methodologien für die Entwicklung verteilter Systeme wurden eine Reihe von Entwurfsparadigmen, unter anderem das Client/Server-, das Remote Evaluation- und das Mobile Agent-Paradigma beschrieben und deren Einsatz anhand eines allgemeinen Szenarios verdeutlicht. Zum Ende des Grundlagenabschnittes wurde die Programmiersprache Java als de-facto Standard für die Entwicklung von Mobile Code-Systemen vorgestellt und einerseits die Mechanismen der durch Java realisierten schwachen Migration erläutert, andererseits die Probleme und entsprechenden Lösungsmöglichkeiten einer starken Migration in Java betrachtet.

3.2 Mobile Agenten

Eine kurze Übersicht über das Paradigma mobiler Agenten wurde bereits im vorigen Abschnitt gegeben. Nachfolgend soll nun der Begriff eines mobilen Agenten eingeführt und genauer definiert werden. Es werden zwei verschiedene Definitionen aus unterschiedlichen Perspektiven gegeben, die anschließend durch Betrachtung der einen mobilen Agenten konstituierenden Eigenschaften ergänzt werden. Im Anschluss daran werden Vor- und Nachteile mobiler Agenten gegenübergestellt und ein möglicher Nutzen dieses Paradigmas kritisch beleuchtet. Am Ende dieses Abschnitts wird schließlich die Eignung mobiler Agenten speziell im Kontext mobiler adhoc Netzwerke untersucht.

3.2.1 Einführung

Wie in Abschnitt 3.1 bereits erwähnt, beruht das Paradigma der mobilen Agenten auf der Idee mobilen Programmcodes und stellt die bis heute höchste Abstraktionsebene in diesem Bereich dar. Seine Anfänge hatte das Konzept des Verschickens von Programmcodes zur entfernten Auswertung bereits 1969 mit der *Decode Encode Language* [Rulifson 1969] und dem später eingeführten Paradigma der *Remote Evaluation* [Stamos und Gifford 1990] (vergleiche auch Abschnitt 3.1.4). Einen Schritt weiter gehen die *mobilen Objekte*, die neben dem Programmcode auch noch Daten mit sich führen, aber im Gegensatz zu mobilen Agenten nicht proaktiv sind und typischerweise nicht mehr als einmal migrieren. Den jüngsten Vorgänger der mobilen Agenten stellen schließ-

lich die *mobilen Prozesse* dar. Diese führen nicht nur den Programmcode sowie Daten mit sich, sondern auch noch den aktuellen Ausführungszustand, der nach einer Migration die Fortsetzung der zuvor unterbrochenen Programmausführung an derselben Programmposition erlaubt. Dieser Mechanismus bietet allerdings keinen einfachen Weg ausschließlich das Ergebnis einer Berechnung an seine Ursprungsplattform zurück zusenden, ohne dass der gesamte Prozess samt seines kompletten Adressraumes auch zurückkehrt [Wong u. a. 1999, Braun 2003].

Bevor im Folgenden mit den Begriffsdefinitionen für einen mobilen Agenten fortgefahren wird, soll das Entwurfsprinzip anhand eines Nutzungsszenarios in der Praxis verdeutlicht werden. Ein gängiges Beispiel hierfür ist eine Bilddatenbank im Internet. Man stelle sich vor, ein Benutzer möchte in dieser Bilddatenbank nach einem speziellen Bild suchen. In dem Client/Server-Paradigma stellt der Benutzer eine, ggf. auch mehrere Anfragen an die Datenbank und bekommt eine unter Umständen sehr große Menge an Ergebnissen zurück (*data shipping*), die durch einen speziellen Algorithmus lokal auf Seite des Benutzers gefiltert werden muss (vgl. Abbildung 3.8).

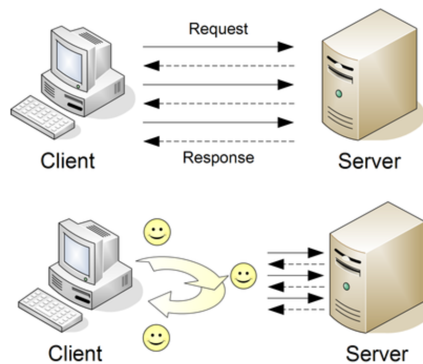

Abbildung 3.8: Client/Server versus mobile Agenten (nach [Pham und Karmouch 1998])

In dem Paradigma mobiler Agenten beauftragt der Benutzer einen Softwareagenten mit der Suche. Dieser verkörpert den Algorithmus und migriert auf den Computer, auf dem auch die Bilddatenbank angesiedelt ist (*code shipping*), kann dort lokal Anfragen an die Datenbank stellen und die Ergebnisse filtern. Schließlich kehrt der Agent mit der bereits gefilterten Ergebnismenge zurück zu seinem auftraggebenden Benutzer und präsentiert ihm eine Auswahl von Ergebnissen [Acharya u. a. 1997, Cabri u. a. 1998, Braun 2003].

Dieses Beispiel zeigt ein typisches Einsatzgebiet für mobile Agenten und verdeutlicht die Vorteile, die mobile Agenten bringen können. Zu diesen Vorteilen gehören unter anderem (vgl. Abschnitt 3.2.4) die Reduzierung der Netzwerklast und hierdurch eine Verringerung der Antwortzeit, eine höhere Verlässlichkeit bei temporären Netzwerkfehlern und schließlich kann dieser Ansatz, speziell bei mobilen Geräten, den Energieverbrauch derselben reduzieren [Braun und Rossak 2005].

3.2.2 Definitionen

Dem Begriff des mobilen Agenten kann man sich von zwei Seiten nähern. Auf der einen Seite steht die künstliche Intelligenz, die den Begriff des Softwareagenten als intelligenten Agenten mit Eigenschaften wie *Autonomie*, *Reaktivität*, *Proaktivität* und *sozialer Kompetenz* formuliert [Wooldridge und Jennings 1995]. Auf der anderen Seite stehen die Softwaretechnik und die verteilten Systeme, die eine eher technische Sicht auf den Begriff einnehmen. Für beide Sichtweisen haben [Braun und Rossak 2005] versucht eine Definition eines mobilen Agenten zu geben.

Aus Sicht der künstlichen Intelligenz wird ein mobiler Agent folgendermaßen definiert:

> *Mobile software agents are computer programs that act as representatives in the global network of computer systems. The agent knows its owner, knows his or her preferences, and learns by communicating with its owner. The user delegates tasks to the agent, which is able to search the network efficiently by moving to the service or information provider. Mobile agents support nomadic users because the agent can work asynchronously while the users is offline. Finally, the agent reports results of its work to the user through different communication channels such as electronic mails, web sites, pagers, or mobile phones.* [Braun und Rossak 2005]

Im Gegensatz zu dieser eher Endbenutzer-zentrierten Definition fokussiert die Definition der Softwaretechnik bzw. der verteilten Systeme die technischen Aspekte eines mobilen Agenten und zielt auf eine niedrigere Ebene der Abstraktion:

> *Mobile agents refer to self-contained and identifiable computer programs, bundled with their code, data, and execution state, that can move within a heterogeneous network of computer systems. They can suspend their execution on an arbitrary point and transport themselves to another computer system. During this migration the agent is transmitted completely, that is, as a set of code, data, and execution state. At the destination computer system, an agent's execution is resumed at exactly the point where it was suspended before.*
> [Braun und Rossak 2005]

Diese zweite Definition hebt nicht so sehr die Eigenschaften eines Agenten an sich hervor, sondern bezieht sich vor allem auf den Aspekt der Mobilität und ist somit auch, bis auf den Punkt der Proaktivität, auf mobile Prozesse anwendbar. Im Rahmen dieser Arbeit wird der Begriff des mobilen Agenten vornehmlich als eine Vereinigung beider Definition aufgefasst. Mobile Agenten haben demnach sowohl die Charakteristiken, die ihnen aus dem Lager der künstlichen Intelligenz angedacht sind, als auch diejenigen, die ihnen seitens der verteilten Systeme zugesprochen werden. Einzig der Aspekt der starken Mobilität (Migration des Ausführungszustandes) wird zu Gunsten der schwachen Mobilität substituiert.

3.2.3 Charakteristiken

Ein mobiler Agent zeichnet sich, neben den Eigenschaften eines herkömmlichen Agenten wie z.B. Autonomie, Proaktivität, Adaptivität etc., durch weitere spezifische Eigenschaften aus [Braun und Rossak 2005, Shiao 2004]:

1. Mobile Agenten müssen eine eindeutige Identität haben. Dieses Kriterium gilt zwar auch für herkömmliche Agenten, ist aber bei mobilen Agenten weniger einfach zu handhaben, da häufig - je nach Implementation - der Name der Ursprungs-Agentenplattform Teil der Identität ist und in verteilten Systemen somit auch eine Eindeutigkeit der Plattformnamen sichergestellt werden muss.

2. Typischerweise werden mobile Agenten in großräumigen und heterogenen Netzwerken eingesetzt. Diese zeichnen sich unter anderem dadurch aus, dass keine Aussagen über die Verlässlichkeit der verbundenen Computer sowie die Stabilität der Verbindungen getroffen werden können. Mobile Agenten müssen daher fähig sein, auf unvorhersehbare Ereignisse, zum Beispiel Abbruch einer Kommunikationsverbindung, entsprechend zu reagieren.

3. Der Prozess der Migration wird von dem mobilen Agenten selbst initiiert (engl. *autonomous mobility*) und nicht - wie etwa bei *Mobile Object*-Systemen - durch das Betriebssystem oder die Middleware.

4. Während *Mobile Object*-Systeme die Migration ausschließlich zum Zwecke der Lastverteilung einsetzen, migrieren mobile Agenten auf andere Systeme auch aus verschiedenen anderen Gründen, z.B. um evtl. nur dort vorhandene Ressourcen in Anspruch zu nehmen.

5. Im Gegensatz zu mobilem Programmcode in den *Remote Evaluation*- und *Code On Demand*-Paradigmen ist es einem mobilen Agenten erlaubt mehrmals zu migrieren. Die Eigenschaft bezeichnet man auch als *multi-hop* Fähigkeit.

6. Die Kontinuität des Agenten muss bei einer Migration sichergestellt werden (engl. *moving continuity*). Wenn ein Agent von einem Gerät zu einem anderen Gerät migriert, muss sein Zustand nach der Migration dem Zustand des Agenten vor der Migration entsprechen.

Die oben aufgeführten Eigenschaften müssen nicht alle zwingend auf mobile Agenten anwendbar sein. Zum Beispiel muss ein mobiler Agent nicht zwangsläufig eine Migration selbst veranlassen können, sondern kann auch reaktiv durch eine Entscheidung der Agentenplattform oder des Benutzers zur Migration veranlasst werden (*implizite* versus *explizite Migration*, vgl. Abschnitt 3.1.2). Ebenso ist die Kontinuität eines Agenten kein konstituierendes Merkmal, da bei der schwachen Migration der Ausführungszustand des Agenten unberücksichtigt bleibt und dieser auf der Zielplattform daher komplett neu gestartet werden muss.

3.2.4 Vor- und Nachteile

Über einen möglichen Nutzen mobiler Agenten wird in der Literatur viel diskutiert. Auf der einen Seite werden sie als die Lösung für eine Vielzahl von Problemen bei dem Entwurf von verteilten Systemen angesehen, auf der anderen Seite stehen die Kritiker, die den Einsatz mobiler Agenten in jeglicher Hinsicht als kontraproduktiv bezeichnen. Festzustellen ist lediglich, dass sich die Entwicklung verteilter Anwendungen unter Verwendung mobiler Agenten noch nicht durchgesetzt hat und weiterhin auf herkömmliche Entwicklungsparadigmen zurückgegriffen wird, sowohl im forschenden als vielmehr auch im industriellen Sektor.

Unter anderem aus diesem Grund können auch keine quantitativen Analysen der Vor- und Nachteile dieses Paradigmas gegeben werden. Eine Bewertung kann im Rahmen dieser Arbeit daher nicht vorgenommen werden. Stattdessen werden die aufgeführten Vorteile als vertretbar angesehen und die Nachteile - insbesondere die Kritik an der Möglichkeit zur Lastreduzierung in einem Netzwerk - als Ausgangspunkt für etwaige Verbesserungen genommen.

Vorteile

Das Paradigma mobiler Agenten entstand aus der Fortführung der Idee mobiler Objekte und mobiler Prozesse, welche sich in der Entwicklung verteilter Systeme bereits etabliert haben. Auch wenn mobile Agenten einen neuen und interessanten Ansatz in der Entwicklung darstellen, müssen deutliche Argumente für ihre Verwendung gegeben werden, bevor sie unter Umständen die herkömmlichen Entwicklungsparadigmen ablösen bzw. ergänzen können. Die im folgenden aufgeführten Vorteile entstammen aus [White 1997, Shiao 2004, Braun und Rossak 2005].

Delegation von Aufgaben Ein Benutzer kann einen Agenten als Stellvertreter für sich benennen und ihm Aufgaben anvertrauen, die der Agent dann selbständig und ohne permanente Kontrolle und ständigen Kontakt mit dem Benutzer ausführen kann. In einer sich immer weiter vernetzenden Welt räumt die Eigenschaft der Mobilität einem Agenten einen größeren Horizont ein und erweitert somit seine Möglichkeiten.

Asynchrone Bearbeitung Sobald ein mobiler Agent initialisiert ist, kann er seine Ursprungsplattform bei nächster Gelegenheit verlassen, um seinen Aufgaben im Netzwerk nachzugehen. Der Benutzer muss hierbei nicht zwangsläufig mit dem Netzwerk verbunden bleiben, sondern kann die Verbindung zu einem beliebigen späteren Zeitpunkt wieder herstellen, um den Agenten, nach Bearbeitung seiner Aufgaben, mit den Ergebnissen wieder entgegen zu nehmen. Dieser Vorteil ist insbesondere im Bereich des *Nomadic Computing* interessant, bei dem die Benutzer selbst mobil sind und Netzwerkverbindungen häufig teuer, unzuverlässig und langsam sind.

Anpassbare Dienstschnittstellen In verteilten Systemen werden die Schnittstellen für Anwendungsdienste häufig als eine Sammlung primitiver Funktionen angeboten, um ihren Klienten eine möglichst allgemein nutzbare Funktionalität zu bieten. Es ist die Aufgabe der Klienten die primitiven Funktionen für die eigenen spezifischen Anforderungen miteinander zu verknüpfen, wodurch sie mehrfach einzelne Anfragen über das Netzwerk verschicken müssen. Ein mobiler Agent, welcher die Anwendungslogik zur Orchestrierung der primitiven Funktionen mit sich führt und somit für den Benutzer eine hochgradig spezialisierte Schnittstelle simuliert, kann das Datenaufkommen im Netzwerk durch lokalen Aufruf der Funktionen auf Anbieterseite reduzieren.

Reduzierung der Netzwerklast Dieser Punkt steht in enger Beziehung zu anpassbaren Dienstschnittstellen und ist der wohl am häufigsten zitierte Vorteil mobiler Agenten. Die Grundidee besteht darin, den Programmcode zur Bearbeitung von Daten in die Nähe der Daten zu bringen, anstatt auf herkömmliche Weise andersherum (*code shipping vs. data shipping*). Hierdurch soll vor allem das Datenaufkommen sowie die Antwortzeiten im Netzwerk reduziert werden, da die Daten schon Vor-Ort bearbeitet, gefiltert und komprimiert werden können.

Dynamische Adaptation Vielfach können mobile Agenten ihre Umwelt wahrnehmen und autonom auf Änderungen der Umweltbedingungen reagieren. So können sie selbst das Ziel und den Zeitpunkt einer Migration entscheiden. Auf diese Weise können sie zum Beispiel ihr nächstes Migrationsziel auf Basis der aktuellen Umweltbedingungen (Netzwerk- oder Serverauslastung) feststellen und so einen für sie passenden Ausführungsort wählen.

Parallele Verarbeitung Ein mobiler Agent kann eine Kaskade von Klonen erstellen, die auf anderen Knoten im Netzwerk laufen, und diesen Teilaufgaben zuweisen, um somit eine parallele und schnellere Bearbeitung zu ermöglichen.

Robustheit und Fehlertoleranz Mobile Agenten sind zwar nicht per se robust und fehlertolerant, allerdings sind dem Entwickler die Eigenschaften wahrscheinlich eher zugegen als bei anderen Paradigmen, da die Umwelt mobiler Agenten in ihrer Natur unbeständig ist und mobile Agenten auf unvorhersehbare Situationen, z.B. Ausfall einer Netzwerkverbindung, vorbereitet sein müssen.

Nachteile

In [Vigna 2004] wurden zehn Gründe für den verhaltenen Einsatz von mobilen Agenten in der Praxis aufgeführt. Unter anderem weil es schwierig sei, mobile Agenten zu entwerfen, zu entwickeln, zu testen und zu kontrollieren sei der breite Einsatz zum Scheitern verurteilt. Außerdem wurde der im vorigen Abschnitt genannte Vorteil, mobile Agenten würden zur Reduzierung des Datenaufkommens im Netzwerk beitragen, in Frage gestellt. Im Folgenden soll lediglich dieser Kritikpunkt näher betrachtet werden, da er im Rahmen dieser Arbeit von besonderer Bedeutung ist und in der Literatur am meisten Beachtung findet[6].

> *„[Mobile agents are a] solution in search of a problem“*
> John Ousterhout, Erfinder der TCL-Skriptsprache

In einer Reihe unterschiedlicher Experimente konnte eine Reduzierung des Datenaufkommens durch mobile Agenten sowohl bestätigt als auch widerlegt werden [Carzaniga u. a. 1997, Strasser und Schwehm 1997, Hagimont und Ismail 1999, Vigna 2004, Braun u. a. 2001, Braun und Rossak 2005]. Die Experimente beruhen meistens entweder auf mathematischen Modellen, mit Hilfe derer Aussagen für Szenarien mit unterschiedlichen Konfigurationen getroffen werden, oder auf der Simulation sowohl eines Netzwerkes als auch des Agentenverhaltens mit unterschiedlichen Simulationsparametern. In der Praxis konnte dieser Vor- oder Nachteil durch aussagekräftige[7] quantitative Evaluierungen bisher weder untermauert noch widerlegt werden.

[Strasser und Schwehm 1997] haben bei der Untersuchung der Leistung mobiler Agentensysteme herausgefunden, dass ein hybrider Ansatz, bei dem sowohl mobile Agenten als auch Client/Server-Kommunikation eingesetzt werden, am performantesten sei. Auch sie verwendeten ein mathematisches Modell, welches unter der Annahme vollständiger Kenntnis der Umwelt eine Migrations- und Kommunikationsstrategie errechnete, die eine optimale Mischung der betrachteten Paradigmen darstellen soll. Die Grundsatzfrage, wann das Paradigma mobiler Agenten anderen Paradigmen bei dem Entwurf verteilter Systeme vorzuziehen ist, hängt von eben dieser vollständigen Kenntnis der Umwelt ab. Hierzu gehören unter anderem die Größe einer Anfrage, die Größe einer Antwort, die Größe des Programmcodes, die Anzahl der involvierten Geräte und je nach Modell auch noch Umweltbedingungen, wie zum Beispiel der Datendurchsatz und die Latenzzeit des Netzwerkes. In allen theoretischen Untersuchungen wurden diese Parameter von vornherein statisch festgelegt, was aber nach [Braun und Rossak 2005] nicht unbedingt

[6] Für eine weitergehende Betrachtung der anderen Kritikpunkte sei unter anderem auf [Vigna 2004] und darin referenzierte Quellen verwiesen

[7] [Hagimont und Ismail 1999] haben ihre Experimente unter nahezu praktischen Randbedingungen im Internet durchgeführt und dabei zwei selbst entwickelte Testanwendungen für den Leistungsvergleich zwischen Client/Server-Aufrufen und mobilen Agenten herangezogen. Doch auch diese Untersuchung kann lediglich als Indikator angesehen werden, da sie keinen Anspruch auf Allgemeingültigkeit erhebt.

von der Theorie in die Praxis übertragbar sei. Außerdem konzentrieren sich die Untersuchungen ausschließlich auf Leistungsaspekte und berücksichtigen nicht den Vorteil der asynchronen Bearbeitung im Falle von unverlässlichen Netzwerkverbindungen [Braun und Kern 2005].

3.2.5 Optimierung der Migrationseffizienz

Als Gründe, warum mobile Agenten - verglichen mit anderen Paradigmen - ein höheres Verkehrsaufkommen im Netzwerk verursachen, haben [Braun und Rossak 2005] in diversen Experimenten drei wesentliche Punkte ausgemacht, die alle auf die sehr einfachen Migrationstechniken[8] der untersuchten Systeme zurückzuführen sind. Hierzu gehören:

1. Der Programmcode eines Agenten ist zumeist umfangreicher als eine einfache Client/Server-Anfrage und verursacht bei jeder Migration feste Kosten hinsichtlich der zu übertragenden Datenmenge.

2. Werden push-Strategien für die Übertragung eingesetzt, so wird normalerweise der gesamte Programmcode eines Agenten an eine entfernte Plattform übertragen (engl. *push-all-to-next*), auch wenn dort nur Teile verwendet werden. Ähnliches gilt für Systeme, die ihre pull-Strategien über den Standard-Classloader sowie die Serialisierungsmechanismen von Java realisieren. Auch hier werden Klassen übertragen, die unter Umständen überhaupt keine Verwendung finden [Braun u. a. 2005]. Ein weiteres Problem stellt der in Java integrierte Codespeicher für Klassen dar, der zum einen unterschiedliche Versionen einer Klasse nicht unterscheiden kann und zum anderen oftmals nicht auf Ebene der Plattform, sondern auf Ebene einer einzelnen Agenteninstanz[9] arbeitet, was dazu führen kann, dass einzelne Klassen überflüssigerweise mehrfach übertragen werden.

3. Die Daten eines Agenten werden bei jeder Migration als eine Einheit mitgeführt, was wiederum auf den Einsatz der Java-eigenen Serialisierungsmechanismen zurückzuführen ist[10]. Dies kann zum Beispiel dazu führen, dass ein Agent zusammen mit Ergebnisdaten auf andere Plattformen migriert, obwohl diese Daten erst wieder auf seiner Ursprungsplattform verwendet werden.

Um diese Probleme zu bewältigen, gibt es eine ganze Reihe von Möglichkeiten, um die *Migrationseffizienz* mobiler Agenten zu erhöhen. Die Migrationseffizienz wird hierbei folgendermaßen definiert:

> „*The migration efficiency of a mobile agent defines how many code units (on the level of statements, methods, or classes) and data units (variables or objects) of an agent have been used (read or written) on remote agencies proportional to the number of code units and data units that have been transmitted.*" [Braun u. a. 2005]

[8] Ein Großteil der Systeme implementiert lediglich einfache *pull*-Strategien (über den Standard-Classloader von Java) oder die *push-all-to-next*-Strategie (vgl. Abschnitt 3.1.2)
[9] zum Beispiel bei Grasshopper [Magedanz u. a. 1999]
[10] Ein anschauliches Beispiel hierfür findet sich in [Braun und Rossak 2005, Seite 99ff.].

Demnach besitzt ein Agent eine niedrige Migrationseffizienz, wenn viele Programm- oder Dateneinheiten von diesem an dem entfernten Ausführungsort nicht benutzt und entsprechend überflüssigerweise übertragen wurden. Um die Migrationseffizienz zu erhöhen, gilt es Techniken zu entwickeln, die nur den Teil des Programmcodes und der Daten transferieren, die am Bestimmungsort mit hoher Wahrscheinlichkeit benötigt werden. Zusätzlich gibt es noch weitere Möglichkeiten, um den Vorgang der Migration, abgesehen von der zu übertragenden Datenmenge, zu beschleunigen. Hierbei werden die Mechanismen optimiert, die bei den einzelnen Schritten des Vorgangs wirken. Auch diese Möglichkeiten werden im Folgenden kurz vorgestellt.

Kernpunkte der Leistungsoptimierung

Die Entscheidung, in welcher Situation eine Migration sinnvoller als ein entfernter Prozeduraufruf ist, bezeichnen [Braun u. a. 2005] als das Migrationsentscheidungs-Problem (engl. *migration decision problem*). Wie Versuche in [Braun u. a. 2001] gezeigt haben, kann dieses Problem nicht bereits während der Entwurfs-, Implementations- oder der sogenannten *Deployment*[11]-Phase entschieden werden, da es von inhärent dynamischen Parametern abhängt, die erst zur Laufzeit verfügbar sind (z.B. Netzwerkauslastung, Größe einer Anfrage/Antwort, Ausführungswahrscheinlichkeit von Programmteilen etc.). Deshalb sollte das Problem in geeigneter Weise durch einen mobilen Agenten zum Zeitpunkt der Migration auf Basis der tatsächlichen Parameterwerte entschieden werden. Braun u.a. bezeichnen dies als *adaptive transmission of code and data* [Braun u. a. 2005]. Im Folgenden werden daher verschiedene Möglichkeiten - unterteilt in Laufzeit- und Übertragungsaspekte (siehe auch Abbildung 3.9) - vorgestellt, mit Hilfe derer die Migrationseffizienz gesteigert werden kann.

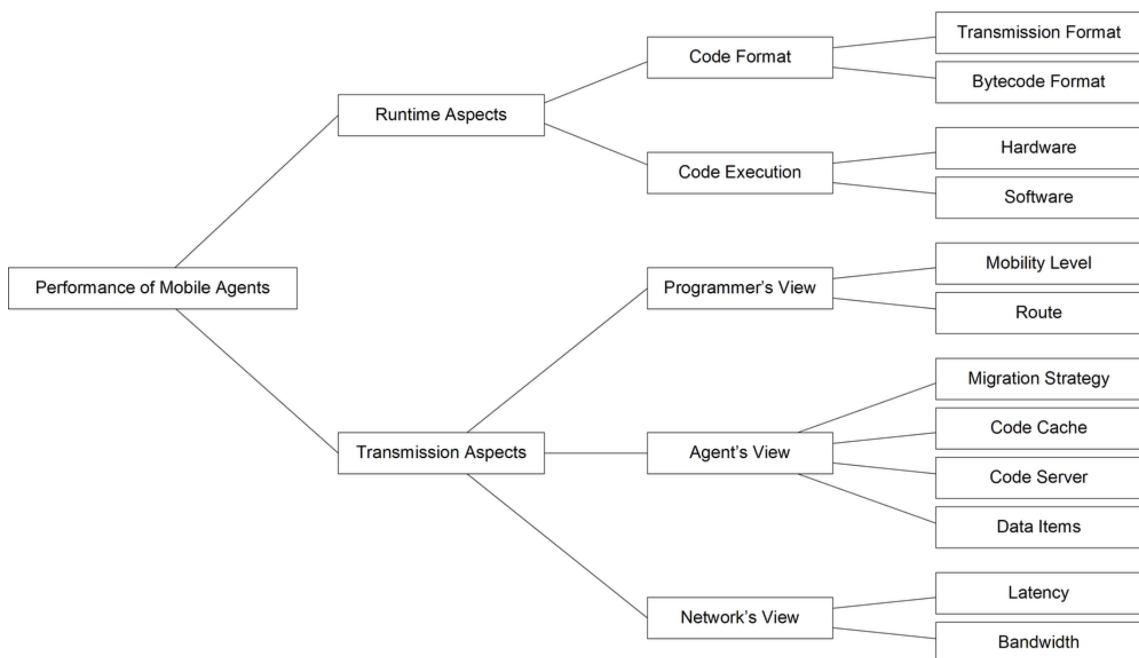

Abbildung 3.9: Klassifikation von Kernpunkten der Leistungsoptimierung bei mobilen Agenten (nach [Braun und Rossak 2005])

[11] Als *Deployment* bezeichnet man die Prozesse zwischen der Akquise und der Ausführung von Software [OMG 2003]

Laufzeitaspekte Zu den Laufzeitaspekten gehören Techniken, die die Ausführung eines Agenten insgesamt beschleunigen. Dies wirkt sich zwar nicht direkt auf die Migrationseffizienz aus, kann aber die Zeit für einen Migrationsvorgang verkürzen, da einzelne Schritte des Migrationsvorgangs optimiert werden. Zwei wichtige Aspekte sind hierbei das Format und die Ausführung des Programmcodes [Braun und Rossak 2005].

Das Format beeinflusst die Größe des Programmcodes und damit die Ausführungsgeschwindigkeit, indirekt somit auch die Dauer eines Migrationsvorgangs. Man unterscheidet zwischen Quelltexten, Zwischenformaten (z.B. den Bytecode von Java) und Maschinencode. Eine Agentenrepräsentation in Maschinencode eignet sich nicht für die Migration, da Agenten meist in heterogenen Umgebungen ausgeführt werden und Maschinencode spezifisch für bestimmte Prozessorarchitekturen ist. Übertragung des Quelltextes eines Agenten ist ebenso wenig sinnvoll, da Quelltexte im Vergleich zu Zwischenformaten meist größer sind und darüber hinaus auf jeder Plattform erst in ausführbaren Programmcode umgewandelt werden müssen. Die beste Lösung ist die Übertragung von Programmcode in einem Zwischenformat, da Zwischenformate meist kompakter als Quelltexte und im Gegensatz zu Maschinencode plattformübergreifend zu verwenden sind sowie bereits architekturunabhängige Codeoptimierungen enthalten können.

Die Ausführungsgeschwindigkeit eines Agenten wird auf der einen Seite durch die Hardware (Prozessor, Speicher etc.) des Systems bestimmt, auf der anderen Seite durch die Software, also das Betriebssystem, die Agentenplattform und - im Falle von Zwischenformaten - die virtuelle Maschine. An dieser Stelle sollen nur zwei Möglichkeiten erwähnt werden, um die Ausführungsgeschwindigkeit auf Ebene der Agentenplattform zu erhöhen. Hier obliegt es dem Entwickler der Plattform, die Mechanismen durch entsprechende Programmierung zu optimieren. So sind zum Beispiel die Serialisierungsmechanismen von Java verhältnismäßig langsam und ein Ansatzpunkt zur Verbesserung. Hierdurch ließen sich insbesondere die ohnehin zeitkritischen Phasen der Serialisierung und Deserialisierung eines Agenten beschleunigen. Eine weitere Möglichkeit stellen sogenannte Prozesspools (engl. *thread pools*) dar. Diese Sammlungen bestehen aus einer festen Anzahl bereits im Vorwege erzeugter Prozesse, denen je nach Bedarf eine Aufgabe (z.B. die Ausführung eines Agenten) zugewiesen wird. Bearbeitet ein Prozess eine Aufgabe, so wird er aus der Sammlung entfernt und nach Abarbeitung schließlich wieder hinzugefügt. Da das Erzeugen neuer Prozesse zeitaufwendig ist, lässt sich über solche Prozesssammlungen die Zeit vom Eintreffen eines Agenten auf einer Plattform bis zum Starten des Agenten verkürzen [Braun und Rossak 2005].

Übertragungsaspekte Viel wichtiger als die Laufzeitaspekte sind die Übertragungsaspekte, da diese sich direkt auf die Migrationseffizienz auswirken. Zu diesen Aspekten gehören alle Techniken, die die Netzwerkbelastung und die Übertragungszeit während einer Migration beeinflussen [Braun und Rossak 2005]. Aus der Sicht eines Programmierers ist der wichtigste Faktor die unterstützte Ebene der Mobilität. Wie bereits erwähnt, unterscheidet man hier zwischen schwacher und starker Mobilität. Schwache Mobilität ist relativ einfach und schnell in Java zu realisieren. Starke Mobilität hingegen wird von Java nicht nativ unterstützt und muss nachträglich in ein System eingebaut werden (vgl. Abschnitt 3.1.5). Hierbei wird der Programmcode durch zusätzliche Informationen angereichert, was zu einem höheren Übertragungsvolumen und einer längeren Ausführungszeit führt. Ist man also nicht unbedingt auf die starke Migration angewiesen, so sollte man dieser aus Gründen der Performanz die schwache Migration vorziehen.

Ein weiterer Punkt ist die Optimierung der Route, entlang derer ein Agent migriert. Idealerweise kennt ein Agent die Ressourcen einer Plattform bereits im Vorwege und kann seine Route, anhand der Verfügbarkeit von Ressourcen, eigenständig planen. Andernfalls müsste er blind von einer Plattform zur nächsten migrieren, um dort gegebenenfalls festzustellen, dass zur Verrichtung seiner Aufgabe die notwendigen Ressourcen fehlen. Eine unkomplizierte Möglichkeit ist die Reduzierung des Übertragungsvolumens durch Komprimierungstechniken. Diese können sowohl den Programmcode als auch den Zustand eines Agenten verlustfrei vor dem Senden komprimieren und auf der empfangenden Plattform wieder entpacken. Hierbei ist jedoch zu beachten, dass der Einsatz von Komprimierungstechniken die Dauer einer Migration auch erhöhen kann, da der Vorgang der Komprimierung rechen- und somit zeitaufwendig ist und je nach Art der Daten unter Umständen nur eine minimale Verdichtung erzielt werden kann.

Um das Übertragungsvolumen weiter zu verringern, gilt es außerdem Verfahren einzusetzen, die die überflüssige Übertragung von Programmcode vermeiden. Da Programmcode im Allgemeinen zur Laufzeit nicht geändert wird, muss bereits im Vorwege übertragener Code nicht wiederholt an dieselbe Plattform übertragen werden. Außerdem sollte es vermieden werden Programmteile zu übertragen, die ein Agent auf der entfernten Plattform ohnehin nicht benötigt. Dies lässt sich zum Beispiel durch intelligente Migrationsstrategien erreichen, die zur Laufzeit erstellt werden (wie in Abschnitt 3.3.2 näher beschrieben). So ließe sich anhand von Ausführungswahrscheinlichkeiten einzelner Programmteile festlegen, welche Teile des Agenten auf jeden Fall übertragen und welche Teile ggf. erst später nachgeladen werden müssen. Eine weitere Möglichkeit ist die Verwendung eines Codespeichers (engl. *code cache*), der einmal erhaltenen Programmcode speichert, sodass dieser in Zukunft nicht wiederholt übertragen werden muss [Braun und Rossak 2005].

Der letzte Punkt betrifft die Handhabung der Daten eines Agenten. Daten, die auf einer Plattform nicht benötigt werden, sollten auch nicht an diese Plattform übertragen, sondern stattdessen an die Ursprungsplattform des Agenten zurückgesandt werden können. Ebenso sollte es möglich sein, Daten, die nicht auf jeder Plattform benötigt werden, erst zur Laufzeit von der Ursprungsplattform nachzuladen, um zu vermeiden, dass der Agent auf seiner Route zu viel Datenballast mit sich führt. Abschnitt 3.3.2 wird sich weitergehend mit diesen Problemen beschäftigen und ein Mobilitätsmodell vorstellen, welches versucht, durch den Einsatz diverser oben beschriebener Techniken, die Migrationseffizienz mobiler Agenten zu erhöhen.

3.2.6 Mobile Agenten und MANETs

Wie bereits erwähnt, stellen mobile ad-hoc Netzwerke (MANETs) besondere Anforderungen an Applikationen. Teilnehmer des Netzwerkes können unvorhersehbar dauerhaft oder temporär aus dem Netzwerk ausscheiden und neue Teilnehmer können jederzeit dem Netzwerk beitreten. Die Netzwerkverbindungen zwischen den Teilnehmern sind weniger beständig und haben eine deutlich geringere Datenrate und eine höhere Nachrichtenlaufzeit als in Infrastrukturnetzwerken. Mobile Geräte, die häufig Bestandteil eines solchen Netzwerkes sind, verfügen meist über eingeschränkte Ressourcen (Prozessor, Speicher, Netzwerkschnittstellen, Energie etc.) und die Verbindung mit dem Netzwerk kann unter Umständen - zum Beispiel im Falle einer GPRS- oder UMTS-Verbindung - sehr kostspielig sein.

Aufgrund der Eigenschaften mobiler Agenten, bietet sich dieses Paradigma besonders für den Einsatz in derartig heterogenen und dynamischen Netzen an. Insbesondere die Eigenschaften der Autonomie und der Mobilität privilegieren einen Agenten für die verteilte oder entfernte Bearbei-

tung seiner Aufgaben in einem MANET, da er hierdurch, ohne Rücksprache mit seinem Benutzer, Entscheidungen anhand der lokal herrschenden Gegebenheiten treffen und seinen Ausführungsort bei Bedarf wechseln kann [Fuggetta u. a. 1998, Lawrence 2002b, Berger und Watzke 2002, Weyns u. a. 2004].

Den langsamen und unter Umständen teuren Verbindungen können mobile Agenten durch die Möglichkeit der Reduzierung des Netzwerkverkehrs entgegnen. Dass dieser Vorteil nicht in allen Szenarien gültig ist, wurde bereits im vorhergehenden Abschnitt erläutert. Allerdings haben [Braun und Rossak 2005] festgestellt, dass dieser Vorteil umso stärker wirkt, wenn man lediglich die Netzwerkschnittstelle des Klienten betrachtet. Wenn zum Beispiel ein Klient mit seinem mobilen Gerät über eine GPRS-Verbindung[12] an dem MANET teilnimmt, so gilt es die zu übertragenden Daten bzw. die Kosten an seiner Netzwerkschnittstelle zu reduzieren, das Datenaufkommen in dem übrigen Teil des Netzwerkes ist aus Klientensicht nicht weiter interessant.

Zusätzlich können wichtige Ressourcen mobiler Geräte, insbesondere Energie, entlastet werden, wenn die Bearbeitung von Aufgaben in einem MANET an andere Teilnehmer delegiert wird. Unter Umständen ist die Ausführung eines Agenten auf einem bestimmten mobilen Gerät, zum Beispiel aufgrund fehlender Hard- oder Software-Ressourcen, auch überhaupt nicht möglich. Auch in diesem Fall profitiert eine Anwendung von der Möglichkeit entsprechende Programmteile in Form eines mobilen Agenten zur Bearbeitung an besser ausgestattete Geräte zu übermitteln und später lediglich die Ergebnisse der Bearbeitung wieder empfangen zu müssen [Shiao 2004].

In diesem Fall ist es noch nicht einmal nötig, dass der Benutzer die Verbindung zum Netzwerk aufrecht erhält, sei es aus Kostengründen oder aufgrund instabiler Netzwerkverbindungen. Durch die asynchrone Bearbeitung der Aufgabe, kann er sich jederzeit aus dem Netzwerk entfernen. Tritt er dem Netzwerk später erneut bei, kann er den Agenten nach erledigter Arbeit wieder auf seinem Gerät empfangen [Cabri u. a. 1998, Braun und Rossak 2005, Shiao 2004]. Auch hinsichtlich der Fehlertoleranz, ist die asynchrone Bearbeitung ein wichtiger Punkt, da von vornherein nicht mit einer ständigen Verbindung gerechnet werden kann [Fuggetta u. a. 1998].

Alle bisher genannten Punkte setzen voraus, dass ein mobiler Agent seine Umwelt in irgendeiner Art und Weise wahrnehmen muss, was ohnehin eine der grundlegenden Eigenschaften von Agenten darstellt. Zu dieser Wahrnehmung gehören idealerweise Netzwerkparameter (zum Beispiel Qualität, Kosten, Durchsatz und Latenzzeit), andere erreichbare Plattformen innerhalb des Netzwerkes (entweder direkte Nachbarn oder die gesamte Topologie) sowie nutzbare Ressourcen (Hardware, Dienste etc.) auf diesen Plattformen.

Wie in Abschnitt 3.2.4 bereits erwähnt, lassen sich auf Basis solcher Beobachtungen nahezu optimale Migrationsstrategien errechnen, die wiederum den Aspekt der Reduzierung des Netzwerkverkehrs durch mobile Agenten bestärken. Hierzu werden im folgenden Abschnitt Mobilitätsmodelle vorgestellt, die einen Großteil der technischen Aspekte eines Migrationsvorgangs allgemein beschreiben, und auf Grundlage nahezu vollständigen Wissens über die Umwelt mittels verschiedener Techniken versuchen, die Migrationseffizienz mobiler Agenten zu steigern.

[12] Bei GPRS-Verbindungen wird von dem Mobilfunkanbieter in der Regel das Verkehrsvolumen und nicht die Dauer der Verbindung berechnet

3.2.7 Zusammenfassung

Das Paradigma mobiler Agenten stellt nach den mobilen Objekten und den mobilen Prozessen die bisher weitestgehende Abstraktion bezüglich der Mobilität von Programmcode und Daten dar. Die Eigenschaften mobiler Agenten adaptiv, autonom und proaktiv ihren Ausführungsort im Netzwerk zu wechseln, prädestiniert sie für den Einsatz in hochgradig dynamischen Umgebungen, wie zum Beispiel mobilen ad-hoc Netzwerken. Doch bestehen auch grundlegende Zweifel an den Vorteilen des Paradigmas gegenüber anderen Paradigmen, wie zum Beispiel dem Client/Server-Paradigma.

Eine Reihe von Untersuchungen haben gezeigt, dass zwar die mobilen Agenten zugesprochenen Vorteile in der Theorie Bestand haben, die Umsetzung des Paradigmas in bestehenden Systemen jedoch verbesserungswürdig ist. Insbesondere die Annahme, mobile Agenten würden zur Reduzierung des Datenvolumens im Netzwerk beitragen, indem sie den Programmcode zu den Daten bringen anstatt andersherum, ist stark umstritten. Um diesen Kritikpunkt zu entkräften, wurden eine Reihe von Mechanismen vorgestellt, die die Migrationseffizienz mobiler Agenten erhöhen sollen. Mit Hilfe dieser Mechanismen soll einerseits die Ausführung eines Agenten beschleunigt, andererseits aber auch die Menge an Daten reduziert werden, die bei einer Migration über das Netzwerk transferiert werden müssen. Die praktische Umsetzung dieser Mechanismen sowie das Konzept von Mobilitätsmodellen werden im nachfolgenden Abschnitt erläutert.

3.3 Mobilitätsmodelle

In den vorhergehenden Abschnitten 3.1 und 3.2 wurden bereits die Grundlagen der Codemobilität sowie das Paradigma mobiler Agenten mit seinen Vor- und Nachteilen näher erläutert. Im Zuge dessen wurden eine ganze Reihe verschiedener Möglichkeiten der Migration identifiziert. Alle diese Möglichkeiten beziehen sich im Wesentlichen auf die Frage, wie ein Agent migrieren soll und welche Teile von einem Agenten bei der Migration transferiert werden sollen. Bevor in Kapitel 4 auf die nächste essenzielle Frage eingegangen wird, wann ein Agent migrieren soll, wird im Folgenden der Begriff eines Mobilitätsmodells erläutert, mit Hilfe dessen sich die wichtigsten technischen Aspekte einer Migration beschreiben lassen.

Mit dem Kalong-Modell wird außerdem ein konkretes Mobilitätsmodell näher vorgestellt, welches durch eine Vielzahl von Mechanismen die Nachteile einfacher Migrationstechniken zu umgehen versucht, indem es eine Reihe von Techniken verwendet, die in Abschnitt 3.2.5 bereits als Lösungsansätze zur Optimierung des Migrationsvorgangs beschrieben wurden. Im Anschluss daran wird eine formale Beschreibungssprache für den Prozess der Migration vorgestellt mit deren Hilfe sich die unterschiedlichen Möglichkeiten der einzelnen Migrationsschritte beschreiben lassen. Außerdem können mittels dieser Sprache konkrete Implementierungen mobiler Agenten bzw. Agentenplattformen hinsichtlich ihrer Migrationstechniken verglichen und klassifiziert werden. Zur Veranschaulichung wird hierfür abschließend ein konkretes Mobilitätsmodell beschrieben, welches von der *Aglets*-Agentenplattform [Lange und Oshima 1998] implementiert wird.

3.3.1 Einführung

Mit Hilfe eines Mobilitätsmodells (engl. *mobility model*) lassen sich fast alle wichtigen technischen Aspekte der Migrationsmechanismen einer Agentenplattform beschreiben. Weitergehend ist es möglich, über ein solches Modell unterschiedliche Migrationstechniken von verschiedenen

Agentenplattformen zu vergleichen und zu klassifizieren. In einem Mobilitätsmodell sind drei verschiedene Sichten definiert, wobei jede Sicht genau zwei von sechs Phasen der Migration umfasst. Die sechs Phasen der Migration sowie die einzelnen Sichten (nach [Braun und Rossak 2005]) sind in Abbildung 3.10 dargestellt.

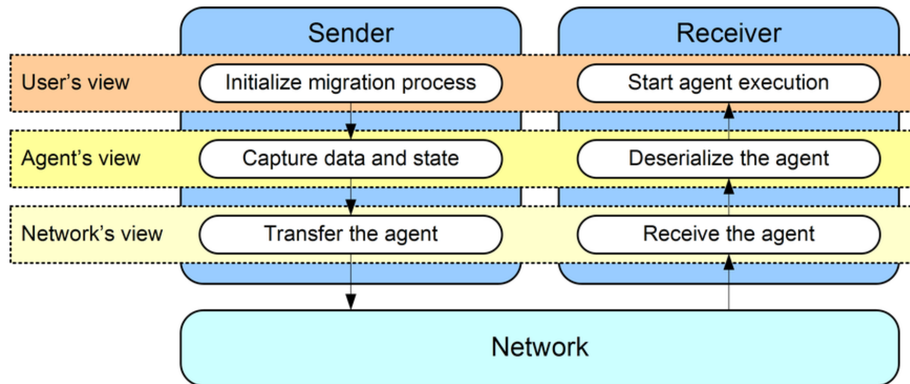

Abbildung 3.10: Die sechs Phasen des Migrationsprozesses und die drei Sichten des Mobilitätsmodells (nach [Braun und Rossak 2005])

Die Benutzersicht (engl. *user's view*) umfasst alle Belange der Benutzungsschnittstelle der Migration. Hierzu zählen im Einzelnen der Migrationsbefehl, die Technik, mit der die Ausführung eines Agenten an entfernter Stelle fortgesetzt wird, und die Adressierung des Migrationsziels. Die zweite Sicht, die Sicht des Agenten (engl. *agent's view*), fokussiert den Aspekt der Daten- und Programmcodeverlagerung. In dieser Ebene sind die Migrationsstrategien untergebracht, die beschreiben, welcher Teil des Programmcodes wohin verschickt werden soll und wie die Daten zu handhaben sind. Schließlich beschreibt die unterste Ebene die Sicht des Netzwerkes (engl. *network's view*) und die Technik zum Übertragen des serialisierten Agenten über das Netzwerk.

Bei der Beschreibung von Lösungsansätzen (vgl. Abschnitt 3.2.5) für das bei der Migration von mobilen Agenten verursachte hohe Datenvolumen, wurden diese drei Sichten bereits kurz eingenommen und daraus Maßnahmen zur Optimierung des Migrationsprozesses abgeleitet (siehe Abbildung 3.9). Bevor in Abschnitt 3.3.3 eine formale Beschreibung der drei Sichten mittels der Mobilitätssprache folgt, wird im folgenden Abschnitt das *Kalong*-Mobilitätsmodell vorgestellt, welches konkrete Optimierungskonzepte für alle drei Sichten beinhaltet.

3.3.2 Das Kalong-Modell

Das *Kalong*-Mobilitätsmodell wurde an der Universität Jena entwickelt und erlaubt Agenten eine schwache Migration mittels adaptiver Migrationsstrategien. Diese Strategien können von einem Agenten zur Laufzeit vor jeder Migration flexibel und sehr genau unter Berücksichtigung von Umweltparametern festgelegt werden und beschreiben für jedes Programmfragment des Agenten, auf welche Weise es übertragen werden soll [Braun 2003, Braun und Kern 2005, Braun und Rossak 2005, Braun u. a. 2005, Kern und Braun 2005]. Motiviert wurde die Entwicklung von Kalong durch eine Analyse der Nachteile mobiler Agenten gegenüber dem Client/Server-Paradigma, der Grundsatzfrage in welchen Fällen das Paradigma mobiler Agenten anderen Paradigmen vorzuziehen ist (bezeichnet als *Migrationsentscheidungs-Problem*, vgl. Abschnitt 3.2.5) und wie die Effizienz der Migration als Kernkompetenz mobiler Agenten weiter gesteigert werden könnte [Kern und Braun 2005, Braun u. a. 2005].

Anstatt das Migrationsentscheidungs-Problem zu lösen, setzen Braun u.a. den Fokus des Problems neu. Sie empfehlen, mobile Agenten in jedem Fall einem hybriden Ansatz (siehe [Strasser und Schwehm 1997]) vorzuziehen, bei dem (statisch oder dynamisch) zwischen entfernten Prozeduraufrufen und Programmcode-Migration entschieden wird (vgl. Abschnitt 3.2.4). Der Grundgedanke hierbei ist, dass entfernte Prozeduraufrufe durch Agentenkommunikation ersetzt werden und die Migration von Agenten dann zum Einsatz kommt, wenn durch sie die Leistung einer Anwendung gesteigert werden kann [Braun u. a. 2005].

Die in Abschnitt 3.2.4 aufgestellte Behauptung, mobile Agenten würden im Vergleich zu anderen Paradigmen ein höheres Datenaufkommen im Netzwerk produzieren, führten Braun u.a. im Wesentlichen auf die sehr einfachen und statischen Migrationstechniken zurück, die bei der Migration in existierenden Agentenplattformen Anwendung finden. Zum Beispiel ist es in *Aglets* [Lange und Oshima 1998] nicht möglich zu entscheiden, welche Klassen mittels einer push- und welche mittels einer pull-Strategie übertragen werden sollen und *Grasshopper* [Magedanz u. a. 1999] unterstützt sogar ausschließlich nur pull-Strategien [Braun und Rossak 2005]. Ein mathematisches Modell, mit Hilfe dessen Braun u.a. eine Analyse der Netzwerklast durchführten, untermauert diese Feststellung [Braun und Rossak 2005]. Als Konsequenz dieser Analyse entwickelten sie eine Reihe von Mechanismen, die eine strategiebasierte Migrationsplanung erlauben. Diese Mechanismen bilden die Grundlage für das Kalong-Mobilitätsmodell und beruhen darauf, dass:

1. die Entscheidung, ob ein Agent entfernt kommunizieren oder migrieren soll, adaptiv von diesem zur Laufzeit anhand von Umweltparametern getroffen wird (auch als *adaptive decision between communication and migration* bezeichnet [Braun u. a. 2005])
2. im Falle der Migration der Agent, ebenfalls abhängig vom Kontext, erst zur Laufzeit entscheidet, welche Mechanismen er bei der Migration einsetzen möchte (von Braun u.a. als *adaptive transmission of code and data* bezeichnet [Braun u. a. 2005]).

Das Kalong-Mobilitätsmodell widmet sich insbesondere dem zweiten Aspekt und ermöglicht es mobilen Agenten zur Laufzeit sowohl auf der Basis erlernten Wissens über die Anwendung selbst (z.B. Ausführungswahrscheinlichkeit bestimmter Programmteile) als auch durch Beobachtung ihrer Umwelt eine Entscheidung zu fällen, *wie* bzw. unter Zuhilfenahme welcher Mechanismen migriert werden soll. Das *Wie* bezieht sich hierbei auf die *Migrationsstrategie*, welche nach [Braun u. a. 2005] als die Art und Weise wie Code und Daten von einem Ort an einen anderen Ort verschoben werden, definiert wird.

Mechanismen zur Optimierung der Migrationseffizienz

In Abschnitt 3.2.5 wurden bereits einige Lösungsansätze zur Optimierung des Migrationsvorgangs in allgemeiner Form beschrieben. Diese Ansätze werden von dem Kalong-Modell konkret ausgestaltet und resultieren in einer Reihe von Mechanismen, die die Migrationseffizienz mobiler Agenten steigern sollen. Hierbei werden ausschließlich die Übertragungsaspekte und nicht die Laufzeitaspekte berücksichtigt (vgl. Abbildung 3.9).

Externer Zustand Kalong definiert eine neue Repräsentation eines Agenten. Ein Agent besteht nunmehr nicht nur aus Programmcode und seinem Objektzustand, sondern zusätzlich noch aus einem *externen Zustand*, welcher beliebige Datenelemente enthalten kann [Braun u. a. 2005, Braun und Rossak 2005].

Zu jeder Zeit kann ein Agent selbständig entscheiden, welche Datenelemente er als Teil seines externen Zustandes definiert. Wenn ein Agent migriert, werden die Elemente des externen Zustands, im Gegensatz zu den Elementen des Objektzustandes, nicht automatisch mit transferiert. Befindet sich ein Agent zum Beispiel auf seiner Ursprungs- bzw. Heimatplattform, so kann er dort Teile seines externen Zustandes hinterlegen. Befindet er sich hingegen auf einer entfernten Plattform, so kann er, falls eine Verbindung zur Ursprungsplattform besteht, ebenfalls Elemente des externen Zustandes an diese zur Verwahrung übersenden, andernfalls muss er sie weiter mit sich führen. Andersherum kann sich ein Agent benötigte Elemente, die als Teil seines externen Zustandes auf seiner Ursprungsplattform liegen, jederzeit auf seine aktuelle Plattform übertragen lassen. Hierdurch ergeben sich gleich mehrere Vorteile:

1. Der Agent muss Daten nicht ständig bei sich führen, sondern kann sie bei Bedarf nachladen bzw. wenn kein Bedarf mehr an diesen besteht, diese wieder zurück zu seiner Heimatplattform übermitteln.

2. Dadurch, dass der Agent in seiner serialisierten Form weniger Daten mit sich führt, muss unter Umständen auch weniger Programmcode für den Agenten übertragen werden. In Abschnitt 3.2.5 wurden bereits die Nachteile der Java-eigenen Serialisierungsmechanismen angesprochenen, die bei Deserialisierung eines Objektes zwangsläufig alle von diesem Objekt referenzierten Klassen benötigen und ggf. über das Netzwerk übertragen müssen. Je weniger Datenobjekte ein Agent nun mit sich führt, desto weniger Programmcode muss bei der Deserialisierung gebunden werden.

3. Als Nebeneffekt führt die Möglichkeit externe Datenelemente auf die Ursprungsplattform auszulagern bzw. von dieser nachzuladen, auch zu einer erhöhten Sicherheit gegenüber feindlichen Agentenplattformen, die unter Umständen ein Interesse haben, private Daten des Agenten auszulesen bzw. zu ändern.

Somit lässt sich bei der Migration ein geringeres Übertragungsvolumen erzielen, indem auf der einen Seite nicht weiter benötigte Daten zurück zum Benutzer auf die Ursprungsplattform gesendet werden und auf der anderen Seite Daten erst bei Bedarf wieder von der Ursprungsplattform nachgeladen werden können, sodass der Agent bei jeder Migration möglichst wenig Daten mitzuführen hat.

Codeeinheiten und Codebasen Die durch Kalong eingeführte Repräsentation eines Agenten bezieht sich jedoch nicht nur auf die Daten des Agenten, es wird neben dem externen Zustand auch noch eine neue Repräsentation für den Programmcode eingeführt. Anstatt einzelne Klassen oder komplette Bibliotheken (Java-Archive, Jars) bei der Migration zu übertragen, führt Kalong das Konzept der Codeeinheit (engl. *code unit*) ein. Eine Codeeinheit besteht aus mindestens einer Klasse und bündelt idealerweise Klassen mit derselben Ausführungswahrscheinlichkeit bzw. Klassen, die zur Bearbeitung derselben Teilaufgabe benötigt werden. Eine Agenteninstanz kann dabei vor ihrer ersten Migration selbständig entscheiden, wie sie ihre eigenen Codeeinheiten definiert, sodass es durchaus möglich sein kann, dass zwei Agenten desselben Typs verschiedene Codeeinheiten haben und umgekehrt eine Klasse Teil mehrerer Codeeinheiten gleichzeitig ist.

Zu jeder Codeeinheit gehört mindestens eine Codebasis (engl. *code base*), von der diese Einheit geladen werden kann. Normalerweise ist die Ursprungsplattform einer Agenten immer eine Codebasis für seine individuellen Codeeinheiten. Die Beschreibung von Codeeinheiten umfasst also

immer ein oder mehrere Adressen von Agentenplattformen, die bei Bedarf sukzessive kontaktiert werden können, um Codeeinheiten anzufordern.

Der Vorteil, Programmcode für den Transfer nicht mehr nur in Klassen und Bibliotheken zu gliedern, sondern in selbst definierte Einheiten zu unterteilen, liegt in einer besseren Ausnutzung der Netzwerkressourcen. Auf der einen Seite kann ein Agent vor einer Migration mittels einer push-Strategie einfach und genau entscheiden, welche Teile seines Programmcodes an die entfernte Plattform verschickt werden sollen, auf der anderen Seite müssen bei einer pull-Strategie nicht mehrere Netzwerkverbindungen (bei Java eine Verbindung pro Klasse) aufgebaut werden und nicht alle Klassen einer gesamten Bibliothek (Java-Archiv) übertragen werden.

Zerlegen von Klassen Bisher wurde immer davon gesprochen, Programmcode in Form von Java-Klassen, Java-Archiven oder Codeeinheiten bei der Migration zu übertragen. Codeeinheiten sollten zum Beispiel, wie oben erwähnt, Klassen mit ähnlicher Ausführungswahrscheinlichkeit zusammenfassen. Um dies noch etwas tiefergehend zu verfolgen, untersuchten [Braun u. a. 2005, Kern und Braun 2005] die typische Struktur des Programmcodes von Agenten und fanden, dass die Ausführungswahrscheinlichkeit auf Methodenebene um einiges aussagekräftiger ist. Typischerweise besteht der Programmcode eines Agenten aus folgenden Arten von Methoden:

- Methoden, die nur während der Startphase eines Agenten auf dessen Ursprungsplattform ausgeführt werden.
- Methoden, die zur Bearbeitung von Aufgaben benötigt werden, während der Agent einzelne Plattform des Netzwerkes durchläuft.
- Methoden, die erst gegen Ende des Lebenszyklus eines Agenten, typischerweise auf dessen Ursprungsplattform, ausgeführt werden.
- Methoden, die bei unerwarteten Ausnahmesituationen aufgerufen werden.

Als Konsequenz dieser Untersuchung schlagen Braun u.a. vor, die Methoden der zweiten Art in sinnvoller Weise für die Migration zu gruppieren und die übrigen Methoden auf der Ursprungsplattform des Agenten zu belassen und erst nach Bedarf zu laden. Um den Programmierer nicht vor die Aufgabe zu stellen, diese Aufteilung manuell durchzuführen, wird in [Kern und Braun 2005, Braun u. a. 2005] ein Verfahren vorgestellt, welches die Ausführungswahrscheinlichkeiten von Methoden analysiert und Methoden mit ähnlichen Ausführungswahrscheinlichkeiten als eine Einheit zusammenfasst. Auf diese Art können zum Beispiel Methoden, die sehr wahrscheinlich auf einer Plattform ausgeführt werden, zusammen übertragen werden, während hingegen Methoden, die in gleichem Maße unwahrscheinlich ausgeführt werden, erst bei Bedarf zusammen geladen werden. In einer Vorverarbeitung werden die Ausführungswahrscheinlichkeiten von Methoden durch eine statische Analyse des Bytecodes und durch die dynamische Erstellung von Profilen zur Laufzeit berechnet. Auf den resultierenden Ausführungswahrscheinlichkeiten aufbauend, beginnt dann die Zerlegung der Klassen (engl. *class splitting*) auf Ebene des Bytecodes in kleinere Übertragungseinheiten.

Hierbei werden die ursprünglichen Methoden, ggf. zusammen mit Methoden mit derselben Ausführungswahrscheinlichkeit, in eine neue Klasse ausgelagert und sogenannte Stellvertretermethoden (auch als *Stub-Methoden* bezeichnet) an deren ursprünglicher Stelle neu eingefügt. Der einzige Zweck dieser Stellvertretermethoden ist die spätere Weiterleitung eines Aufrufes durch Instanziierung eines Objektes der neu erzeugten Klasse und Aufruf der Originalmethode.

Letztendlich bekommt man durch dieses Verfahren zwar mehr, aber dafür kleinere Klassen, die unter Umständen auch mehrere Methoden mit gleicher Aufrufwahrscheinlichkeit beinhalten, sodass das Laden von Klassen über das Netzwerk schneller wird und weniger überflüssige Codefragmente übertragen werden müssen.

Codeserver und Spiegelplattformen Im Normalfall unterscheidet man zwei verschiedene Typen von Plattformen: 1) die Ursprungsplattform eines Agenten (auch Heimatplattform genannt) und 2) allgemein entfernte Plattformen. Kalong führt zwei weitere Typen von Plattformen ein, sogenannte Codeserver und Spiegelplattformen (engl. *mirror agencies*).

Eine Codeserver-Plattform dient als Codebasis für das Laden von Codeeinheiten. Im vorigen Abschnitt wurde erwähnt, dass nach Verlassen einer Plattform, alle Daten des Agenten, also auch die ihm zugehörigen Codeeinheiten, gelöscht würden. Als Teil seiner Migrationsstrategie kann der Agent jedoch entscheiden, die Plattform als einen Codeserver für bestimmte Codeeinheiten zu deklarieren. In diesem Fall wird der Name der Plattform für die entsprechenden Codeeinheiten als Codebasis vermerkt und die Codeeinheiten werden nach der Migration des Agenten nicht gelöscht, sodass sie zu einem späteren Zeitpunkt per pull-Strategie nachgeladen werden können. Natürlich kann ein Agent einen Codeserver auch jederzeit wieder aufgeben. In diesem Fall sendet er einen festgelegten Befehl an die Agentenplattform, welche daraufhin die Codeeinheiten löscht.

Analog zu der Deklaration eines Codeservers für Programmcode, erlaubt Kalong auch die Bestimmung sogenannter Spiegelplattformen, auf denen Daten hinterlassen werden können. Wie bereits oben erwähnt, ist es einem Agenten erlaubt, Teile seines externen Zustandes ausschließlich an seine Ursprungsplattform zu übertragen bzw. von dieser anzufordern. Ist die Ursprungsplattform eines Agenten aber zum Beispiel durch eine Netzwerkverbindung mit sehr geringer Datenrate an den übrigen Teil des Netzwerkes angebunden (zum Beispiel ein mobiles Gerät über eine WLAN-Verbindung), kann es unter Umständen sinnvoll sein, eine andere Plattform im Netzwerk auszuwählen, die als temporäre Ursprungsplattform fungiert. Eine solche Plattform nennt man Spiegelplattform.

Codespeicher Das Zwischenspeichern (engl. *caching*) von Programmcode ist eine allgemein verwendete Technik, mit Hilfe derer das Datenaufkommen im Netzwerk reduziert werden kann. Hierbei wird häufig verwendeter Programmcode nach erstmaligem Laden lokal in einem Codespeicher der Plattform hinterlegt. Zukünftig muss dieser Programmcode nicht erneut an die Plattform übertragen werden, sodass das Übertragungsvolumen bei einer Migration reduziert werden kann. Java unterstützt bereits einen solchen Codespeicher als Teil des integrierten Classloaders, jedoch weist dieser zwei wesentliche Defizite auf. Zum einen verhindert der Codespeicher lediglich das wiederholte Nachladen von Programmcode von anderen Plattformen (pull-Strategie), ist aber bei Einsatz von push-Strategien nutzlos. Zum anderen kann er nicht mit verschiedenen Versionen von Klassen umgehen, die denselben Namen tragen (wie z.B. bei JADE [Bellifemine u. a. 2007]). Als Konsequenz hieraus müsste für jede Agenteninstanz ein eigener Classloader und entsprechend ein eigener Codespeicher eingesetzt werden, um Konflikte zu vermeiden (wie z.B. bei Grasshopper [Magedanz u. a. 1999]), was den Nutzen eines Codespeichers auf ein Minimum reduziert [Braun u. a. 2005].

Aus diesen Gründen umfasst Kalong einen eigenen Codespeicher, welcher von allen Agenten einer Plattform gemeinsam genutzt wird. Dieser arbeitet, genau wie Codespeicher von Java, auf Basis von Klassen und nicht von Codeeinheiten. Er wird sowohl für pull- als auch für push-Strategien eingesetzt und bietet außerdem die Möglichkeit, Klassen nicht nur anhand von

Namen zu identifizieren, sondern auch über sogenannte *MD5-Hashes* [Rivest 1992], also eine Art eindeutige, 16 Byte umfassende Kurzfassung einer Klasse, sodass er auch mit verschiedenen Versionen einer Klasse umgehen kann [Braun u. a. 2005, Kern und Braun 2005]. Um die Vorteile des Codespeichers zu nutzen, ist es notwendig, dass sich zwei Plattformen zu Beginn eines Migrationsprozesses über die zu versendenden und gegebenenfalls vorhandenen Klassen austauschen. Dieser Vorgang ist in Kalong als Teil des verwendeten *Simple Agent Transfer Protocol* realisiert [Braun und Rossak 2005].

Migrationsstrategien In Abschnitt 3.1.2 wurden bereits gängige Migrationsstrategien beschrieben. Grob lässt sich hierbei zwischen *push-* und *pull-*Strategien unterscheiden. Der Vorteil der pull-Strategien ist, dass nur wirklich benötigte Klassen (bzw. Klassenbibliotheken) nachgeladen werden, dafür jedoch eine offene Netzwerkverbindung bestehen muss - was insbesondere im Fall mobiler Geräte häufig nicht der Fall ist - und für jede Anfrage eine neue Verbindung erstellt wird. Bei push-Strategien wird schon im Vorwege der Programmcode eines Agenten sowie ggf. sein serialisierter Zustand an den Zielort übertragen. Hierdurch bleibt die Autonomie des Agenten gewahrt, da keine Netzwerkverbindung zum eventuellen Nachladen von Klassen mehr benötigt wird. Untersuchungen haben gezeigt, dass weder eine reine push- noch eine reine pull-Strategie in allen Fällen zu einer optimalen Migrationseffizienz führt, wobei die Strategie, mit der höchsten Migrationseffizienz (vgl. Abschnitt 3.2.5) als optimal angesehen wird [Braun u. a. 2001, Braun u. a. 2005].

Wie bereits erwähnt, abstrahiert das Kalong-Mobilitätsmodell über Codeeinheiten von einzelnen Klassen und Java-Archiven. Darüber hinaus erlaubt es jedem Agenten, eine für sich individuell angepasste Migrationsstrategie für jeden einzelnen Migrationsvorgang zu erstellen. Hierzu gehört, dass ein Agent selbst entscheiden kann, welche Teile seines externen Zustandes er gegebenenfalls zu seiner Ursprungsplattform zurücksenden möchte (siehe oben) und welche Codeeinheiten zusammen mit seinem Objektzustand und dem übrigen Teil seines externen Zustandes an die Zielplattform übertragen werden sollen. Darüber hinaus kann eine Migrationsstrategie, wie im nachfolgenden Beispiel beschrieben, auch die Einrichtung von Codeservern oder Spiegelplattformen festlegen [Braun und Rossak 2005, Braun u. a. 2005].

Beispiel und Evaluation

In [Kern und Braun 2005] ist ein Planungsalgorithmus vorgestellt, der für einen Agenten eine optimale Migrationsstrategie unter Zuhilfenahme der von Kalong bereitgestellten Mechanismen erstellen kann. Dieser Algorithmus arbeitet auf Basis vollständigen Wissens über die Umwelt, das bedeutet, dass eine Reihe tatsächlicher Umweltparameter (z.B. Latenzzeit und Datendurchsatz der Netzwerkverbindungen, Ausführungswahrscheinlichkeiten einzelner Programmteile, die Reiseroute des Agenten etc.) bereits im Vorwege bekannt sind. Für die Evaluation wird in [Kern und Braun 2005, Braun und Kern 2005] ein Beispielszenario (siehe Abbildung 3.11) beschrieben, anhand dessen das Vorgehen des Algorithmus verdeutlicht werden soll. Dieses Szenario besteht aus einem Agenten, der auf einem mobilen Gerät gestartet wird und fünf weitere, im Vorwege festgelegte Plattformen besuchen soll, um dort verschiedene Aufgaben auszuführen. Für die Bearbeitung der Aufgaben verwendet der Agent auf allen Plattformen genau eine von fünf unterschiedlichen Klassen. Das mobile Gerät ist über eine langsame und instabile Verbindung mit dem Netzwerk verbunden, während die anderen Plattformen des Netzwerkes über eine schnelle Verbindung untereinander kommunizieren können.

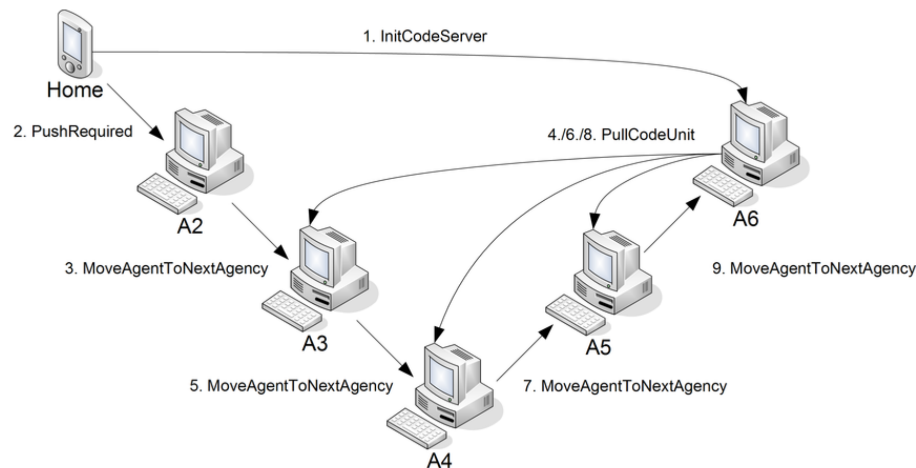

Abbildung 3.11: Kalong: Beispiel für eine Migrationsstrategie (nach [Kern und Braun 2005])

Die Migrationsstrategie, als Ergebnis des Planungsalgorithmus, sieht zuerst die Einrichtung eines Codeservers auf einer der schnell angebundenen Plattformen (in der Abbildung 3.11 dargestellt für Plattform *A6*) vor. Anschließend wird der Agent mit seinen Daten und nur dem notwendigen Programmcode auf Plattform *A2* mittels einer push-Strategie übertragen. Für diese erste Migration ist es schneller eine push-Strategie einzusetzen anstatt den Programmcode von dem entfernten Codeserver zu laden. Im Folgenden migriert der Agent weiter auf die Plattformen *A3* bis *A6*, wobei nur sein Zustand transferiert wird und der entsprechende Programmcode mittels einer pull-Strategie vom Codeserver auf Plattform *A6* geladen wird.

Verglichen mit einer einfachen push-Strategie, die den Zustand des Agenten zusammen mit allen Klassen von Plattform zu Plattform überträgt, ist der oben beschriebene Ansatz im Testaufbau 44% schneller (13,416 Sekunden anstatt 24,00 Sekunden). Verglichen mit einer einfachen pull-Strategie (1139 Sekunden), bei der alle Plattformen den benötigten Programmcode über die langsame Netzwerkverbindung von dem mobilen Gerät laden müssen, ist der Ansatz sogar 98% schneller [Kern und Braun 2005].

Zusammenfassung

In diesem Abschnitt wurde das Kalong-Mobilitätsmodell beschrieben. Kalong erlaubt Agenten die schwache Migration mittels adaptiver Migrationsstrategien, welche die Effizienz eines Migrationsvorgangs verbessern sollen. Mit Hilfe dieser Migrationsstrategien kann zur Laufzeit festgelegt werden, welche Teile des Programmcodes und Datenraumes eines Agenten in die Ziel-Ausführungsumgebung transferiert werden sollen. Zu einer Migrationsstrategie gehören:

- die Möglichkeit, Daten eines Agenten in einen externen Zustand auszulagern, der auf der Ursprungsplattform des Agenten vorgehalten wird,
- die Aufteilung des Programmcodes in Codeeinheiten und die weitergehende Zerlegung von Klassen auf Ebene von Methoden mit gleicher Ausführungswahrscheinlichkeit,
- die Konzepte eines Codeservers und einer Spiegelplattform, um an beliebigen Stellen des Netzwerkes Programmcode und Daten hinterlegen zu können,
- die Verwendung von Codespeichern zum lokalen Zwischenspeichern von empfangenem Programmcode und

- die Möglichkeit, Migrationsstrategien zur Laufzeit zu definieren, um zu bestimmen, mit welchen Mechanismen Programmcode und Daten von einer Plattform auf eine andere Plattform übertragen werden sollen.

Anhand eines Beispielszenarios wurde eine Migrationsstrategie exemplarisch vorgestellt. Für dieses Szenario zeigte sich eine deutliche Verbesserung der Migrationseffizienz, was in praktischen Versuchen nachgewiesen wurde.

3.3.3 Mobilitätssprache

Mit dem Kalong-Modell wurde im vorigen Abschnitt bereits ein Mobilitätsmodell vorgestellt. Für die formale Beschreibung eines Mobilitätsmodells und damit der einzelnen Sichten auf die Migrationsphasen, haben [Braun und Rossak 2005] mit der *Mobility Language* (MoL) eine eigene Beschreibungssprache entworfen. Mit dieser Sprache lassen sich alle Techniken beschreiben, die eine Agentenplattform für die Migration implementiert. Da Kalong keine eigenständige Agentenplattform ist, sondern vielmehr eine unabhängige Softwarekomponente, die von Agentenplattformen verwendet werden kann, wird Kalong in diesem Abschnitt nicht weiter behandelt. Stattdessen wird die Verwendung der im Folgenden vorgestellten Grammatik anhand eines anderen Beispiels, der *Aglets*-Agentenplattform, verdeutlicht. Erst in Abschnitt 4.6 wird die Beschreibungssprache dann wieder Verwendung finden, wenn es darum geht, im Kontext der ereignisbasierten Migration ein eigenes, erweitertes Mobilitätsmodell zu entwickeln.

Die Grammatik der Mobilitätssprache ist in der *Erweiterten Backus-Naur Form* (EBNF) [ISO/IEC 2001] gegeben. Im Folgenden sind die zur Beschreibung von Migrationstechniken wichtigsten Ableitungsregeln (*Produktionen*) aufgeführt, eine detaillierte Beschreibung der vollständigen Grammatik findet sich in [Braun 2003, Braun und Rossak 2005].

Benutzersicht

Wie bereits in der Einführung erwähnt, setzt sich die Beschreibung eines Mobilitätsmodells aus der Benutzer-, der Agenten- und der Netzwerksicht zusammen. Die Sicht des Benutzers beschreibt alle Aspekte der Migration, die für den Programmierer eines Agenten wichtig sind. Hierzu gehören die Adressierung der Zielplattform, der Ort, an dem ein Agent erzeugt wird und von welchem der Programmcode geladen werden kann, was für Typen von Daten unterstützt werden und schließlich welche Arten der Mobilität verwendet werden können.

```
1. <MobilityModel>      ::= <User> + <Agent> + <Network>
2. <User>               ::= <Naming> + <Creating> + <Code> +
                            <Data> + <Migration>
3. <Naming>             ::= <AgencyName> + <AgencyAddress>
[...]
9. <Creating>           ::= 'CreateAt' + '=' + 'CurrentAgency' +
                            (';' + 'RemoteAgency') + '.'
10. <Code>              ::= 'Code' + '=' + <CodeSources> + '.'
[...]
13. <Data>              ::= 'DataTypes' + '=' + <DataTypes> + '.'
14. <DataTypes>         ::= <DataType> + ({ ';' + <DataType>})
15. <DataType>          ::= [ 'Proxy' | 'Static' | 'Moving' | 'Copying' ] +
                            '+' + ['Class' | 'Jar']
```

Die `Naming`-Produktion spezifiziert die Form der Adresse der Zielplattform. Mit der `Creating`-Produktion wird dann die Plattform angegeben, auf der der Agent erstellt und initialisiert werden soll. Dies kann entweder die lokale Agentenplattform oder aber eine entfernte Plattform sein. Dies ist zum Beispiel nützlich, falls die lokale Plattform nicht über ausreichend Ressourcen verfügt, um einen Agenten zu erstellen. Über die `Code`-Produktion wird bestimmt, von welchen Quellen der Programmcode eines Agenten geladen werden kann. Eine solche Quelle ist zum Beispiel der Klassenpfad der lokalen Plattform, das Dateisystem oder aber eine entfernte Quelle, die zum Beispiel über das HTTP- oder FTP-Protokoll das Laden von Programmcode erlaubt. Die `Data`- und `DataType`-Regeln legen fest, welche Arten von Daten unterstützt werden, unterteilt in Klassen und Klassenbibliotheken (*Java Archive*, Jar). Jede Art von Daten verhält sich bei der Migration anders (vgl. Abschnitt 3.1.2). Braun unterscheidet hier vier verschiedene Arten:

Static Diese Art von Daten ist nicht mobil und kann nicht von entfernten Plattformen aus zugegriffen werden.

Moving Diese Daten sind mobil und werden bei der Migration zusammen mit dem Agenten verschoben, sind also nach der Migration auf der Ursprungsplattform nicht mehr vorhanden.

Copying Diese Daten sind ebenfalls mobil und bleiben auch nach der Migration auf der Ursprungsplattform bestehen. Eine Kopie der Daten wird dem serialisierten Agenten beigefügt. Änderungen an den ursprünglichen Daten oder der Kopie sind nicht transparent.

Proxy Proxy-Daten sind nicht mobil, entfernter Zugriff ist aber möglich. Daten dieser Art existieren nur auf der Ursprungsplattform, auf sie kann aber transparent über sogenannte *Proxy*-Objekte zugegriffen und sie können verändert werden.

Die Entwurfsmöglichkeiten bezüglich der Migration werden in der Benutzersicht durch folgende Produktionen beschrieben, die den Initiator einer Migration benennen, die Art der Mobilität sowie den Effekt einer Migration bestimmen und schließlich das Verhalten im Fehlerfall festlegen.

```
16. <Migration>        ::= <Initiator> + <Mobility> +
                               <DestinationAddress> + <Effect> + <Error>
17. <Initiator>        ::= 'MigrationInitiator' + '=' + 'Agent' +
                               <OtherInitiator> + '.'
18. <OtherInitiator>   ::= (';' + 'OtherAgent' + ('withVeto')) +
                               (';' + 'Agency' + ('withVeto')) +
                               (';' + 'Owner' + ('withVeto'))
19. <Mobility>         ::= 'Mobility' + '=' + [ 'Weak' + '.' + <Weak> |
                               'Strong' + '.' + <Strong>]
20. <Weak>             ::= 'WeakMobility' + '=' +
                               ['FixedMethod' | 'ArbitraryMethod'] + '+' +
                               ['Command' | 'Ticket'] + '.'
21. <Strong>           ::= 'StrongMobility' + '=' +
                               ['SourceCodeTransformation' | 'JVMModification'] + '.'
26. <MigrationEffect>  ::= 'MigrationEffect' + '=' + <Effects> + '.'
27. <Effects>          ::= <Effect> + ({ ';' + <Effect>})
28. <Effect>           ::= ['Move' | 'Copy' | 'Cloning']
29. <Error>            ::= 'MigrationError' + '=' +
                               ['Restart' | 'ErrorMethod' | 'Exception'] + '.'
```

Der Initiator einer Migration kann entweder der Agent selbst oder eine andere Instanz, zum Beispiel ein anderer Agent, die Agentenplattform oder der Eigentümer (Benutzer) des Agenten, sein und wird über die `Initiator`-Produktion angegeben. Im Grunde widerspricht die Initiierung einer Migration seitens einer externen Quelle der Autonomie des Agenten, kann aber unter bestimmten Umständen sinnvoll sein, zum Beispiel beim Beenden der Plattform. Im Falle externer Initiierung kann es nützlich sein, dem zu migrierenden Agenten ein Veto-Recht einzuräumen, über das er als letzte Instanz gegen eine Migration stimmen kann.

Die `Mobility`-Regel beschreibt die Art der Mobilität, also ob es sich um schwache oder starke Mobilität handelt (vgl. Abschnitt 3.1.2). Da bei der schwachen Migration der momentane Ausführungszustand des Agenten unberücksichtigt bleibt, kann man angeben, ob der Agent auf der Zielplattform mit der festgelegten Startmethode aufgerufen werden soll (Terminal `FixedMethod`) oder ob man dem Agenten einen beliebigen Methodennamen mitgeben kann, der die Methode bezeichnet, die von der Zielplattform nach der Migration mittels Reflection aufgerufen wird (Terminal `ArbitraryMethod`). Die Unterscheidung zwischen Migration über ein Kommando oder über ein Ticket bezieht sich auf die Dynamik hinsichtlich der Entscheidung über das nächste Migrationsziel. Bei der Migration mittels eines Kommandos, behält der Programmierer die Kontrolle, wann (an welcher Programmstelle) und wohin ein Agent migrieren soll. Ein Ticket hingegen, welches das nächste Migrationsziel eines Agenten enthält, kann zur Laufzeit geändert werden. In diesem Fall ist es für den Programmierer nicht mehr offensichtlich, was der Agent nach der Migration ausführen wird.

In Abschnitt 3.1.5 wurde erläutert, welche Möglichkeiten es in der Programmiersprache Java gibt, um eine starke Migration zu realisieren. Hierzu gehörten die Änderungen an dem Programmcode des Agenten (sowohl Quell- als auch Bytecode) und Änderungen an der Java Virtual Machine (JVM). Genau dies spiegelt die `Strong`-Produktion wieder. Die `Effect`-Regel bestimmt, ob ein Agent vor der Migration entweder geklont, kopiert oder bei der Migration verschoben wird. Ein Klon eines Agenten erhält dieselben Daten (und ggf. denselben Ausführungszustand) wie das Original und wird dann an die entfernte Plattform verschickt. Eine Kopie hingegen erhält keine Kopie der Daten, sondern wird ganz neu instanziiert und so an die entfernte Plattform übermittelt.

Die letzte Regel der Benutzersicht bezieht sich auf das Verhalten eines Agenten im Fehlerfall und hängt von der Art der Mobilität (stark oder schwach) ab. Möglichkeiten sind das Neustarten eines Agenten (z.B. falls der gewünschte Methodenname auf der Zielplattform nicht gefunden werden kann, vgl. Terminal `ArbitraryMethod`), das Aufrufen einer speziellen Methode zur Fehlerbehandlung oder das Anzeigen eines Ausnahmefalls über eine sogenannte *Exception* (dt. Ausnahme).

Agentensicht

Die Agentensicht beschäftigt sich mit der Art und Weise, wie der Programmcode eines Agenten im Netzwerk übertragen werden kann. Die Übertragung der Daten, unterteilt in verschiedene Arten von Daten, wurde bereits im vorigen Abschnitt behandelt. Das Versenden von Programmcode aus Agentensicht wird bestimmt durch die Codestrategien, ggf. dem Einsatz eines Programmcode-Zwischenspeichers (engl. *code cache*) und der Angabe von Programmpaketen, die nicht versandt werden sollen, da sie auf allen Plattformen vorhanden sein sollten (z.B. native Java-Pakete oder die Bibliotheken der Agentenplattform selbst).

```
30. <Agent>             ::= <CodeTransfer> + (<CodeCache>) +
                            (<UbiquitousClasses>)
31. <CodeTransfer>      ::= 'CodeRelocation' + '=' + <CodeStrategies> + '.'
32. <CodeStrategies>    ::= <CodeStrategy> + ({ ';' + <CodeStrategy>})
33. <CodeStrategy>      ::= [<Push> | <Pull>]
34. <Push>              ::= 'Push' + <ClassClosure> + 'To' + <PushTarget>
35. <ClassClosure>      ::= ['SerializedAgentClosure' | 'JarClosure' |
                            'AgentClassClosure' | 'UserDefinedClosure']
36. <PushTarget>        ::= ['NextServer' | 'ManyServers']
37. <Pull>              ::= 'Pull' + [ 'Jar' | 'Class' ] +
                            'From' + <PullTargets>
38. <PullTargets>       ::= <PullTarget> + ({ '+' + <PullTarget>})
39. <PullTarget>        ::= [ <AgencyType> | 'ClassLoader' |
                            'ClassPath' | 'Cache' | 'CodeSource']
40. <AgencyType>        ::= ['Home' | 'Remote' | 'LastServer']
41. <UbiquitousClasses> ::= 'UbiquitousClasses' + '=' + ['SystemDefined' |
                            'UserDefined' | 'AgentDefined'] + '.'
[...]
```

Die `CodeStrategy`-Produktion erlaubt den Einsatz von push- und/oder pull-Strategien (vgl. Abschnitt 3.1.2). Die Produktion für push-Strategien erlaubt das Versenden verschiedener Codepakete entweder an den nächsten Server (Terminal `NextServer`) oder an eine Reihe von Server (Terminal `ManyServers`). Die Codepakete werden durch sogenannte *Closures* (dt. Abschluss) definiert. Die `AgentClosure` umfasst alle Klassen, die der Agent während seiner Lebenszeit benutzen könnte, die `SerializedAgentClosure` hingegen nur die Klassen der Objekte, die in der serialisierten Form des Agenten enthalten sind. Die `JarClosure` transferiert alle Klassen eines Java-Archives, sobald eine der Klassen angefordert wird, und die `UserDefinedClosure` überlässt es schließlich dem Benutzer anzugeben, welche Klassen zur Laufzeit transferiert werden sollen.

Die `Pull`-Produktion gibt an, wie Programmcode bei Bedarf von einer Plattform angefordert werden kann. Hier unterscheidet man zum einen, ob einzelne Klassen oder ganze Archive angefordert werden sollen und zum anderen, von wo der Programmcode zu laden ist. Hierbei lassen sich über die `PullTarget`-Regel ein oder mehrere Ziele angeben, die der Reihe nach angefragt werden. So ist es möglich, eine bestimmte Agentenplattform als Ziel anzugeben, die Verantwortung an einen speziellen Classloader abzugeben, über den Standard-Classloader Klassen aus dem lokalen Dateisystem bzw. dem sogenannten Klassenpfad (engl. *classpath*) zu laden oder die geforderte Klasse aus einem besonderen Zwischenspeicher für Klassen (engl. *class cache*) zu holen. Schließlich erlaubt die `Ubiquitous`-Regel die Angabe einer Menge derjenigen Klassen, die auf allen Plattformen bereits vorhanden sein sollten. Hierzu gehören im Allgemeinen die nativen Java-Klassen sowie die Klassen der Agentenplattform selbst.

Netzwerksicht

Die letzte der Sichten beschäftigt sich mit den Aspekten der Datenübertragung auf der Netzwerkschicht. Eine Übertragungsstrategie (engl. *transmission strategy*) definiert hierbei, wie ein Agent tatsächlich von der Ursprungs- bis zur Zielplattform übertragen wird.

```
44. <Network>               ::= <MigrationProtocol> +
                                <TransmissionProtocols>
45. <MigrationProtocol>     ::= 'MigrationProtocol' + '=' +
                                ['Synchronous' | 'Asynchronous'] +
                                ('FailureAtomic') + '.'
46. <TransmissionProtocols> ::= 'TransmissionProtocol' + '='
                                <NetworkProtocols> + '.'
47. <NetworkProtocols>      ::= <NetworkProtocol> +
                                ({ ';' + <NetworkProtocol>})
```

Zu einer Übertragungsstrategie gehören ein `MigrationProtocol` und ein oder mehrere `TransmissionProtocols`. Ein Migrationsprotokoll kann entweder synchron oder asynchron ablaufen. Ein asynchrones Protokoll bietet den Vorteil höher Leistung, ist jedoch weniger verlässlich, da die entfernte Plattform den Empfang eines Agenten nicht quittiert. Über das Terminal `FailureAtomic` kann angegeben werden, ob ein Migrationsprotokoll garantiert, dass ein Agent entweder ganz oder im Falle von Fehlern gar nicht übertragen wird. Die `TransmissionProtocols`-Regel erlaubt schließlich die Angabe ein oder mehrerer Netzwerkprotokolle, zum Beispiel *TCP/IP*, *HTTP*, *SSL*, *CORBA*, *SOAP* etc., die für den Versand eingesetzt werden können.

Beispiel

Um die Anwendung der Mobilitätssprache zu demonstrieren, soll im Folgenden das Mobilitätsmodell einer existierenden Agentenplattform, der *Aglets*-Plattform [13] [Lange und Oshima 1998], beschrieben werden (für weitere Details sei auf [Braun 2003, Braun und Rossak 2005] verwiesen). Die Beschreibung besteht aus mehreren Zeilen, die jeweils ein Name-Wert Paar enthalten. Der Name bezieht sich auf einen bestimmten Entwurfsaspekt und der Wert entweder auf eine einzelne Entwurfsentscheidung oder auf eine Menge - durch Semikolon separierter - Entwurfsalternativen.

```
// Benutzersicht
[...]
CreateAt            = CurrentAgency.
Code                = ClassPath+Class; ClassPath+Jar; FileSystem+Class;
                      FileSystem+Jar; HTTP+Class; HTTP+Jar;
                      MigrationProtocol+Class; MigrationProtocol+Jar.
DataTypes           = Static; Copy; Move.
MigrationInitiator  = Agent; OtherAgent; Owner.
Mobility            = Weak.
WeakMobility        = FixedMethod+Command.
```

Aus Sicht des Benutzers, kann ein Agent in Aglets lediglich auf der aktuellen Plattform erstellt werden, es ist also nicht möglich, ein oder mehrere entfernte Plattformen als Ziel für die Instanziierung anzugeben. Auch kann lediglich eine einzige Codequelle, also ein Ort, von wo der Programmcode des Agenten geladen werden soll, angegeben werden. Diese Codequelle ist jedoch flexibel wählbar und kann entweder über den Standard-Classloader, lokal im Dateisystem oder

[13] Der Begriff Aglet leitet sich von den in Java bekannten *Applets* ab und bezeichnet zusammengefasst ein Java-Objekt, welches die Fähigkeit der Mobilität, Persistenz und Kommunikation mit anderen Aglets besitzt.

entfernt über das HTTP-Protokoll angesprochen werden. Mit Ausnahme des Datentyps Proxy, unterstützt Aglets durch den normalen Java-eigenen Serialisierungsmechanismus sowohl nicht-mobile statische Datentypen als auch das Kopieren oder Verschieben mobiler Datentypen.

Der Initiator eines Migration muss nicht zwangsläufig der Agent sein. Eine Migration kann auch von externer Seite angestoßen werden, also durch einen anderen Agenten oder den Benutzer, nicht jedoch von der Agentenplattform selbst. Aglets unterstützt lediglich eine schwache Form der Mobilität, bei der Programmcode und Daten transferiert werden, ohne dass der Ausführungszustand berücksichtigt wird. Auf einer entfernten Plattform angekommen, wird der deserialisierte Agent durch Aufruf einer fest vorgegebenen Methode gestartet.

```
// Agentensicht
[...]
MigrationEffect   = Move.
MigrationError    = Restart.
CodeRelocation    = PushSerializedAgentClosureToNextServer;
                    PushJarClosureToNextServerwithbaseJar;
                    PullClassFromClassLoader+ClassPath+Cache+CodeSource.
```

Als Effekt einer Migration wird ein migrierender Agent auf jeden Fall verschoben, Kopieren oder Klonen sind in Aglets nicht vorgesehen. Tritt während der Migration ein Fehler auf, so wird der migrierte Agent an entfernter Stelle einfach neu gestartet, falls möglich. Die Migrationsstrategie besteht nicht nur aus einer simplen push- oder pull-Strategie, sondern aus einer Kombination dieser beiden. Bei der Migration werden alle Klassen, die der Agent gerade benutzt bzw. deren Objekte in der serialisierten Form des Agenten enthalten sind, mittels einer push-Strategie auf die entfernte Plattform übertragen. Gehören ein oder mehrere Klassen zu Java-Archiven (Jars), so wird das komplette Archiv übertragen. Wird bei der Ausführung an entfernter Stelle schließlich eine Klasse benötigt, die nicht per push-Strategie übertragen wurde, so wird diese zuerst an verschiedenen Stellen lokal gesucht (Classpath, Cache etc.) und bei Fehlschlagen der Suche schließlich von der Ursprungsplattform per pull-Strategie nachgeladen.

```
// Netzwerksicht
MigrationProtocol    = Synchronous.
TransmissionProtocol = TCP/IP; HTTP.
```

Das für die Migration verwendete Protokoll arbeitet synchron, Sender und Empfänger stehen also während des Migrationsvorgangs in ständiger Verbindung und können so den Erfolg oder Misserfolg des Vorgangs feststellen. Als Übertragungsprotokolle werden von diesem Migrationsprotokoll sowohl TCP/IP als auch das HTTP-Protokoll unterstützt.

3.3.4 Zusammenfassung

Mit Hilfe von Mobilitätsmodellen lassen sich die Arten der Mobilität sowie die hierfür verwendeten Mechanismen, die von einer Agentenplattform implementiert werden, beschreiben. Ein solches Modell kann einerseits zur Beschreibung von Anforderungen an ein neu zu entwickelndes System, andererseits aber auch für den Vergleich der Migrationstechniken verschiedener existierender Systeme verwendet werden. Mit dem Kalong-Mobilitätsmodell wurde ein Modell vorgestellt, welches durch vielerlei Mechanismen versucht, insbesondere die Migrationseffizienz mobiler Agenten zu steigern. Die einzelnen Mechanismen setzen dabei die Lösungsansätze zur Optimierung der Migrationseffizienz um, die in Abschnitt 3.2.5 vorgestellt wurden. Schließlich wurde

mit der Mobility Language eine formale Beschreibungssprache für Mobilitätsmodelle eingeführt. Diese Sprache beschreibt drei Sichten auf die einzelnen Phasen der Migration, wobei jede Sicht eine bestimmte Abstraktionsebene einnimmt. Mit Hilfe der EBNF-Grammatik dieser Sprache wird im folgenden Kapitel ein neues Mobilitätsmodell entworfen, welches das Kalong-Modell umfasst und weitergehend ergänzt.

3.4 Zusammenfassung

Die Migration von Programmcode und Daten stellt ein bedeutendes Entwurfs- und Konstruktionsprinzip verteilter Anwendungen dar. In diesem Kapitel wurden diesbezüglich eine Reihe von Abstraktionen, Mechanismen, Technologien und Entwurfsparadigmen vorgestellt. Im Kontext dieser Arbeit gehört hierzu vor allem das Paradigma mobiler Agenten.

Durch die Eigenschaften der Adaptivität, Proaktivität und Autonomie, eignen sich mobile Agenten in besonderem Maße für die Ausführung von Aufgaben in verteilten Systemen. Ihr besonderer Vorteil gegenüber anderen Paradigmen liegt in der potenziellen Reduzierung des Übertragungsvolumens und damit auch der Antwortzeiten einer Anwendung. Dieser Vorteil ist umso bedeutender, wenn man die speziellen Anforderungen von mobilen ad-hoc Netzwerken betrachtet, denn für diese hochgradig dynamischen Netzwerke eignen sich die herkömmlichen Paradigmen (z.B. das Client/Server-Paradigma) nur bedingt.

Doch konnten mobile Agenten die an sie gestellten Erwartungen in der Praxis nicht gänzlich erfüllen, was unter anderem auf das schlechte Abschneiden existierender Agentenplattformen hinsichtlich der Migrationseffizienz mobiler Agenten zurückzuführen ist. Speziell dieser Kritikpunkt wurde in einer Reihe von Untersuchungen beleuchtet, woraus Lösungsvorschläge zur Optimierung des Migrationsprozesses abgeleitet wurden. Diese Lösungsvorschläge fanden Einzug in den Entwurf des Kalong-Mobilitätsmodells, welches Agenten eine sehr effiziente Migration erlaubt. Hierbei können die Agenten adaptiv zur Laufzeit eigene Migrationsstrategien erstellen und so entscheiden, auf welche Art und Weise ihr Programmcode und ihre Daten von einem Ausführungsort zum anderen transferiert werden sollen.

4 Konzeption der kontextabhängigen und eigenverantwortlichen Migration

Das in diesem Kapitel zu entwickelnde Konzept einer kontextabhängigen, autonomen und eigenverantwortlichen Migration mobiler Agenten in heterogenen Umgebungen umfasst im Wesentlichen zwei Aspekte. Als erstes sollen allgemein die Handlungs- und Entscheidungsmöglichkeiten eines mobilen Agenten durch kontinuierliche Wahrnehmung seiner dynamischen Umwelt erweitert werden. Die Wahrnehmung umfasst hierbei sowohl die in der Umwelt vorhandenen Ressourcen als auch die Vielfalt der Vorgänge in der Umwelt. Diese Umweltinformationen bilden die Basis für eine höhere Autonomie des Agenten gegenüber seinem Benutzer, da Informationen über vorhandene Ressourcen fortan zum Umweltmodell eines Agenten gehören und nicht von dem Benutzer (oder dem Programmierer) vorgegeben werden müssen. Sie stellen aber auch die Grundlage für die kontextabhängige Migration dar, denn ein Agent bekommt die Möglichkeit eine Migrationsentscheidung abhängig von den Gegebenheiten der Umwelt zu fällen.

Der zweite Aspekt des Konzeptes betrifft die Migration selbst und adressiert zum einen alle Phasen eines Migrationsvorgangs, zum anderen aber auch etwaige Maßnahmen zur Absicherung dieses Vorgangs, um das Risiko des Verlustes eines Agenten bzw. des Verlustes der Kommunikationsverbindung zu dessen Benutzer zu minimieren. Diese Absicherung ist Teil der eigenverantwortlichen Migration und entsprechende Maßnahmen sollen von einem Agent autonom und in Abhängigkeit von seiner gegenwärtigen Umwelt getroffen werden können. Auf diese Weise kann ein Agent sich selbständig in offenen Netzen bewegen, um den zur Bearbeitung seiner Aufgabe angemessenen Ausführungsort aufzusuchen, ohne dass der Benutzer das Risiko des Verlustes seines Agenten eingeht.

Als eine Maßnahme zur Absicherung des Migrationsvorgangs soll ein Agent Kopien seiner selbst in verschiedenen Ausführungsumgebungen hinterlassen können. Diese Kopien dienen einerseits als Sicherheitskopien für den Fall des Abbruchs von Kommunikationsverbindungen, andererseits sollen diese Kopien aber auch aktiv Teilaufgaben bearbeiten können, um die Gesamtdauer einer Bearbeitung zu reduzieren, was insbesondere in oft kurzlebigen ad-hoc Netzwerken von Vorteil sein kann. Durch die resultierende Erhöhung der Autonomie und Verantwortung eines Agenten einerseits, und der Möglichkeit Aufgaben in dynamischen Umgebungen schnell und verlässlich zu bearbeiten anderseits, soll letztlich der Benutzer profitieren und entlastet werden, da dieser nur noch Kenntnis darüber besitzen muss, *was* ein Agent zu erledigen hat und nicht mehr *wie* und vor allem *wo* der Agent die Aufgabe bearbeitet.

Die in dieser Arbeit adressierten Umgebungen (heterogene Infrastruktur- und ad-hoc Netzwerke) bzw. deren besondere Charakteristiken stellen jedoch eine Reihe von Anforderungen an die Wahrnehmung der Umwelt sowie an die Migration. Hierzu gehören die Berücksichtigung der Heterogenität und damit einhergehend die Unterstützung möglichst vieler Discovery-Verfahren, Arten der Ressourcenrepräsentation und Migrationsmechanismen. Hinsichtlich der möglichen Dynamik der Umwelt, sollten diese darüber hinaus zur Laufzeit an die Umweltbedingungen adaptiert werden können, sodass die Wahrnehmung der Umwelt und die Migration von Agenten verlässlich und effizient realisiert werden können. Die einzelnen Gesichtspunkte werden im Folgenden näher erläutert.

Kontextabhängigkeit Der Begriff des Kontextes soll sich in diesem Fall auf die Umwelt des Agenten beziehen, welche ausschließlich die für einen Agenten interessanten Entitäten respektive Ressourcen enthält. Die Möglichkeit abhängig von dieser Umwelt Aktionen auszuführen, bedeutet, dass ein Agent ein Modell seiner Umgebung benötigt und über etwaige Ereignisse in seiner Umgebung informiert werden muss. Sobald die Umwelt einen bestimmten, durch den Agenten vorgegebenen Zustand eingenommen hat oder ein festgelegtes Ereignis aufgetreten ist, wird der Agent informiert und kann sich zur Ausführung einer Aktion entschließen.

Migration und Verteilung Unter einer Aktion als Antwort auf ein Umweltereignis lassen sich prinzipiell beliebige Operationen verstehen. Der Fokus dieser Arbeit liegt jedoch insbesondere auf den anwendungsunabhängigen Aktionen des Migrierens und Verteilens bzw. Klonens. Die Migration wird hierbei als das Verschieben eines Agenten von einer Ausführungsumgebung in eine andere angesehen, während das Klonen das Erzeugen einer Kopie des Agenten in einer entfernten Ausführungsumgebung bezeichnet, was im Kontext dieser Arbeit auch als Distribution (von lat. *distribuere*, dt. zuteilen, verteilen) eines Agenten verstanden wird.

Heterogene Umgebungen Die Ausführung eines Agenten geschieht nicht zwangsläufig in einer statischen und homogenen Umgebung, sondern vielmehr in dynamischen Umgebungen, in denen der Agent auf wechselnde heterogene Systeme trifft. Die Heterogenität zielt hierbei nicht nur auf die Hardware der einzelnen Geräte, sondern ebenfalls auf die Softwareinfrastruktur. Mit der Migration bzw. Distribution eines Agenten in heterogenen Umgebungen sei jedoch weniger die Ausführung in verschiedenartigen Ausführungsumgebungen (z.B. auf unterschiedlichen Agentenplattformen) gemeint, sondern vielmehr die Adaption des Agenten an unterschiedliche Verfahren und Modelle, die in den verschiedenen Umgebungen eingesetzt werden. Dies betrifft insbesondere Verfahren zum Austausch von Ressourceninformationen, Migrationsverfahren sowie Repräsentationsmodelle für Ressourcen. Speziell die Kenntnis der unterschiedlichen Charakteristiken von ad-hoc und Infrastrukturnetzwerken und die damit einhergehende Vielfalt an Verfahren und Modellen sowie die damit verbundenen speziellen Anforderungen sollen von einem Agenten bei der Migration berücksichtigt werden.

Selbstorganisation Die Forderung, dass die Distribution von dem Agenten selbst organisiert werden soll, geht einher mit der konstituierenden Eigenschaft der Autonomie von Agenten. Der Benutzer soll seinen Agenten lediglich mit einer Aufgabe betrauen müssen, wie und vor allem wo diese Aufgabe bearbeitet wird, soll transparent sein. Der Agent soll selbst entscheiden können, abhängig von dessen Kontext, ob er die Aufgabe lokal oder entfernt bearbeiten möchte und ob es gegebenenfalls effektiver wäre, weitere Instanzen seiner selbst mit Teilaufgaben zu betrauen und im Netzwerk zu verteilen. Letztgenannter Punkt unterscheidet die Selbstorganisation auch von der Autonomie, da neben einer selbständigen Entscheidung auch unter Umständen komplexere Verwaltungsaufgaben hinsichtlich der Verteilung und späteren Zusammenführung von einem Agenten übernommen werden müssen.

Eigenverantwortlichkeit Wie oben bereits erwähnt soll die Entscheidung eines Agenten zu migrieren oder sich zu verteilen autonom getroffen werden. Der Agent bestimmt also selbständig über die Ausführung einer bestimmten Aktion. Eine Aktion aus eigenem Antrieb auszuführen umfasst jedoch nicht zwangsläufig, sondern höchstens implizit, dass der Agent auch die damit verbundene Verantwortung übernehmen muss. Die Forderung nach Eigenverantwortlich-

keit soll daher explizit ausdrücken, dass der Agent die Konsequenzen seiner Aktionen kennen und ggf. nötige Vorkehrungen treffen muss, um ein Scheitern der Bearbeitung oder den Verlust des Kontaktes zu seinem Benutzer aufgrund autonom gefällter Migrations- oder Verteilungsentscheidungen zu verhindern. Insbesondere für die Migration und Distribution in dynamischen Umgebungen bedeutet dies, dass der Agent zum Beispiel das Hinterlegen von Sicherheitskopien veranlassen, Transaktionskontrollen für den Transfer beantragen oder über größere Distanzen und Zeiträume hinweg den Kontakt zu seinem Benutzer wieder herstellen können muss.

Der Rest des Kapitels gliedert sich wie folgt: der nachfolgende Abschnitt 4.1 erläutert die weitere Motivation für die kontextabhängige, eigenverantwortliche Migration unter anderem anhand der konstituierenden Charakteristiken mobiler Agenten. Im Anschluss daran folgt eine Beschreibung verschiedener Anwendungsszenarien, die die praktischen Einsatzmöglichkeiten des Konzeptes verdeutlichen sollen. Basierend auf diesen Szenarien werden in Abschnitt 4.3 die Anforderungen an den Benutzer, die Agenten und die Ausführungsumgebung zur Unterstützung des Konzeptes erarbeitet.

Nach diesen einführenden Abschnitten beginnt die Ausgestaltung des Konzeptes mit dem Abschnitt 4.4, in dem das Modell einer Ressourcenumwelt entwickelt wird. Hierzu gehören vor allem die Identifikation von Entitäten und relevanten Umweltereignissen. Dieses Modell stellt schließlich die Basis für die Resource Awareness-Komponente dar, welche als Teil der Middleware dafür verantwortlich ist, ein Umweltmodell zu erstellen und kontinuierlich zu aktualisieren. Auf die einzelnen Aufgaben dieser Komponente wird in Abschnitt 4.5 näher eingegangen. Der Aspekt der eigenverantwortlichen Migration wird in den darauf folgenden Abschnitten 4.6 und 4.7 betrachtet, in denen das zu verwendende Mobilitätsmodell spezifiziert wird und anschließend die Aufgaben einer Migrationskomponente beschrieben werden, welche als Middleware-Komponente für alle Belange der Migration sowie etwaiger Sicherungsmaßnahmen zuständig ist. Den Abschluss des Kapitels bildet die Betrachtung zweier verwandter Arbeiten, die entweder ein ähnliches Ziel mit anderen Ansätzen verfolgen oder ähnliche Ansätze in einem anderen Kontext verwenden.

4.1 Motivation

In Abschnitt 3.2.2 wurden zwei Definitionen mobiler Softwareagenten nach [Braun und Rossak 2005] gegeben. Die Definitionen unterscheiden sich in der jeweiligen Perspektive, die sie auf das Konzept eines Softwareagenten einnehmen. Die erste Definition fokussiert den Benutzer (auch als Eigentümer eines Agenten verstanden), während die zweite Definition eine eher technik-zentrierte Sicht einnimmt. Diese Definitionen lassen sich wie folgt zusammenfassen:

> *Ein mobiler Agent agiert als Repräsentant seines (nomadischen) Benutzers und kann in einem Netzwerk asynchron Aufgaben in dessen Auftrag ausführen. Zur Bearbeitung der Aufgaben kann sich der Agent im Netzwerk zusammen mit seinem Programmcode und seinem Ausführungszustand frei bewegen. Der Agent kennt seinen Benutzer und kehrt nach abgeschlossener Bearbeitung schließlich zu diesem zurück um die Ergebnisse zu präsentieren.*

Die Eigenschaften der Autonomie, Reaktivität und Proaktivität eines Agenten, wie sie als konstituierende Charakteristiken durch [Wooldridge und Jennings 1995] beschrieben wurden, finden in diesen Definitionen von mobilen Agenten keine explizite Erwähnung. Es ist unklar, ob diese

Kriterien auch auf die Eigenschaft der Mobilität an sich anzuwenden sind, was die Fragen aufwirft: 1) ob sich ein Agent autonom in einem Netzwerk, angetrieben entweder durch äußere Einflüsse (reaktiv) oder eigene Initiative (proaktiv) bewegen können sollte, 2) was die Auslöser für eine Migrationsentscheidung sind, 3) welche besonderen Maßnahmen ein Agent in dynamischen Umgebungen treffen muss, um den Verlust des Kontaktes zu seinem Benutzer zu verhindern und seine eigene Integrität zu wahren und 4) wie der Benutzer in diesem Fall den Zugriff und die Kontrolle über seine Agenten behält?

In dieser Arbeit wird argumentiert, dass eine Migrationsentscheidung eines Agenten autonom von diesem getroffen werden darf, wobei diese Entscheidung sowohl reaktiv aufgrund von Umweltereignissen als auch proaktiv aus eigenem Antrieb erfolgen kann. Da die Mechanismen der Migration neben dem Verschieben zusätzlich auch das entfernte Klonen eines Agenten erlauben, soll den Agenten darüber hinaus auch das selbstbestimmte Erzeugen von Duplikaten erlaubt werden, sofern es die Bearbeitung ihrer Aufgabe voran bringt. In beiden Fällen soll der Agent jedoch in die Pflicht genommen werden, selbst Vorkehrungen für den Fall des Verlustes der Verbindung zu seinem Benutzer zu treffen und somit eine Migration bzw. Verteilung in eigener Verantwortung durchzuführen. Zusätzlich wird gefordert, dass der Benutzer die Kontrolle über seine Agenten behalten und jederzeit, sofern eine Kommunikationsverbindung zu diesen besteht, die Agenten, ggf. auch deren Duplikate und Sicherheitskopien, in seine aktuelle Ausführungsumgebung zurück beordern können soll. Für diese Punkte sollten Agentenrahmenwerke den Entwicklern von Agenten entsprechende Funktionen als Teil ihres Rahmenwerkes (engl. *framework*) zur Verfügung stellen. In den folgenden Abschnitten werden daher eine Reihe von Anforderungen erarbeitet und nötige Komponenten in der Middleware identifiziert, die es Agenten erlauben, ihre sich verändernde Umgebung wahrzunehmen und sich autonom und eigenverantwortlich in dieser zu bewegen.

Bei der Gestaltung der Middleware-Komponenten sollen unter anderem auch die Anforderungen an die Programmschwerpunkte *Konvergenz* und *Komplexität* aus dem Förderprogramm *IT2006* des BMBF[1] berücksichtigt werden [BMBF 2007], durch die einerseits ein hohes Maß an Interoperabilität und andererseits das Verbergen der internen Komplexität des Systems gegenüber dem Benutzer gefordert wird. Die Interoperabilität ist insbesondere wichtig, da sowohl mobile (nomadische) Benutzer als auch mobile Agenten unter Umständen mit einer Vielzahl verschiedenartiger Systeme und Umgebungen konfrontiert werden, deren angebotene Ressourcen nutzbar gemacht werden sollten. Das Verbergen der Komplexität geht einher mit der Forderung nach Autonomie für mobile Agenten und soll letztendlich den Benutzer entlasten. Für den Benutzer steht schließlich der Nutzen eines Agenten im Vordergrund und je selbständiger der Agent ist, desto weniger Aufmerksamkeit und Verständnis gegenüber dem System muss der Benutzer aufbringen, wobei die grundsätzliche Möglichkeit zur Kontrolle und Einflussnahme seitens des Benutzers erhalten bleiben muss.

Alle oben aufgeführten Punkte zusammen bilden darüber hinaus die Grundlage, mobile Agenten als ständige Begleiter eines (nomadischen) Benutzers einzusetzen. Im Zuge der allgegenwärtigen Informationsverarbeitung (engl. *ubiquitous computing*) sollten nicht nur informationsverarbeitende Geräte durchgängig verfügbar sein, der Benutzer sollte auch jederzeit und überall auf seine eigenen, unter Umständen an ihn angepassten Anwendungen zugreifen können. Hierfür würde sich die Realisierung der Anwendungen mittels mobilen, autonom dem Benutzer folgenden Agenten, anbieten. Die im nächsten Abschnitt beschriebenen Anwendungsszenarien werden diesen Aspekt weiter verdeutlichen.

[1] Bundesministerium für Forschung und Bildung

4.2 Anwendungsbeispiele

In den folgenden Abschnitten werden fünf Anwendungsszenarien entwickelt, die teils unterschiedliche Aspekte und unterschiedliche Sichten auf das vorgestellte Konzept der kontextabhängigen und eigenverantwortlichen Migration abdecken. Die hier aufgeführten Beispiele sollen den Sinn und Nutzen einer Migration, abhängig von der Wahrnehmung der Umwelt, verdeutlichen und Anwendungsmöglichkeiten aufzeigen, die den Einsatz des Konzeptes in der Praxis rechtfertigen sollen.

4.2.1 Beispiel 1: Verteilte Berechnungen

Die verteilte Bearbeitung einer rechenintensiven Aufgabe ist ein praktisches Beispiel für den Einsatz mobiler Agenten. Würde man einen einzelnen stationären Agenten mit dieser Aufgabe betrauen, so hinge die Berechnungszeit allein von der lokalen Umgebung des Agenten ab. Ist das Gerät, auf dem der Agent läuft, ressourcenarm oder laufen eine Reihe anderer Prozesse nebenher, so ist entweder eine relativ lange Berechnungszeit zu erwarten oder die Berechnung kann unter Umständen auch überhaupt nicht durchgeführt werden.

Zwei interessante Möglichkeiten bieten sich in diesem Fall an, um eine schnelle Berechnung zu ermöglichen. Entweder der Agent selbst migriert auf ein anderes System, welches mehr (freie) Ressourcen bietet (zum Beispiel einen schnelleren Prozessor oder mehr Speicher) oder Teile der Aufgabe werden, sofern dies möglich ist, auf andere Systeme verteilt, dort berechnet und die Ergebnisse für eine spätere Zusammenführung wieder an das Ursprungssystem zurück gesandt. Beide Fälle stellen besondere Anforderungen an die zugrunde liegende Infrastruktur des Agenten. Zum einen müssen die Adressen (idealerweise auch die Ressourcenausstattung) anderer erreichbarer Systeme bekannt sein und zum anderen muss ein Mechanismus existieren, der es erlaubt, den Programmcode, also den zu berechnenden Algorithmus, sowie bestimmte Berechnungsparameter an ein entferntes System zu übermitteln.

Im Folgenden wird ein Szenario entworfen, welches eine verteilte Berechnung eines Algorithmus mit Hilfe mobiler Agenten präsentiert. Zur besseren Veranschaulichung wird als Aufgabe die Berechnung der Mandelbrotmenge[2] genommen. Diese Aufgabe lässt sich einfach in mehrere Teilaufgaben zerlegen, da die einzelnen Berechnungsergebnisse unabhängig von vorherigen Ergebnissen sind. Ausgangspunkt sei ein *Master*-Agent M, der auf einem Computer C_0 gestartet wird. Dieser Agent besteht aus zwei Teilen, einerseits dem Programmcode P_1 zum Berechnen der Mandelbrotmenge und andererseits dem Programmteil P_2 zum Zerlegen der Gesamtaufgabe in Teilaufgaben bzw. zum Zusammenführen der Teilergebnisse zu einem Gesamtergebnis. P_1 erwartet für die Berechnung einen Bereichsparameter, der den zu berechnenden Abschnitt der Mandelbrotmenge repräsentiert[3], P_2 verlangt die Adresse des Agenten, an den das Ergebnis nach Ende der Berechnung übermittelt werden soll.

Bevor ein Agent mit der Ausführung des Algorithmus beginnt, überprüft er zunächst, ob es von Vorteil wäre, die ihm übergebene Aufgabe als Ganzes selbst zu bearbeiten (zum Beispiel wenn der zu berechnende Bereich sehr klein ist) oder ob es ggf. sinnvoller wäre, die Aufgabe in weitere Teilaufgaben zu unterteilen. Im ersten Fall wird die Berechnung einfach durchgeführt und das Ergebnis an den auftraggebenden Agenten zurück gesandt. Im zweiten Fall jedoch kann

[2] Für eine Erklärung der Mandelbrotmenge und deren Berechnung sei unter anderem auf [Wikipedia 2007a, Munafo 2007] verwiesen

[3] Da das Gesamtergebnis der Berechnung als Bild dargestellt werden kann, eignet sich als Bereichsparameter die Start- und Endzeile des Bildes.

der Agent nach weiteren Geräten suchen, denen er eine Teilaufgabe anvertrauen kann. Einzige Bedingung ist, dass auf diesen Geräten noch kein Agent ausgeführt wird, der bereits die Mandelbrotmenge berechnet. Abhängig von der Anzahl der gefundenen Kandidaten, kann der Agent nun seine Aufgabe in gleich große (oder abhängig von der Ressourcenausstattung der Geräte auch in unterschiedlich große) Teile gliedern und die Teilaufgaben an die entfernten Geräte übermitteln.

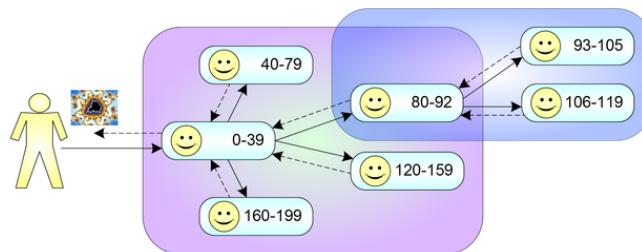

Abbildung 4.1: Anwendungsbeispiel: Verteilte Berechnung der Mandelbrotmenge. Die angegebenen Zahlenwerte repräsentieren die zu berechnenden Bildzeilen (eigene Darstellung)

Da nicht im Vorwege davon ausgegangen werden kann, dass der entsprechende Programmcode auf allen Computern vorhanden ist, soll der Programmcode in Form eines mobilen Agenten zu diesen Computern transferiert werden. Hierfür wird der Mechanismus des entfernten Klonens (engl. *remote cloning*, vgl. Abschnitt 3.1.2) verwendet, der eine exakte Kopie des Agenten an die entfernten Computer übermittelt und dort deren Ausführung startet. Die für die Ausführung benötigten Parameter werden im Rahmen des Klonprozesses an die neuen Instanzen übergeben, hierbei zeigt der Adressparameter für Programmteil P_2 auf die Agenteninstanz, die die Aufteilung der Aufgabe durchgeführt hat. Nach dem Starten der neuen Agenteninstanzen überprüfen diese wiederum, ob sie ihren Teil der Aufgabe selbst bearbeiten oder ob es sinnvoller wäre, weitere Plattformen in die Berechnung einzubeziehen (siehe oben). Auf diese Weise spannt sich ein Berechnungsbaum im Netzwerk auf. Die Teilergebnisse werden von jedem Knoten, der seinerseits Teilaufgaben vergeben hat, zusammengefasst und nach vollständiger Integration den Ästen des Baumes entlang bis zum Master-Agenten M auf dem Computer C_0 geleitet.

In dem Szenario in Abbildung 4.1 ist eine mögliche Verteilung der Berechnung dargestellt. Der Benutzer übergibt eine Aufgabe an einen Agenten, welcher daraufhin überprüft, ob es sinnvoll wäre, die ihm übertragene Aufgabe in weitere Teilaufgaben zu unterteilen. In seiner Umwelt (durch das große Rechteck dargestellt) finden sich vier freie Plattformen, die der Agent in die Berechnung mit einbeziehen kann. Er trifft die Entscheidung, alle Rechner mit einzubeziehen und lässt einen Klon seiner selbst auf jeden Rechner migrieren. Die Klone ihrerseits überprüfen auf den entfernten Rechnern wiederum, ob die Aufgabe wiederholt zerlegbar und weitere freie Ressourcen bekannt sind. Die in der Abbildung dargestellten Zahlen repräsentieren dabei die zu berechnenden Bildzeilen.

Das hier beschriebene Vorgehen wird zwar nicht zu einer optimalen Berechnungszeit führen, wohl aber zu einer möglichen Reduzierung derselben, ohne einen besonderen Mehraufwand seitens des Anwendungsprogrammierers. Die hier beschriebenen Mechanismen erlauben einen höheren Abstraktionsgrad bei der Programmierung von verteilten Berechnungen, da die Mechanismen zur Verteilung der Aufgaben im Netzwerk idealerweise von der Infrastruktur, das heißt von der Agentenplattform, bereit gestellt werden. Insbesondere hervorzuheben ist die allgemei-

ne Verwendbarkeit dieses Ansatzes. So lässt sich die Berechnung der Mandelbrotmenge einfach durch jedwede andere Aufgabe ersetzen, die Bearbeitung erfolgt dann selbstorganisiert im Netzwerk. Selbstorganisiert bedeutet in diesem Fall, dass das *Deployment* der Agenten entfällt, da sowohl Programmcode als auch Daten durch entferntes Klonen automatisch im Netzwerk verbreitet werden. Hierbei können stark belastete oder ressourcenarme Geräte von der Berechnung ausgenommen werden, falls zusätzlich die Auslastung einzelner Computer berücksichtigt wird.

Unberücksichtigt bleibt hingegen die Frage, inwiefern sich das Paradigma mobiler Agenten zum Beispiel vom Paradigma der entfernten Auswertung (engl. *remote evaluation*, vgl. Abschnitt 3.1.4) in diesem Szenario unterscheidet, und ob der Einsatz mobiler Agenten nicht überflüssig ist und zu höheren Unkosten bezüglich des Ressourcenverbrauchs führt. Hierfür sei auf Abschnitt 3.2.5 sowie [Braun u. a. 2005] verwiesen. Dort wird vorgeschlagen, den Einsatz mobiler Agenten in verteilten System der Einfachheit halber anderen Paradigmen in jedem Fall vorzuziehen und den Nachteilen mobiler Agenten (zum Beispiel die Verursachung höheren Datenaufkommens im Netzwerk) durch adaptive Mechanismen zu entgegnen. Zum Beispiel könnte man dieses Szenario weitergehend optimieren, wenn man den Programmteile P_2 nicht an jene Knoten überträgt, deren Bereichsparameter einen so kleinen Abschnitt eingrenzen, dass sich eine weitere Aufteilung der Aufgaben nicht lohnt. Auch könnte man fordern, dass die Ergebnisse nicht den Ästen des Baumes entlang versandt werden, sondern direkt dem Master-Agenten übermittelt werden. Eine weitere Möglichkeit bestünde in der kontinuierlichen Beobachtung der Umwelt, um zum Beispiel eine (Teil-)Aufgabe, ggf. während ihrer Berechnung, weiter aufzuteilen, sobald neue Computer dem Netzwerk beitreten, oder einen Agenten innerhalb des Netzwerkes auf andere Geräte zu verschieben, sobald die Auslastung eines Gerätes durch andere nebenläufige Prozesse zu hoch wird.

4.2.2 Beispiel 2: Selbstorganisierende Dienstnetzwerke

Eine Firma bietet ihren Kunden im Internet eine Reihe von Diensten an. Jeder angebotene Dienst soll ständig verfügbar sein und eine möglichst kurze Antwortzeit bieten. Anstatt teure Serverhardware anzuschaffen, sollen die ohnehin meist brachliegenden Ressourcen der Mitarbeiter-Computer für die Erbringung der Dienste genutzt werden, ohne jedoch die Mitarbeiter bei ihrer Arbeit zu beeinträchtigen[4]. Da jeder Dienst von einem (oder mehreren) Agenten angeboten wird, müssen auf allen Computern entsprechende Ausführungsumgebungen installiert werden, die automatisch im Hintergrund gestartet werden, sobald ein Mitarbeiter sich an seinem Computer anmeldet und automatisch beendet werden, sobald ein Mitarbeiter seinen Computer am Feierabend herunterfährt.

Alle laufenden Ausführungsumgebungen tauschen in regelmäßigen Intervallen per Multicast Informationen über die bei ihnen laufenden Agenten bzw. Dienste und deren Auslastung aus. Auf diese Weise hat jede Ausführungsumgebung eine komplette Sicht auf das Netzwerk mit allen angebotenen Diensten und kann bei Empfang einer Dienstanfrage, diese an einen Agenten weiterleiten, welcher einen entsprechenden Dienst erbringt. Um einerseits die Verfügbarkeit aller Dienste zu garantieren und andererseits möglichst kurze Antwortzeiten zu erreichen, haben die Agenten die Möglichkeit innerhalb des Netzwerkes zu migrieren und sich selbst auf anderen Computern zu reproduzieren. Wenn beispielsweise ein Mitarbeiter seinen Computer herunterfährt und damit die Ausführungsumgebung beendet, bekommen alle dort laufenden Agenten ein Signal und migrieren auf einen beliebigen anderen Computer. Sofern immer mindestens ein

[4] Nach dem Vorbild von *Seti@Home* [Berkeley University 2007] oder *grid.org* [United Devices 2007].

Computer eingeschaltet bleibt, ist die Verfügbarkeit sichergestellt. Ein Fehler, der zum unerwarteten Absturz eines Computers oder einer Ausführungsumgebung und damit zum Verlust eines Dienstagenten führt, wird von den anderen Ausführungsumgebungen nach Ausbleiben einer Multicast-Nachricht von dieser Plattform schnell erkannt und es können geeignete Maßnahmen (z.B. Starten einer Sicherheitskopie des Agenten oder Benachrichtigen des Administrators) ergriffen werden. Um möglichst kurze Antwortzeiten eines Dienstes zu erreichen, ist es notwendig, die Auslastung der Dienste zu überwachen. Stellt ein Agent fest, dass seine Ausführungsumgebung durch andere Diensterbringer stark ausgelastet ist, kann er von sich aus in eine andere Ausführungsumgebung migrieren. Agenten, die selber stark ausgelastet sind, können ein Duplikat ihrer selbst auf eine beliebige, vornehmlich wenig ausgelastete Ausführungsumgebung transferieren und somit für Lastverteilung sorgen. Letztendlich kann natürlich auch der Benutzer, sobald er die Ressourcen seines Computers für eigene Zwecke braucht, Agenten von seiner Plattform auf andere Plattformen migrieren lassen.

4.2.3 Beispiel 3: Ad-hoc Tauschbörsen

Herkömmliche Tauschbörsen beruhen entweder auf einem zentralen Verzeichnis oder auf der Weiterleitung von Suchanfragen durch das gesamte Netzwerk. Sie adressieren vornehmlich Weitverkehrsnetze und sind auf den Umgang mit Tausenden oder Millionen von Klienten ausgelegt. Wenn ein Benutzer eine Suchanfrage aufgibt, so wird diese Anfrage an das zentrale Verzeichnis übermittelt bzw. bei dezentralen Tauschbörsen zwischen den Knoten des Netzwerkes postuliert. Als Ergebnis erhält der Benutzer eine Ergebnisliste, aus der bestimmte Titel ausgewählt und auf das lokale Gerät geladen werden können.

Die Gegebenheiten in mobilen ad-hoc Netzwerken unterscheiden sich stark von Infrastrukturnetzwerken, wodurch auch andere Anforderungen an eine Tauschbörse entstehen. Die wichtigsten Punkte hierbei sind das Fehlen einer Infrastruktur, das unter Umständen zeitlich kurze Bestehen eines solchen Netzwerkes, die geringe Datenrate zwischen den Netzwerkknoten sowie die instabilen Verbindungen zwischen den Teilnehmern. Da ein ad-hoc Netzwerk nur wenige, oftmals sogar nur zwei Teilnehmer umfasst, ist die Wahrscheinlichkeit, auf Anhieb ein positives Ergebnis auf eine Suchanfrage zu erhalten, eher gering. Daraus folgt, dass Suchanfragen über einen längeren Zeitraum in unterschiedlichen ad-hoc Netzwerken mit wechselnden Teilnehmern gestellt werden müssen, um letztlich ein passendes Angebot zu finden.

In diesem Beispiel wird davon ausgegangen, dass der Benutzer einen Agenten mit ein oder mehreren Suchanfragen betraut und diesen dann vorerst auf dem eigenen Gerät belässt. Des Weiteren wird davon ausgegangen, dass Geräte, die sich in gegenseitiger Funkreichweite befinden, automatisch ein ad-hoc Netzwerk aufbauen, der Benutzer also nicht ständig seine Zustimmung geben muss. Auf diese Weise würde das Gerät des Benutzers im Alltag ständig Kontakt zu anderen Geräten aufnehmen können: an der Kasse im Kaufhaus, in der Straßenbahn, im Restaurant oder in der Konzerthalle.

Der Suchagent des Benutzers wird aktiv, sobald sich ein entferntes Gerät in Reichweite befindet. Zu diesem Zeitpunkt migriert jeweils ein Klon des Agenten auf alle erreichbaren Geräte und durchsucht deren Verzeichnisse auf mögliche Übereinstimmungen mit der Anfrage des Benutzers. Da die Agenten weitgehend autonom agieren sollen, entscheiden sie selbständig, mit welchen Daten sie zu ihrem Besitzer zurück migrieren sollen. Auf diese Weise sammelt ein Agent möglicherweise eine ganze Reihe verschiedener Daten auf dem Gerät seines Benutzers an, welchem es später obliegt, manuell eine Auswahl der Daten zu treffen.

4.2.4 Beispiel 4: Kaufhausagent

In diesem Szenario sollen einem Kaufhausbesucher Informationen, Hilfestellungen sowie Rabatte oder Gewinnspielchancen durch einen persönlichen Kaufhausagenten (KA) geboten werden. Der Kaufhausagent hat hierbei folgende Kompetenzen:

Wegweiser Der KA kann den Kunden durch das Kaufhaus leiten, ihm den Weg zu bestimmten gesuchten Produkten weisen und Etagenpläne anzeigen.

Produktinformationen Der Kunde kann über den KA Informationen zu sämtlichen Produkten des Kaufhauses abfragen. Dies können zum Beispiel ausführliche Produktbeschreibungen, technische Datenblätter, Preislisten, Sonderangebote, Werbespots, Inhaltsstoffe, Herkunftsorte, Informationen für Allergiker, Produktvergleiche, Kundenbewertungen etc. sein.

Gewinnspiele Der KA soll dem Kunden Gewinnspiele unterbreiten können. Hierbei werden dem Benutzer Aufgaben gestellt, die er im Kaufhaus erfüllen muss. Vorstellbar wäre zum Beispiel eine Rabatt-Rallye zu ausgewählten Produkten, wobei zu jedem Produkt eine Frage beantwortet werden muss. Nach erfolgreicher Bearbeitung der gestellten Aufgaben, stehen dem Kunden schließlich Preise oder Rabatte an der Kasse zu.

Statistiken Neben dem Mehrwert für den Kunden, kann auch das Kaufhaus von dem Agenten profitieren, da dieser durchgehend Daten erfasst, die zum Beispiel den Weg des Kunden durch das Kaufhaus aufzeigen, die Aufenthaltsdauer in bestimmten Abteilungen oder vor bestimmten Produktregalen, das Abfragen bestimmter Produktinformationen etc. Außerdem könnte die allgemeine Kundenzufriedenheit steigen, denn der Kunde kann sich im Kaufhaus besser zurecht finden, sich selbständig über Produkte informieren und ist nicht auf die Verfügbarkeit von Beratern angewiesen. Und schließlich bieten sich durch die Gewinnspiele die Möglichkeit, einem Kunden eine Reihe von Produkten zu präsentieren und zu bewerben.

Für die technische Umsetzung dieser Punkte gibt es eine Reihe von Möglichkeiten. Im Kontext dieser Arbeit stehen hierfür insbesondere mobile Agenten und mobile Geräte, sowie der Einsatz von ad-hoc Netzwerken im Vordergrund. Grundvoraussetzung ist, dass ein Kunde selbst über ein mobiles Gerät verfügt, welches die Fähigkeit besitzen muss, sich in ad-hoc Netzwerken anzumelden. Durch die zunehmende Verbreitung leistungsfähiger digitaler Assistenten (zum Beispiel *Personal Digital Assistants* oder *Smart Phones*), die über ein oder mehrere Funkschnittstellen (z.B. *Bluetooth* oder *W-LAN*) oder eine Infrarotschnittstelle zur Datenübertragung verfügen, ist diese Forderung durchaus erfüllbar. Auf Seite des Kaufhauses müssen flächendeckend Funk- bzw. Infrarotstationen platziert werden, sodass das mobile Gerät des Kunden kontinuierlich mit mindestens einer Station verbunden ist. Hierbei ist es nicht zwangsläufig erforderlich, dass die Stationen auch untereinander verbunden sind, da alle Kundendaten auf dem Gerät des Kunden gespeichert und nur mit dessen ausdrücklicher Genehmigung im Kassenbereich des Kaufhauses an einen Zentralcomputer übermittelt werden.

Im Eingangsbereich des Kaufhauses wird ein Kunde über die Vorteile eines persönlichen Kaufhausagenten informiert. Der Kunde muss selbst entscheiden, ob er einen solchen Agenten auf seinem Gerät mitnehmen möchte und dem Gerät ggf. die Verbindung zu den ad-hoc Netzwerken des Kaufhauses gestatten. Tritt das Gerät im Eingangsbereich dem dortigen ad-hoc Netzwerk bei, so erstellt der zentrale Kaufhauscomputer eine neue Instanz eines Kaufhausagenten und versieht diese mit einer eindeutigen ID, bevor die Instanz über das ad-hoc Netzwerk auf das mobile Gerät

des Kunden migriert. Um dem Kunden eine lange Wartezeit für die Übertragung aller Informationen (z.B. alle Produktbeschreibungen, Datenblätter etc.) zu ersparen, werden nur der Agent sowie allgemeine Informationen (z.B. Etagenpläne, das Gewinnspiel) an das mobile Gerät übertragen, später benötigte Informationen werden bei Bedarf von den jeweiligen Funkstationen angefordert.

Bewegt sich der Kunde mit dem Agenten durch das Kaufhaus, wechselt auch - bedingt durch die geringe Übertragungsreichweite der Funkschnittstellen - ständig das aktuelle ad-hoc Netzwerk, in dem das Gerät gerade eingebucht ist. Dies kann zur Bestimmung des ungefähren Aufenthaltsortes des Kunden verwendet werden, entweder durch Ortung aller erreichbaren Funkstationen oder über Latenzzeitmessungen bei der Nachrichtenübertragung zur Abschätzung der Entfernung zur verbundenen Funkstation. Diese Informationen kann der Agent unter anderem verwenden, um dem Kunden den Weg durch das Kaufhaus zu einem bestimmten Produkt zu weisen oder um ungefähre Bewegungsprofile des Kunden zu erstellen.

Informationen zu bestimmten Produkten kann der Kunde sich durch Eingabe der EA-Nummern (*European Article Number* oder einfach *Strichcode*) anzeigen lassen. Hierfür werden von dem Gerät benötigte Daten einfach mittels einer pull-Strategie von der nächstgelegenen Funkstation nachgeladen. Die gleichen Prinzipien werden auch bei der Teilnahme an einem Gewinnspiel verwendet. Der Kunde bekommt von seinem Agenten zum Beispiel die Vorgabe bestimmte Produkte innerhalb des Kaufhauses aufzusuchen. Dass ein Kunde sich wirklich bei dem Produkt befindet, kann der Agent durch seine bereits angesprochene Fähigkeit zur ungefähren räumlichen Orientierung feststellen. Auf dieselbe Art, wie der Agent Produktinformationen von den Funkstationen nachladen kann, werden auch Gewinnspielfragen nachgeladen, die von dem Kunden beantwortet werden müssen.

Wenn der Kunde nach Beenden seines Einkaufs an die Kasse kommt, kann er selbst entscheiden, welche Informationen des Agenten er an den zentralen Computer des Kaufhauses übermitteln möchte. Hierzu gehören zum Beispiel das Bewegungsprofil, persönliche Kundendaten (z.B. als Ersatz für eine Kundenkarte), Auswertung des Gewinnspiels etc. Da alle kundenbezogenen Daten ausschließlich lokal auf dem mobilen Gerät des Benutzers liegen, hat dieser die komplette Kontrolle und kann Missbrauch seitens des Kaufhauses ausschließen, solange er der Übertragung nicht ausdrücklich zugestimmt hat. Hat der Kunde im Laufe seines Aufenthalts im Kaufhaus Rabatte gesammelt (zum Beispiel durch Teilnahme an Gewinnspielen), so kann der zentrale Kaufhauscomputer entweder digitale Gutscheine an das mobile Gerät des Kunden übermitteln oder direkt Gutschriften im Kassensystem vornehmen. Außerdem kann der Kassencomputer weiterführende Informationen zu den gekauften Produkten (z.B. aktuelle Versionen von Handbüchern, Informationen über Garantieleistungen etc.) an den Agenten des Kunden übermitteln.

Nach Verlassen des Kaufhauses kann der Kunde schließlich selbst entscheiden, ob er seinen persönlichen Kaufhausagenten behalten oder von seinem mobilen Gerät entfernen möchte. Behält er den Agenten, so kann er fortan, auch ohne sich im Kaufhaus aufhalten zu müssen, die bereits empfangenen Produktinformationen ansehen, zum Beispiel, um etwaige Kaufentscheidungen zu Hause fällen zu können. Betritt der Kunde irgendwann ein weiteres Mal das Kaufhaus, so kann der Agent bei Bedarf auch erneut verwendet werden. In diesem Fall können detaillierte Personenprofile erstellt (wie heutzutage z.B. auch durch Kundenkarten) und an dem Einkaufsverhalten des Kunden orientierte Werbung oder Gewinnspiele präsentiert werden.

4.2.5 Beispiel 5: Mobiler Nachrichtendienst

Ein Nachrichtenagent hat die Aufgabe, für seinen Benutzer nach interessanten Nachrichten (Politik-, Wetter-, Staunachrichten etc.) im Internet suchen. Hierfür besucht er in regelmäßigen Zeitintervallen eine Menge vorgegebener Internetseiten, extrahiert bestimmte Informationen und präsentiert diese seinem Benutzer in angemessener Weise. Der Benutzer kann die präsentierten Nachrichten bewerten, um dem Agenten anzuzeigen, ob die Inhalte für ihn interessant waren oder nicht. Auf diese Weise lernt der Agent im Laufe der Zeit die Interessen seines Benutzers und kann in Zukunft auf Basis des erlernten Wissens schon im Vorwege uninteressante Nachrichten herausfiltern.

Für den Benutzer ist es daher von besonderer Bedeutung, diese bereits an die eigenen Interessen adaptierte Agenteninstanz, für die Nachrichtenbeschaffung überall zur Verfügung zu haben. Es ist durchaus vorstellbar, die Instanz an einer zentralen Stelle im Internet zu hinterlegen und je nach Aufenthaltsort des Benutzers, diesen Agenten auf das lokal zur Verfügung stehende Gerät migrieren und von dort aus seine Arbeit verrichten zu lassen. Praktischer jedoch wäre es, wenn der Agent seinen Benutzer ständig begleiten und ihm überall (zum Beispiel im Auto, in der Bahn, zu Hause und im Büro etc.) auf Anforderung die gesammelten Nachrichten präsentieren könnte. Außerdem könnte der Agent, sofern er unterwegs einen Internetzugang gestellt bekommt, aktuelle Nachrichten für seinen Benutzer aus dem Internet holen.

Die Bedingung in diesem Anwendungsbeispiel soll sein, dass der Benutzer ein mobiles Gerät besitzt, welches über eine Funkschnittstelle verfügt und sich automatisch in mobilen ad-hoc Netzwerken registrieren kann. Außerdem soll der Benutzer sich in keiner Weise um die Mitnahme des Agenten kümmern müssen. Der Agent muss selbst dafür Sorge tragen können, seinen Benutzer zu begleiten und muss für Ausnahmefälle (z.B. Verlust des Agenten) Vorkehrungen in Form einer Art von Sicherheitskopie treffen können.

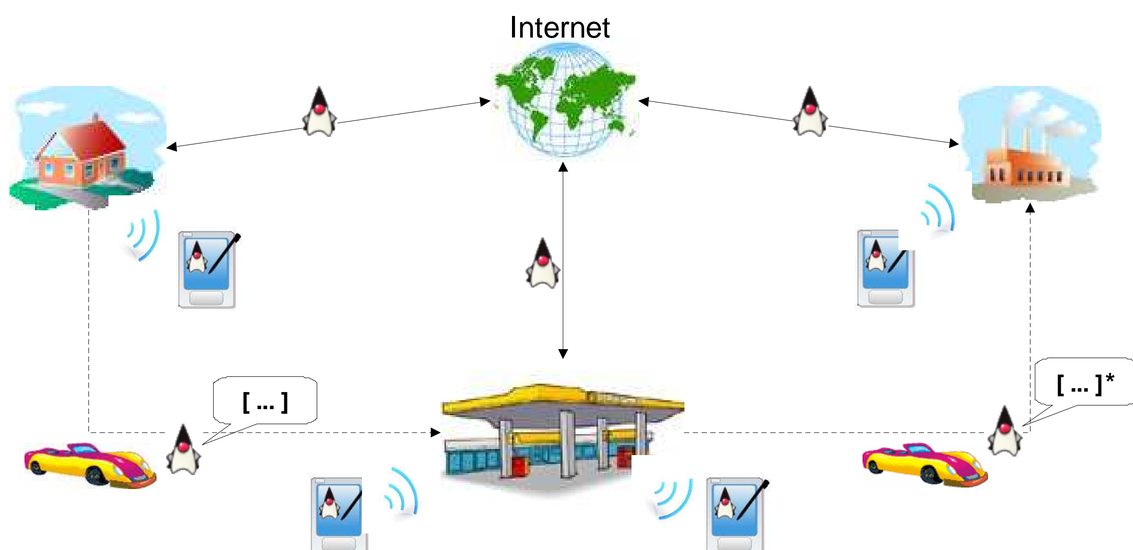

Abbildung 4.2: Anwendungsbeispiel: Mitnahme eines persönlichen Nachrichtenagenten (eigene Darstellung)

Abbildung 4.2 zeigt ein Szenario, in dem der Nachrichtenagent seinen Benutzer sowohl zu Hause, im Auto und schließlich auch bei der Arbeit über aktuelle Nachrichten informieren kann, und dabei versucht, jede sich bietende Möglichkeit der Nachrichtenbeschaffung zu nutzen. Es

wird davon ausgegangen, dass der Benutzer morgens, bevor er sich auf den Weg zur Arbeit macht, seinen Nachrichtenagenten auf dem heimischen Computer startet. Dieser verbindet sich mit den entsprechenden Internetseiten, extrahiert die Informationen und filtert sie vorab nach den Interessen des Benutzers. Der Agent beendet seine Arbeit sobald er ein Anzeichen dafür wahrnimmt, dass der Benutzer sich auf den Weg zur Arbeit begibt. Dies könnte zum Beispiel das Herunterfahren des Computers oder ein Absinken der Verbindungsqualität vom Computer zum mobilen Gerät des Benutzers sein. Wird ein solches Ereignis durch den Agenten wahrgenommen, migriert er möglichst verzögerungsfrei und effizient auf das mobile Gerät, d.h. er migriert bestenfalls nur mit seinem Zustand sowie mit neuen Nachrichten auf das Gerät, sofern der Programmcode bereits auf dem Gerät vorhanden ist.

Während der Autofahrt ins Büro, kann der Benutzer sich die aktuellen Nachrichten von dem Agenten laut vorlesen lassen und somit die Zeit sinnvoll nutzen. Auf dem Weg muss der Benutzer eine Tankstelle anfahren, um neues Benzin aufzunehmen. Die Tankstelle bietet ihren Kunden als besonderen Service, einen öffentlichen Zugangspunkt für Agenten, inklusive eines Internetanschlusses an. Sobald das mobile Gerät sich in Funkreichweite der Basisstation der Tankstelle befindet und Informationen über die dort angebotenen Ressourcen eingeholt hat, gibt es die Information über den nutzbaren Internetzugang an den Nachrichtenagenten weiter.

Um die Energieressourcen des mobilen Gerätes zu schonen, entschließt sich der Agent, auf den Zugangspunkt der Tankstelle zu migrieren, um von dort neue Nachrichten aus dem Internet zu laden und die empfangenen Daten zu filtern. Da der Agent sich in eine nicht vertrauenswürdige Umgebung begibt, veranlasst er eine Sicherheitskopie seiner selbst, für den Fall des Verlustes, auf dem mobilen Gerät zu erstellen. Nach Beendigung der Arbeit auf dem Zugangspunkt oder bei drohendem Verlust der Funkverbindung zu dem mobilen Gerät des Benutzers, migriert der Agent zusammen mit den neuen Nachrichten zurück auf das mobile Gerät.

Im weiteren Verlauf der Fahrt kann der Agent dem Benutzer die neuen Nachrichten, ggf. zum Beispiel auch Staumeldungen etc., präsentieren. Wenn der Benutzer schließlich seinen Arbeitsplatz erreicht und den dortigen Computer einschaltet, nimmt der Agent die neu zur Verfügung stehenden Ressourcen wahr, migriert auf den Arbeitsplatzcomputer und verrichtet seine Aufgaben dort solange, bis sich der Benutzer mit seinem mobilen Gerät wieder entfernt.

4.3 Anforderungsanalyse

Aus den im vorigen Abschnitt aufgeführten Beispielszenarien lassen sich eine Reihe von Anforderungen ableiten. Für eine bessere Übersicht sollen diese Anforderungen in vier Gebiete unterteilt werden: 1) Anforderungen an den Benutzer, 2) Anforderungen an einen Agenten, 3) Anforderungen an die Infrastruktur und 4) allgemeine Anforderungen an die Sicherheit.

An den Benutzer sollen möglichst wenig Anforderungen gestellt werden, da dieser schließlich von dem Konzept profitieren soll. Idealerweise arbeiten die Mechanismen derart transparent, dass der Benutzer die Funktionsweise nicht unbedingt verstehen muss, den Effekt aber sofort erkennen kann. Auch an die Agenten sollen keine allzu hohen Anforderungen gestellt werden, wenn auch zwangsläufig ungleich mehr als an den Benutzer. Die Anforderungen an den Agenten sind für die Programmierer von Bedeutung, da sie mit neuen Möglichkeiten und Funktionen der Infrastruktur umgehen, nicht jedoch die dahinter stehenden Mechanismen im Detail verstehen müssen. Die weitaus höchsten Anforderungen werden an die Infrastruktur bzw. die Ausführungsumgebung gestellt. Unter Ausführungsumgebung sei an dieser Stelle in erster Linie nicht

die Hardware-, sondern die Softwareinfrastruktur und insbesondere die Agentenplattform, verstanden, denn sie ist für die komplette Umsetzung der Mechanismen verantwortlich und muss die entsprechenden Schnittstellen für die Agenten implementieren. Zuletzt sollen auch noch einige Anforderungen an die Sicherheit formuliert werden, denn ein praktischer Einsatz des Konzeptes ist ohne Berücksichtigung grundlegender Sicherheitsüberlegungen nicht verantwortbar.

4.3.1 Benutzer

Das Konzept der autonomen, eigenverantwortlichen Migration soll für den Benutzer einen Mehrwert und keine zusätzliche Belastung darstellen. Im Idealfall bemerkt der Benutzer nicht einmal, dass sein Agent zeitweilig auf fremden Systemen ausgeführt wird, um die aufgetragene Aufgabe zu bearbeiten. Für den Benutzer steht die möglichst schnelle Fertigstellung der Bearbeitung im Vordergrund, sowie die Gewissheit, möglichst überall auf seinen Agenten zugreifen zu können, egal ob er sich an einem festen Ort (z.B. zu Hause, im Büro etc.) aufhält oder in irgendeiner Weise unterwegs ist (z.B. im Auto, im Restaurant etc.). So kann sich der Benutzer vollständig auf die Funktion des Agenten konzentrieren und sollte sich keine Gedanken über die Mechanismen machen müssen, die es dem Agenten erlauben, autonom seine Aufenthaltsorte zu wechseln, um seinem Benutzer zu folgen oder sich im Netzwerk zur Beschleunigung der Bearbeitung zu replizieren.

Insofern sollen an den Benutzer keine besonderen Anforderungen gestellt werden. Er muss sich lediglich darüber bewusst sein, dass Agenten die Fähigkeit besitzen, ihm zu folgen und sich bei Bedarf auf fremden Systemen zu replizieren. Dass keine Anforderungen gestellt werden, bedeutet allerdings nicht, dass dem Benutzer keine Möglichkeiten zur Kontrolle eingeräumt werden sollen. Natürlich muss der Benutzer, sofern er es wünscht und sofern dies möglich ist, jederzeit die vollständige Kontrolle über seine Agenten behalten. Hierzu gehört, dass er jederzeit die letzten sowie den aktuellen Aufenthaltsort des Agenten, etwaiger Klone und Sicherheitskopien sowie Daten erfragen kann, diese sofort zu einem Ausführungsort seiner Wahl transferieren oder löschen können soll, und den Agenten bezüglich ihres Verhaltens Vorschriften auferlegen kann.

Falls die Agenten als ständige Begleiter eingesetzt werden sollen, muss der Benutzer natürlich über ein mobiles Gerät verfügen, welches die Agenten aufnehmen kann, und welches der Benutzer entsprechend mit sich führt. Dies ist allerdings keine zwingende Anforderung, da es auch möglich sein sollte, von jedem stationären Gerät aus, entfernt ausgeführte persönliche Agenten auf dieses Gerät zu transferieren, unabhängig davon, ob der Transfer von dem Benutzer initiiert oder proaktiv durch den Agenten durchgeführt wird.

4.3.2 Agenten

Ausgehend davon, dass an den Benutzer keine besonderen Erwartungen gestellt werden sollen, müssen jedoch an die Agenten eine Reihe von Anforderungen gestellt werden. Auch diese sollen auf ein möglichst geringes Maß beschränkt werden, da zu hohe Anforderungen die Realisierung von Anwendungen für einen Programmierer erschweren könnten.

Autonomie Zu den Anforderungen an einen Agenten zählt unter anderem die Forderung nach der Autonomie eines Agenten, welche auch bereits eine der konstituierenden Charakteristiken eines Agenten darstellt. Ein Agent muss autonom eine Entscheidung für eine Migration oder Distribution fällen können, da einerseits nicht sichergestellt werden kann, dass ein Agent in dyna-

mischen Umgebungen durchgängig eine Kommunikationsverbindung zu seinem Benutzer aufrecht erhält, und andererseits der Benutzer nicht durch ständige Rückfragen zusätzlich belastet werden soll. Zwar soll der Benutzer, abhängig vom Einsatzkontext, einem Agenten vorgeben können, dass dieser keine Aktion ohne vorherige Rückfrage ausführen darf, aber prinzipiell soll der Agent in der Lage sein, selbständig Aktionen durchzuführen, bis eine für ihn nicht lösbare Situation auftritt.

Eigenverantwortlichkeit Mit der Autonomie geht die Forderung nach der Eigenverantwortlichkeit des Handelns einher. Agenten müssen selbst entscheiden, wann sie ihren Ausführungsort zu wechseln haben, wohin sie migrieren müssen und ob sie sich im Netzwerk replizieren sollen. Besteht Unsicherheit bezüglich des Ausgangs einer Situation oder die Gefahr, dass ein Agent den Kontakt zu seinem Benutzer aufgrund unvorhergesehener Ereignisse verliert, muss der Agent bereits im Vorfeld Maßnahmen ergreifen, die das Risiko eines Verlustes minimieren. Hierzu gehören zum Beispiel das Hinterlegen von Sicherheitskopien sowie die redundante Verteilung von Daten. Außerdem ist ein Agent, der sich selbst im Netzwerk repliziert, für seine Duplikate verantwortlich. Das bedeutet, es muss einerseits vermieden werden, dass nach Fertigstellung einer Aufgabe Duplikate längerfristig im Netzwerk verbleiben und andererseits muss der Agent die Fähigkeit besitzen, nach Verteilung von Teilaufgaben an entfernte Duplikate, deren Ergebnisse zu einem späteren Zeitpunkt wieder zusammenführen (engl. *mergen*) zu können.

Umweltgewahrsein Des Weiteren ist für einen Agenten die Wahrnehmung seiner Umwelt sowie ein Verständnis ihrer grundlegenden Prinzipien erforderlich. Hierfür muss den Agenten ein Umweltmodell seitens der Ausführungsumgebung bereit gestellt werden, in dem Entitäten und Ereignisse innerhalb der Umwelt repräsentiert sind. Den Agenten obliegt es dann, diese Informationen zu verarbeiten, zu bewerten und Schlüsse für ihr eigenes Handeln daraus zu folgern. Dies bedeutet unter anderem, dass der Agent im Anschluss an eine Lernphase, die Ressourcen seines Benutzers (z.B. dessen mobiles Gerät) gegenüber fremden Ressourcen unterscheiden können muss. Aus Ereignissen, die diese Ressourcen betreffen, muss der Agent notwendige Aktionen ableiten können. Zum Beispiel könnte eine Verminderung der Verbindungsqualität zu dem mobilen Gerät des Benutzers einen drohenden Zugriffsverlust und das Abreißen des Kontaktes zu dem Benutzer bedeuten, und der Agent muss kontextabhängig entscheiden, ob (und ggf. wie) er auf das mobile Gerät migrieren oder zumindest eine Kopie seiner selbst dorthin übertragen soll.

4.3.3 Ausführungsumgebung

Um die oben vorgestellten Abstraktionsebenen für den Benutzer und die Agenten bieten zu können, wird ein Großteil der Anforderungen an die Ausführungsumgebung gestellt. Diese muss alle Mechanismen implementieren, die ein Agent im weitesten Sinne für die autonome und eigenverantwortliche Migration und Distribution benötigt.

Resource Awareness Beginnend mit einer Sicht auf die Umwelt, muss die Ausführungsumgebung den Agenten kontinuierlich ein Modell ihrer Umgebung bereitstellen. In diesem Modell müssen alle verfügbaren Ressourcen sowie weitere Umweltparameter (z.B. Art des Netzwerkes, Verbindungsqualitäten, ggf. auch ungefähre relative/absolute/logische Ortsangaben) enthalten sein und bei Eintreten bestimmter Ereignisse aktualisiert werden. Diese Informationen kann ein

Agent nicht nur zur Erfüllung seiner eigentlich intendierten Aufgabe nutzen (z.B. für verteilte Berechnungen), sondern vor allem auch für Migrations- und Verteilungsentscheidungen.

Die Fragen, *welche Informationen* konkret enthalten sein sollen und was unter *verfügbare Ressourcen* zu verstehen ist, lässt sich an dieser Stelle nur allgemein beantworten und wird erst in Abschnitt 4.4 weitergehend behandelt. Grundsätzlich sollte die Ausführungsumgebung die Möglichkeiten bieten, so viele Informationen wie möglich in das Umweltmodell zu integrieren. Das heißt, es sollten dem Modell möglichst wenige Einschränkungen auferlegt werden. Vorstellbar wäre zum Beispiel die Aufnahme detaillierter Informationen über die Hardware- und Softwareausstattung von Geräten, ggf. aktuelle Leistungsdaten (z.B. Speicherverbrauch, Anzahl laufender Prozesse, Antwortzeiten etc.) sowie genaue Informationen über laufende Agenten und angebotene Dienste in einer Ausführungsumgebung. Der Benutzer bzw. der Programmierer sollte jedoch auch die Möglichkeit haben das Umweltmodell mit eigenen, anwendungsbezogenen Informationen anzureichern, zum Beispiel verfügbare Mediendateien oder Dokumente, aktuelle Wetterdaten am jeweiligen Ort einer Ausführungsumgebung etc. Dem Benutzer bzw. den Agenten obliegt es letztendlich eine interessante und wesentliche Auswahl an Informationen zu treffen.

Zur Erstellung und Aktualisierung des Umweltmodells tauschen die Ausführungsumgebungen untereinander Informationen aus. Um die Skalierbarkeit dieses Ansatzes zu gewährleisten, muss eine Ausführungsumgebung entsprechende Verteilungsmechanismen einsetzen. Das provisorische Versenden aller lokal verfügbaren Informationen an alle potenziell interessierten Nachbarn wird nicht skalieren. Aus diesem Grunde soll die Ausführungsumgebung einerseits die Möglichkeit bieten, dass der Benutzer den Umfang auszutauschender Informationen manuell begrenzen kann, und andererseits müssen die grundlegenden Mechanismen des Austausches so flexibel gestaltet werden, dass verschiedene Verfahren unter Verwendung entsprechender Adapter unterstützt werden können. Vorstellbar wäre zum einen, bestimmte Informationen prinzipiell nur auf Anfrage anstatt proaktiv zu verschicken, zum anderen könnten unterschiedliche Discovery-Verfahren für ad-hoc bzw. Infrastrukturnetzwerke (vgl. Abschnitte 2.3 und 2.4) eingesetzt werden, die im jeweiligen Kontext eine gute Skalierbarkeit gewährleisten.

Um auf Änderungen des Zustandes der Umwelt reagieren zu können, muss es einem Agenten möglich sein, sich bei der Ausführungsumgebung zu registrieren, um bei Eintreten bestimmter Ereignisse sofort informiert zu werden. In diesem Fall wird der Agent unter anderem beim Auffinden neuer Ressourcen, Verschwinden bekannter Ressourcen, Beenden der Plattform, Änderung der Verbindungsqualität zu bestimmten Ressourcen etc. benachrichtigt und kann daraufhin beispielsweise seine Ausführungsumgebung wechseln, entfernte Dienste in Anspruch nehmen oder sich in anderen Umgebungen reproduzieren. Entfernt sich zum Beispiel der Benutzer mit seinem mobilen Gerät vom Arbeitsplatzcomputer, so muss die Ausführungsumgebung des Arbeitsplatzcomputers nach Auswertung der Verbindungsqualität allen für dieses Ereignis registrierten Agenten mitteilen, dass der Benutzer wahrscheinlich den Funkbereich verlassen wird und Agenten, die den Benutzer begleiten möchten, auf das mobile Gerät migrieren müssen. Unter anderem anhand von Umweltparametern kann die Ausführungsumgebung dann Vorgaben für die Art der Migration machen: bei schnell sinkender Verbindungsqualität in ad-hoc Netzen wäre es vorstellbar, dass nur der Agent ohne Daten asynchron[5] migriert und die Daten nachträglich über eine pull-Strategie nachgeladen werden, solange die Verbindung noch besteht.

Wenn unterschiedliche Verfahren zur Verteilung von Informationen eingesetzt werden können, muss auch entsprechend mit der möglichen Vielfalt an Repräsentationsformen umgegangen werden. Die meisten Discovery-Verfahren verwenden proprietäre Formate für die Repräsentation

[5]Asynchrone Migration erwartet keine Bestätigung des Empfängers und ist daher schneller, siehe Abschnitt 3.1.2

von Ressourcen, einige wenige Verfahren hingegen beruhen auf Standardformaten, zum Beispiel die in Abschnitt 2.5 vorgestellten Formate *RDF*, *OWL* oder *WSDL*. Da die Ausführungsumgebung die Informationen, die möglicherweise von unterschiedlichen Verfahren gesammelt werden, in einem zentralen Umweltmodell integrieren soll, muss sie entsprechend auch die jeweiligen Formate verarbeiten können. Hinzu kommt, dass eine Ausführungsumgebung bei Kontakt mit entfernten Ausführungsumgebungen eine Auswahl relevanter Informationen treffen können soll, und damit geht die Forderung nach Unterstützung verschiedener Abfragesprachen einher.

Um der Heterogenität der Umwelt und der damit einhergehenden Vielfalt an Entitäten, Repräsentationsformen und Verfahren zum Austausch von Informationen gerecht zu werden, bedarf es außerdem der Möglichkeit, Verfahren und Repräsentationsformen je nach Kontext adaptiv auswählen zu können. Auch muss die Möglichkeit geboten werden, neue Verfahren und Modelle statisch oder dynamisch in die Ausführungsumgebung zu integrieren, um einerseits auf wechselnde Umgebungen reagieren zu können und andererseits offen für den technischen Fortschritt zu bleiben.

Migrationsmechanismen Wie bereits in den Abschnitten 3.1.2 und 3.3.2 erläutert wurde, gibt es eine Reihe unterschiedlicher Migrationsmechanismen, die sowohl die Ausführungsumgebung als auch die Agenten direkt einsetzen können. Hierzu gehören beispielsweise die Entscheidung über die Art der Migration (schwach oder stark), die Migrationsstrategie (push, pull oder selbstdefinierte Strategien) sowie ggf. die Verwendung bestimmter Sicherheitsmechanismen für den Fall unvorhergesehener Ausnahmesituationen (transaktionale Migration, Sicherheitskopien etc.).

Wenn ein Agent sich entscheidet, den aktuellen Ausführungsort zu wechseln oder bestimmte Daten an einen oder mehrere andere Ausführungsumgebungen zu versenden, muss das Kopieren und das Verschieben, sowohl des Programmcodes als auch des Datenraumes eines Agenten, an entfernte Ausführungsumgebungen unterstützt werden. Außerdem bedarf es der Möglichkeit, sowohl für die starke Migration oder das entfernte Klonen eines Agenten als auch für die Erstellung von Sicherheitskopien, den Zustand eines Agenten auszulesen, zu manipulieren und persistent zu machen bzw. den Agenten aus einem persistenten Zustand heraus neu zu starten[6].

Die Migration soll jedoch nicht nur von dem Agenten selbst angestoßen werden können, auch die Ausführungsumgebung bzw. der Benutzer soll eine Art Zwangsmigration anweisen können. Eine derartige Art der Migration könnte in zweierlei Hinsicht von Bedeutung sein. Falls ein Agent sich in eine Ausführungsumgebung hat migrieren lassen, in der er im Nachhinein nicht willkommen ist (z.B. weil er zu viele Ressourcen beansprucht, die Sicherheit gefährdet etc.), soll diese den Agenten auf seine Heimatplattform zurück schicken können. Ist diese nicht erreichbar, kann die Ausführungsumgebung den Agenten auch in einen persistenten Zustand versetzen und zu einem späteren Zeitpunkt transferieren, oder der Agent wird lokal in einen sogenannten Sandkasten (engl. *sandbox*) verschoben, wo er lediglich eingeschränkten Zugriff auf die Ausführungsumgebung und deren Ressourcen hat. Auf der anderen Seite kann eine Zwangsmigration auch von dem Benutzer angestoßen werden, wenn dieser den Agenten von einer entfernten Ausführungsumgebung zu sich beordert oder von der lokalen Ausführungsumgebung explizit an eine entfernte Umgebung übersenden möchte. Eine solche Zwangsmigration kann auch nach Ablauf einer bestimmten Zeitspanne von der Ausführungsumgebung ausgelöst werden, sodass Agenten zum Beispiel nur wenige Minuten Zeit für die Bearbeitung ihrer Aufgaben haben und anschlie-

[6]Probleme, die durch die Möglichkeit der gleichzeitigen Ausführung identischer Agenteninstanzen in unterschiedlichen Ausführungsumgebungen entstehen, können nicht weitergehend behandelt werden, da sie den Rahmen der Arbeit übersteigen und einen eigenen Forschungsgegenstand darstellen.

ßend automatisch wieder zurückgeschickt oder ggf. solange persistent gemacht werden, bis sie explizit von dem Benutzer wieder nachgefragt werden.

Mit der Migration eines Agenten stellt sich auch das Problem des Nachrichtenversands an diesen Agenten, da dessen Aufenthaltsort dem Absender nicht zwangsläufig bekannt ist. Möchte ein Benutzer zum Beispiel einen entfernt ausgeführten Agenten zurück auf sein eigenes Gerät beordern, so adressiert er eine Nachricht an diesen Agenten und verschickt sie ins Netzwerk ohne den genauen Bestimmungsort zu kennen. Es ist die Aufgabe der Ausführungsumgebungen, die Nachricht solange weiterzuleiten, bis sie den Agenten schließlich erreicht. Verschiedene Ansätze sind hierfür vorstellbar, grundlegend ist jedoch, dass eine Ausführungsumgebung in irgendeiner Form ein Protokoll über Migrationen und Distributionen von Agenten führen muss, um einem Agenten Nachrichten hinterher schicken zu können.

Parametrisierbare Profile Abhängig von den aktuellen Gegebenheiten der Umwelt, muss die Ausführungsumgebung ihre Mechanismen zum Beispiel für das Auffinden von Ressourcen und für die Migration von Agenten etc. flexibel anpassen können, und daher eine Reihe unterschiedlicher Verfahren sowohl für Infrastruktur- als auch mobile ad-hoc Netzwerke unterstützen. Die Verwendung der einzelnen Modelle und Mechanismen soll außerdem auch von dem Benutzer angepasst werden können, sodass ein hoher Grad an Konfigurierbarkeit seitens der Ausführungsumgebung gefordert wird. Zum Beispiel muss der Benutzer bestimmte Einstellungen vornehmen können, um den Verbrauch von Ressourcen durch einzelne Mechanismen einzuschränken. Beispielsweise könnte die Aktualisierung des Umweltmodells, zugunsten geringeren Übertragungsvolumens bei einer teuren UMTS-Funkverbindung, seltener vorgenommen werden als in lokalen Infrastrukturnetzwerken, oder das Erstellen von Sicherheitskopien der Agenten wird aufgrund geringen Speicherplatzes verboten. Da jedoch jeder einzelne Mechanismus durch eine Reihe von Parametern konfigurierbar ist, vom Benutzer jedoch nicht zwangsläufig die Bedeutung jedes Parameters verstanden wird, sollten parametrisierbare Profile, die Voreinstellungen für verschiedene Szenarien bieten, die Handhabung der Einstellungen für den Benutzer vereinfachen und beschleunigen.

4.3.4 Sicherheit

Die Sicherheit spielt in offenen verteilten Systemen eine wichtige Rolle und umfasst eine ganze Reihe von Aspekten, die sowohl bei dem System- als auch dem Anwendungsentwurf eine bedeutende Rolle spielen sollten. Obwohl es schwierig ist, Sicherheitskonzepte nachträglich in ein System einzubauen, können etwaige Sicherheitsbelange in dieser Arbeit nicht näher berücksichtigt werden, da sie den Rahmen dieser Arbeit übersteigen würden. Dennoch sollen an dieser Stelle einige Gefahren für die Sicherheit angedeutet werden, ohne deren Beachtung ein praktischer Einsatz des hier vorgestellten Konzeptes nicht verantwortbar ist. Überlegungen, an welchen Stellen etwaige Sicherheitsmaßnahmen in die Mechanismen und Modelle integriert werden könnten, sollen jedoch nicht vertieft werden.

Bei der Sicherheit lassen sich drei schützenswerte Rollen unterscheiden. An erster Stelle stehen hierbei der Benutzer und dessen persönliche Daten, dann erst folgen der Schutz von Agenten und von Ausführungsumgebungen. Das Gewahrsein über die Umwelt, erlaubt es einem Agenten, den Aufenthaltsort eines Benutzers bzw. seiner Geräte ständig zu überwachen, und in Verbindung mit der kontextabhängigen Migration diesen sogar in gewisser Hinsicht zu verfolgen. Sofern es sich hierbei um nicht vertrauenswürdige Agenten handelt, kann der Benutzer nicht sicher sein, dass

die Agenten keine persönlichen Daten (z.B. Aufenthaltsorte, Bewegungsprofile etc.) erfassen und an beliebige Institutionen übermitteln. Es gibt jedoch auch nützliche Einsatzszenarien für Agenten als ständige Begleiter, in denen sie versuchen, dem Benutzer ständig und überall zur Bearbeitung von Aufgaben zur Verfügung zu stehen (siehe Abschnitt 4.2). Doch auch wenn die Agenten vertrauenswürdig sind, besteht immer die Gefahr, dass persönliche Daten durch eine böswillige Ausführungsumgebung gelesen und ggf. manipuliert werden. So wäre es zum Beispiel im Anwendungsszenario des Nachrichtenagenten (siehe Abschnitt 4.2.5) durchaus möglich, dass eine fremde Ausführungsumgebung das Wissen des Agenten um die Interessen seines Benutzers ausspioniert. Diesbezüglich kann keine Maßnahme vollständige Sicherheit garantieren, eine Reihe von Techniken erlauben es jedoch, den Programmcode und die Daten eines Agenten möglichst gut vor der Ausführungsumgebung zu schützen. Hierzu gehören zum Beispiel Verschleierungs- (engl. *obfuscation*) und Verschlüsselungstechniken.

Durch die geforderte Offenheit des Systems und der Möglichkeit, dass Agenten sich auf beliebige Plattformen bewegen und sogar die Fähigkeit haben, sich zu duplizieren, ergibt sich das Risiko ganz einfacher sogenannter (verteilter) *Denial of Service*-Attacken (dt. Dienstverweigerungsattacken), die durch absichtlich übermäßig hohen Ressourcenverbrauch seitens der Agenten herbeigeführt werden können. Mit diesem Problem einher geht die allgemeine Frage der Autorisierung: was darf ein Agent? Vorstellbare wäre, dass eine Ausführungsumgebung ohne Authentisierung und Authentifizierung die Agenten in einem Sandkasten (engl. *sandbox*) ausführt, in dem sie lediglich beschränkte Zugriffsrechte auf die lokalen Ressourcen erhalten. Ausschließlich vertrauenswürdigen und authentifizierten Agenten sollte ggf. ein unbeschränkter Zugriff gestattet werden. Doch nicht nur Agenten sollten sich authentisieren müssen, auch Ausführungsumgebungen und letztlich sogar ein Benutzer müssen sich gegenüber den Agenten identifizieren. Zum Beispiel muss ein Benutzer die Möglichkeit haben, seine persönlichen Agenten zu kontrollieren und diese, falls sie in entfernten Ausführungsumgebungen laufen, jederzeit in seine lokale Umgebung zu beordern. Hierfür muss jedoch die Identität des Benutzers, entweder durch die entfernte Ausführungsumgebung oder durch den Agenten selbst, verifiziert werden können.

Insgesamt lassen sich vielerlei zu beachtende Sicherheitsaspekte ausmachen, zusammenfassend können hierfür verschiedene Maßnahmen zur Verschlüsselung, Verschleierung, Autorisierung, Authentisierung und Authentifizierung ergriffen werden. Für eine umfassende Übersicht an vorstellbaren Bedrohungen und daraus resultierenden Sicherheitsanforderungen und möglichen Gegenmaßnahmen, sei an dieser Stelle auf weiterführende Literatur verwiesen, unter anderem auf [Vigna 1998, Jansen und Karygiannis 1999, Jansen 2000, Borselius 2002, Zhu u. a. 2005]

4.4 Spezifikation des Umweltmodells

Bei der Anforderungsanalyse des vorhergehenden Abschnittes wurden eine Vielzahl an Anforderungen identifiziert, die sich in Teilen auf das Umweltmodell beziehen. Das Umweltmodell eines Agenten bzw. einer Ausführungsumgebung umfasst in erster Linie Informationen über Entitäten und Ereignisse in der Umwelt, welche für einen Agenten zur Ausführung von Aktionen von Interesse sein könnten. In den folgenden beiden Abschnitten werden daher die Entitäten und Ereignisse identifiziert, welche die Grundlagen des Modells einer Ressourcenumwelt repräsentieren.

4.4.1 Entwicklung eines ressourcenbasierten Umweltmodells

In Abschnitt 2.2.2 wurden mit der natürlichen, der logischen und der sozialen Umwelt bereits die Strukturen sowie die Prinzipien und Prozesse dreier Umweltmodelle vorgestellt. Im Anschluss daran wurde das allgemeine, dreischichtige Umweltmodell von [Weyns u. a. 2004, Weyns u. a. 2005] für den Einsatz in Multiagentensystemen betrachtet. Von diesem Modell ausgehend, soll im Folgenden ein ressourcenbasiertes Umweltmodell entwickelt werden, welches fortan als Grundlage für weitere Betrachtungen innerhalb dieser Arbeit dient. Das zu entwickelnde Umweltmodell soll im Kontext der Resource Awareness den Agenten die Wahrnehmung der für sie interessanten Ressourcen in ihrer Umwelt erlauben. Aufgrund der potenziell hohen Anzahl der im Modell enthaltenen Informationen, ist es jedoch nicht vorteilhaft, wenn jeder Agent sein eigenes Umweltmodell aufbaut und ständig mit sich führt. Stattdessen soll die Ausführungsumgebung für den Aufbau und die Aktualisierung des Umweltmodells verantwortlich sein und den Agenten lesenden Zugriff auf dieses Modell erlauben.

Um die Anwendungsunabhängigkeit zu wahren, soll sich das Modell in erster Linie auf allgemeine Ressourcen beschränken. Hierzu gehören Ressourcen, an denen jeder Agent, unabhängig von seinem konkreten Einsatzkontext, interessiert sein könnte. Jedoch soll es prinzipiell auch möglich sein, zusätzliche anwendungsabhängige Informationen in dieses Modell zu integrieren, sodass eine Anwendung, die ebenfalls auf der Verbreitung von Informationen basiert, die bereits bestehenden Mechanismen zur Verbreitung von Umweltinformationen nutzen kann. Auf diese Weise ließen sich auch weitergehende Konzepte, z.B. einer sozialen Umwelt (siehe Abschnitt 2.2.2), in das Umweltmodell eingliedern. Auch konkrete Anwendungen könnten so das Modell erweitern, als Beispiel hierfür seien Tauschbörsen genannt, die Informationen über angebotene Dokumente als Teil des Umweltmodells bereitstellen könnten.

Wie bereits erwähnt, stellt das Modell von Weyns die Grundlage für das Modell der Ressourcenumwelt dar. Das Modell der Ressourcenumwelt erweitert Weyns Modell lediglich in einigen Punkten, um es dem Kontext dieser Arbeit anzupassen. Dieses erweiterte Umweltmodell ist in Abbildung 4.3 dargestellt.

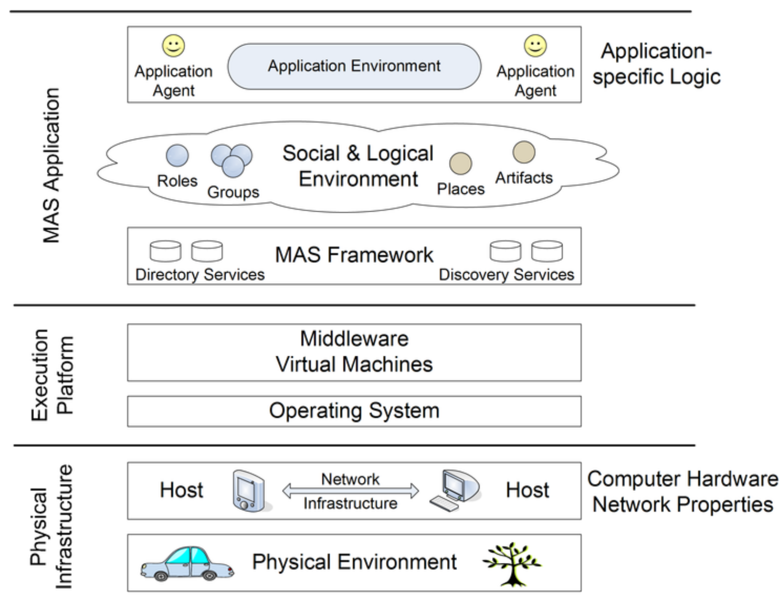

Abbildung 4.3: Erweitertes allgemeines Umweltmodell für MAS (eigene Darstellung)

Folgende Unterschiede lassen sich im Vergleich mit dem Modell von Weyns ausmachen:

- Verzeichnis- und Discoverydienste werden als wichtige Bestandteile eines Multiagentensystems explizit in der Schicht des Multiagentenrahmenwerks aufgeführt, da die Wahrnehmung der Umwelt, wie zuvor erwähnt, Aufgabe der Ausführungsumgebung für Agenten ist. Die Frage, ob derartige Dienste nicht eigentlich Teil der allgemeinen Middleware sind, wird in Abschnitt 5.2.2 diskutiert. An dieser Stelle wird der Einfachheit halber davon ausgegangen, dass die Verwaltung eines Verzeichnis- oder Discoverydienstes von dem Multiagentensystem übernommen wird und diese Dienste nicht unabhängig angeboten werden.
- In [Weyns u. a. 2005] heißt es, dass eine Sicht auf benachbarte Netzwerkknoten und auf logische Verbindungen innerhalb des Netzwerkes in der Anwendungsumwelt (engl. *application environment*) angesiedelt sind. Da diese Sicht in dem Modell der Ressourcenumwelt von den Discovery- und Verzeichnisdiensten erbracht wird, ist sie folgedessen in die Schicht des MAS-Rahmenwerkes verlagert, da sie von allgemeinem Nutzen für jedwede Art von Agentenanwendung sein kann.
- Zwischen der Anwendungsschicht und dem MAS-Rahmenwerk wurde eine weitere Ebene eingeführt, welche Unterstützung für weitergehende Konzepte z.B. aus der sozialen, der logischen und der Anwendungsumwelt bietet. Zu diesen Konzepten gehören beispielsweise *Gruppen* und *Rollen* (vgl. Abschnitt 2.2.2), sog. *Plätze* und *Regionen* (siehe [Weyns u. a. 2004]), *Koordinations-* und *Organisationsartefakte* (vgl. Abschnitt 2.2.3) sowie beliebige Anwendungsdaten (z.B. verfügbare Dokumente). Die Möglichkeit, diese Konzepte in das Modell zu integrieren, sollen an dieser Stelle lediglich erwähnt werden, da eine weitergehende Bearbeitung den Rahmen dieser Arbeit übersteigen würde.

Zu der Entwicklung eines Umweltmodells gehört nach [Zambonelli u. a. 2003] das Ermitteln aller in diesem Modell vorkommenden Entitäten und Ressourcen, die in einem Multiagentensystem ausgeführt, kontrolliert oder konsumiert werden können. Bei der Betrachtung von Abbildung 4.3 lassen sich folgende Entitäten ausmachen:

- die Anwendungsagenten und von diesen angebotene Dienste,
- das MAS-Rahmenwerk, eventuelle Erweiterungen des Rahmenwerkes, sowie die vom Rahmenwerk bereitgestellten Verzeichnis- und Discoverydienste,
- die Middleware bzw. die virtuelle Maschine, in der das Multiagentensystem läuft
- das Betriebssystem,
- die Geräte-Hardware, auf die von den höheren Schichten zugegriffen wird,
- logische Verbindungen, über die weitere Geräte kontaktiert werden können
- und Informationen über den Benutzer des Gerätes, zum Beispiel dessen Name, lokale Rechte, kryptografische Schlüssel zur Authentisierung etc. (nicht in der Abbildung dargestellt).

Zu jeder der aufgeführten Entitäten gehört darüber hinaus noch eine Reihe von Attributen, die die genauen Eigenschaften einer Entität beschreiben. Zum Beispiel wird ein angebotener Dienst nicht allein durch seinen Namen beschrieben, sondern zusätzlich über weitere Merkmale wie unterstützte Protokolle, Sprachen und Ontologien. Auch diese Details werden als Teil des Umweltmodells den Repräsentationen der Entitäten beigefügt.

Nach [Russell und Norvig 2003] lassen sich die Zustände dieses Modells einer Ressourcenumwelt folgendermaßen klassifizieren (vgl. Abschnitt 2.2.1):

- *teilweise observierbar*, da nicht jeder Agent die Umwelt samt aller konstituierender Entitäten wahrnehmen kann und soll,
- *stochastisch*, da in der Umwelt zufällige Ereignisse auftreten können, die nicht durch eine Aktion eines Agenten ausgelöst werden,
- *sequentiell*, da ein Umweltzustand unmittelbar von dem vorhergehenden Umweltzustand abhängt,
- *dynamisch*, da sich die Umwelt durch innere und äußere Einflüsse jederzeit und unvorhersehbar ändern kann,
- *nahezu kontinuierlich*, da die Zahl der möglichen Zustände in dieser Umwelt zwar begrenzt, aber extrem hoch sein kein.

Das in diesem Abschnitt entwickelte Modell einer ressourcenbasierten Umwelt wird in Abschnitt 5.2.2 wieder aufgegriffen. In jenem Abschnitt geht es um die Modellierung und Implementierung einer Komponente für ein Multiagentensystem, welche der Ausführungsumgebung eine Repräsentation der Ressourcenumwelt zur Verfügung stellt.

4.4.2 Identifikation von Umweltereignissen

In Abschnitt 2.2.4 wurden bereits drei grundlegende Klassen von Ereignissen entsprechend ihren Urhebern identifiziert (vgl. auch Abbildung 2.4). In die erste Klasse fallen Ereignisse, die von dem autonomen Prozess der Umwelt ausgelöst werden. Dieser autonome Prozess repräsentiert einen externen Eingriff in die Umwelt, zum Beispiel durch einen Menschen, der eine neue Ausführungsumgebung startet, oder durch eine Wetteränderung, welche sich auf die Qualität von Funkverbindungen auswirkt. Zu der zweiten Klasse von Ereignissen gehören solche, die durch eine Aktion einer Entität eine interne Zustandsänderung in einer anderen Entität auslösen. Solche Ereignisse treten unter anderem bei der Kommunikation zweier Entitäten auf oder bei dem Beenden einer Ausführungsumgebung. Die dritte Klasse umfasst schließlich alle Ereignisse, die nicht durch externe Ursachen begründet sind, sondern lediglich aus einer internen Aktion einer Entität resultieren. Beispiele hierfür sind das Fertigstellen einer Berechnung oder die Bearbeitung einer neuen Aufgabe aufgrund interner Zustandsänderungen.

Im Folgenden sollen lediglich die erste und zweite Klasse von Ereignissen, nicht jedoch Ereignisse, die aus einer internen Aktion einer Entität resultieren, berücksichtigt werden. Die Behandlung der Ereignisse dritter Klasse obliegt dem Programmierer des Agenten und muss nicht zwangsläufig der Umwelt mitgeteilt werden. Die Auflistung erhebt keinen Anspruch auf Vollständigkeit, eine Reihe weiterer Ereignisse sind je nach Kontext vorstellbar und sinnvoll, daher ist die Auswahl auf Ereignisse von allgemeiner Bedeutung beschränkt.

Beenden der Ausführungsumgebung Beim Beenden einer Ausführungsumgebung sollte allen laufenden Agenten die Möglichkeit gegeben werden, eine letzte Aktion auszuführen, bevor auch sie beendet werden. Eine letzte Aktion könnte zum Beispiel die Migration in eine andere erreichbare Ausführungsumgebung sein oder das Speichern der aktuellen Ausführung über einen Persistenzmechanismus für eine spätere Wiederaufnahme der Arbeit. Das Herunterfahren einer Ausführungsumgebung kann zudem von einem Agenten als Indiz dafür

angesehen werden, dass sein Benutzer sich entfernen möchte und der Agent Vorkehrungen treffen muss, ihm zu folgen. Dieses Ereignis sollte jedoch nicht nur lokal, sondern auch entfernt an alle für dieses Ereignis registrierten Agenten und Ausführungsumgebungen kommuniziert werden können, da es zum Beispiel durchaus möglich sein kann, dass entfernte Agenten Daten hinterlegt haben, die sie vor Beenden der Plattform in ihre derzeitige Ausführungsumgebung transferieren möchten.

Auslastung der Ausführungsumgebung Eine Ausführungsumgebung hat immer nur begrenzte Hardware-Kapazitäten zur Verfügung, die sie ihrerseits für die Ausführung von Agenten verwenden und verwalten muss. Zu diesen Kapazitäten gehören unter anderem der Prozessor, der Arbeitsspeicher, der Persistenzspeicher, Netzwerkschnittstellen und Energie. Sobald die Auslastung dieser Ressourcen einen von den Agenten vorgegebenen Schwellenwert erreicht, sollten die entsprechenden, lokal und entfernt für dieses Ereignis registrierten, Agenten benachrichtigt werden. Als Konsequenz könnte ein Agent von sich aus entscheiden, in eine andere, weniger belastete Umgebung zu migrieren oder sich zur Verteilung der Last auf entfernten Systemen zu duplizieren. Auch die Ausführungsumgebung und der Benutzer selbst sollten die Möglichkeit haben, bei zu hoher Belastung Agenten zwangsweise zu migrieren, wobei den Agenten fairerweise eine Art Vetorecht oder ein zeitlicher Aufschub eingeräumt werden könnte.

Ressourcen in der Umwelt Sobald in der Umwelt einer Ausführungsumgebung neue Ressourcen auftauchen oder bekannte Ressourcen verschwinden, sollte dieses Ereignis interessierten Agenten gemeldet werden. Von besonderem Interesse sind hierbei Ressourcen in Form von Ausführungsumgebungen, Diensten und anderen Agenten, aber auch zum Beispiel Netzwerkverbindungen zu entfernten Geräten. Beispielsweise könnte sich ein Agent informieren lassen, sobald eine neue Ausführungsumgebung erreichbar ist und auf diese zwecks besserer Lastverteilung migrieren. Oder ein Agent lässt sich benachrichtigen, sobald auf einer entfernten Plattform ein bestimmter Agent nicht mehr läuft, und veranlasst, dass ein neuer Agent gleichen Typs auf diese Plattform kopiert wird, um ein kontinuierliches Dienstangebot zu gewährleisten.

Änderung der Verbindungsqualitäten Die Verbindungsqualität (auch als Dienstgüte bezeichnet) zweier Geräte untereinander setzt sich aus verschiedenen Parametern zusammen. Zu diesen Parametern gehören unter anderem die Latenzzeit, die Abweichung der Latenzzeit von ihrem Mittelwert, die Paketverlustrate und der Datendurchsatz [Wikipedia 2007b]. Eine Verringerung der Verbindungsqualität zwischen zwei Geräten kann unter anderem durch eine erhöhte Auslastung einer der beiden Endpunkte entstehen oder durch eine Verminderung der Signalqualität, was bei Funkverbindungen unter anderem auf eine zunehmende Distanz zwischen Sender und Empfänger zurückgeführt werden kann. Letzteres kann ein Agent zum Beispiel als Indiz dafür deuten, dass sich sein Benutzer von der aktuellen Ausführungsumgebung entfernt und dass der Agent Maßnahmen ergreifen muss, um diesem zu folgen.

Positionsänderung Die Positionsänderung betrifft ausschließlich Ausführungsumgebungen auf mobilen Geräten. Es gibt verschiedene Möglichkeiten eine solche Änderung festzustellen, zum Beispiel: 1) das mobile Gerät verfügt über einen *GPS*[7]-Empfänger und kann daher

[7] GPS (*Global Positioning System*) ist ein satellitengestütztes System zur Positionsbestimmung auf der Erde

seine Position absolut bestimmen, 2) das mobile Gerät bestimmt eine Änderung der Position relativ zu ein oder mehreren anderen Ausführungsumgebungen anhand veränderter Nachrichtenlaufzeiten oder der Änderung der Signalstärken. Abhängig von einer bestimmten Position könnte ein Agent zum Beispiel die Empfängerplattform einer bevorstehenden Migration oder Kommunikation auswählen (vgl. Anwendungsbeispiel Kaufhausagent, Abschnitt 4.2.4)

Zeitänderung Neben der Positionsänderung könnte auch das Fortschreiten der Zeit als Ereignis für einen Agenten interessant sein. So wäre es zum Beispiel vorstellbar, dass eine Ausführungsumgebung einem Agenten nur einen begrenzten Zeitrahmen für die Ausführung zur Verfügung stellt und ihn nach Ablauf der Zeit wieder zurück in seine ursprüngliche Ausführungsumgebung transferiert.

Migration anderer Agenten Eine weiteres Ereignis, welches für einen Agenten von Interesse sein könnte, ist die Migration eines anderen lokal ausgeführten Agenten in eine entfernte Ausführungsumgebung. Auf diese Weise ließen sich Abhängigkeiten zwischen Agenten ausdrücken und sicherstellen, dass Agenten, die koordiniert Aufgaben in einer Ausführungsumgebung bearbeiten, auch immer in dieselbe Ausführungsumgebung migrieren.

Benutzerinduzierte Ereignisse Der Benutzer soll in jedem Fall die Kontrolle über seine Agenten und seine Ausführungsumgebung behalten können. Aus diesem Grund muss er die Möglichkeit haben, einerseits eigene entfernt laufende Agenten in seine Ausführungsumgebung migrieren zu lassen und andererseits fremde Agenten, die in seiner Ausführungsumgebung laufen, manuell in andere Umgebungen zu verschieben. Vorstellbar wäre auch, dass ein Benutzer die Rechte fremder Agenten zur Laufzeit ändert, zum Beispiel die Prozesspriorität herabsetzt oder die Nutzung des Persistenzspeichers verbietet. Auch solche Ereignisse müssen den Agenten mitgeteilt werden, damit diese entsprechend darauf reagieren können.

4.5 Resource Awareness-Komponente

Die Resource Awareness-Komponente (RA-Komponente) ist für die Wahrnehmung der Umwelt, welches eine wesentliche Voraussetzung für die kontextabhängige Migration darstellt, zuständig. Ihre Hauptaufgabe besteht in der Erstellung und kontinuierlichen Aktualisierung des Umweltmodells. Das bedeutet, sie ist dafür verantwortlich, die interessanten Entitäten und Ereignisse in der Umwelt wahrzunehmen, in eine passende Repräsentation zu überführen und schließlich in das Umweltmodell zu integrieren. Hierfür kann sich die Komponente verschiedenster Discovery-Verfahren und Repräsentationssprachen, die über Adapter an die Schnittstellen der Komponente angepasst werden müssen, bedienen. Um dies zu gewährleisten, müssen die internen Strukturen und Mechanismen der Komponente einerseits flexibel und andererseits generisch gestaltet werden. Der Einsatz externer Verfahren und Repräsentationsformen soll dann abhängig von den Gegebenheiten der Umwelt dynamisch ausgewählt werden können. Um die Verständigung zwischen verschiedenen RA-Komponenten trotz der möglichen Vielfalt unterstützter Verfahren und Repräsentationsformen sicherzustellen, muss zudem ein proprietäres Basisprotokoll entworfen werden, auf welches die RA-Komponenten zurückgreifen können, falls keine gemeinsamen Protokolle vorhanden sind. Dieses proprietäre Protokoll soll darüber hinaus möglichst simpel und effizient gehalten werden, damit es auch auf mobilen Geräten eingesetzt werden kann.

Im Folgenden sollen die wichtigsten Aspekte und Aufgaben der RA-Komponente betrachtet werden. Hierzu gehören die Organisation des Umweltmodells und die Unterstützung verschiedener Erstkontaktmechanismen sowie das Empfangen und die Verbreitung von Umweltinformationen. Damit einher geht auch die Fähigkeit zur Verarbeitung verschiedener Repräsentationssprachen. Außerdem wird das erwähnte Basisprotokoll vorgestellt, welches den Forderungen nach sparsamem Umgang mit Ressourcen und hoher Konfigurierbarkeit gerecht wird.

4.5.1 Organisation des Umweltmodells

Das Umweltmodell stellt die Grundlage der Resource Awareness dar. In diesem Modell werden alle verfügbaren Informationen über Entitäten und Ereignisse (vgl. Abschnitt 4.4) in der Umwelt vorgehalten, kontinuierlich aktualisiert und bei Bedarf etwaigen Interessenten zur Verfügung gestellt. Im vorangegangenen Abschnitt wurden bereits die Entitäten, die in einer Ressourcenumwelt enthalten sind und über die folglich Informationen angeboten und nachgefragt werden können, identifiziert. Doch ist es einerseits nicht in allen Situationen erforderlich, alle verfügbaren Informationen auszutauschen, und andererseits auch hinsichtlich des Datenaufkommens und des Schutzes privater Daten nicht unbedingt wünschenswert. Daher werden besondere Anforderungen an die Organisation dieser gemeinsamen Zugriffsstruktur gestellt. Ein Benutzer soll individuell entscheiden können, für welche Art von Ressourcen er sich in der Umwelt interessiert und welche Informationen über ihn, seine Hard- und Softwareausstattung, auf seinem Gerät angebotene Dienste, laufende Agenten etc. öffentlich zur Verfügung gestellt werden sollen. Mit diesen Präferenzen kann eine Ausführungsumgebung gezielt Anfragen an entfernte Ausführungsumgebungen stellen, welche daraufhin ausschließlich mit den angeforderten Informationen antworten und so insgesamt weniger Daten verschicken müssen. Des Weiteren soll das Umweltmodell flexibel erweiterbar sein, sodass neben den grundlegenden Informationen zusätzlich anwendungsspezifische Informationen integriert werden können.

Für die Realisierung solch einer gemeinsamen Zugriffsstruktur bieten sich verschiedene Möglichkeiten an. Das *ProNav*-Projekt, vorgestellt in Abschnitt 4.8.2, verwendet sogenannte dynamische Karten für die Repräsentation der Umwelt, welche allerdings einen wenig intuitiven Umgang seitens des Benutzers erlauben. Das *Konark* Discovery-Protokoll repräsentiert die Umwelt in Form einer hierarchischen Baumstruktur, deren Knoten abstrakte Diensttypen und deren Blätter konkrete Dienstbeschreibungen darstellen (siehe Abschnitt 2.4.3 sowie Abbildung 2.12). Vorstellbar wäre auch die Integration von Umweltinformationen in bereits bestehende Zugriffsstrukturen einer Ausführungsumgebung. So könnten beispielsweise die gelben Seiten (auch als *Directory Facilitator* bezeichnet) und weißen Seiten (auch *Agent Management System* genannt) einer Agentenplattform, Informationen über entfernt angebotene Dienste und entfernt laufende Agenten aufnehmen[8]. In dieser Arbeit soll das Umweltmodell ähnlich der Konark-Baumstruktur organisiert und hierfür das Konzept von Nachrichtenkanälen (engl. *news channel*) verwendet werden. Dies erlaubt eine einfache thematische Strukturierung der Informationen und Abonnenten eines Kanals können Aktualisierungen von Teilen des Umweltmodells in chronologischer Reihenfolge beziehen. Zudem findet die Metapher solcher Nachrichtenkanäle im Alltag vielfach Verwendung[9], sodass Benutzer mit dem Modell intuitiv umgehen können.

[8] Ein Beispiel hierfür findet sich unter anderem in der *JADE*-Agentenplattform, welches die Föderation verteilter DFs unterstützt und somit den Agenten einen transparenten Zugriff auf Informationen über entfernt angebotene Dienste erlaubt [Bellifemine u. a. 2003]

[9] Man denke an Zeitschriften, Fernsehsender, SMS-Abonnements für Telefone, RSS-Feeds im Internet etc., die sich oftmals auf ein oder wenige Themen (Sport, Nachrichten, Kultur) beschränken.

Nachrichtenkanäle

Bereits in den Neunzigern hat die Firma Microsoft mit den *Active Channels* [Microsoft 1997] und die Firma Netscape mit dem *Netcaster* [Netscape 1997] ein den Nachrichtenkanälen ähnliches internetbasiertes Distributionskonzept für Sparteninformationen vorgestellt. Auch bei diesen beiden Systemen konnten Benutzer Nachrichtenkanäle abonnieren und bekamen - sobald verfügbar - neue Informationen mittels einer *push*-Strategie zugesandt (siehe hierzu auch *Push-Medien* [Wikipedia 2006e]). Der Spezifikation des Umweltmodells aus Abschnitt 4.4 folgend, sollen von der RA-Komponente eine Auswahl von fünf Standard-Nachrichtenkanälen unterstützt werden (siehe Abbildung 4.4).

Abbildung 4.4: Standard-Nachrichtenkanäle der RA-Komponente (eigene Darstellung)

Heartbeat-Kanal Ein Heartbeat (dt. *Herzschlag*) wird in regelmäßigen Intervallen an die Abonnenten des Kanals geschickt, um anzuzeigen, dass die lokale RA-Komponente bzw. die Ausführungsumgebung noch betriebsbereit ist. Anstatt wie viele Discovery-Verfahren Lease-Mechanismen für alle Arten von Umweltinformationen einzuführen (vgl. Abschnitte 2.3 und 2.4), unterstützt die RA-Komponente eine Kombination aus Nachrichtenabonnement und Herzschlag. Abonniert eine RA-Komponente zum Beispiel den Services-Kanal einer anderen RA-Komponente, kann sie davon ausgehen, dass ein Dienst solange verfügbar ist, bis entweder kein Herzschlag mehr empfangen wird oder der Dienstabbruch als Nachricht über den Services-Kanal empfangen wird. Auf diese Art und Weise ist es nicht weiter notwendig aufgrund von Lease-Zeiten alle Kanalinformationen und -abonnements in regelmäßigen Abständen aufzufrischen, was die Gesamtheit der im Netzwerk zu übertragenen Datenmenge reduziert. Um die Datenmenge weiter zu reduzieren, wird außerdem zwischen impliziten und expliziten Herzschlägen unterschieden. Ein impliziter Herzschlag ist jede Art von empfangener Nachricht. Wird ein solcher impliziter Herzschlag versendet bzw. empfangen, setzen die beteiligten RA-Komponenten das Intervall für den nächsten zu sendenden expliziten Herzschlag entsprechend weiter in die Zukunft.

System-Kanal Der System-Kanal versendet Informationen über das lokale Gerät. Zu diesen Informationen gehören Angaben zum Gerät selbst (Name, Geräteart, Standort, Beschreibung), dessen Hardware-Ausstattung (Prozessor, Speicher, Bildschirm, Netzwerkschnittstellen) und der Auslastung der einzelnen Komponenten (Speicherverbrauch, Prozessorlast etc.), die Software-Ausstattung (Betriebssystem, Laufzeitumgebung, Agentenplattform) sowie Angaben über die lokale RA-Komponente (Kontaktdaten, angebotene Kanäle, unterstützte Repräsentationsformen).

Services-Kanal Über den Services-Kanal lässt sich das Angebot an Diensten einer Ausführungsumgebung verfolgen. Der Services-Kanal erhält seine Informationen direkt von dem lokalen *Di-*

rectory Facilitator und versendet Zusammenfassungen der Dienstbeschreibungen (Name, unterstütze Protokolle, Sprachen, Ontologien) an interessierte Abonnenten.

Agents-Kanal Der Agents-Kanal versendet Informationen über die aktuell laufenden bzw. gestoppten Agenten einer Ausführungsumgebung. Unter Zuhilfenahme dieses Nachrichtenkanals lässt sich ein einfacher Monitoring-Mechanismus aufbauen, welcher beispielsweise bei den selbstorganisierenden Dienstnetzwerken aus Anwendungsbeispiel 2 (vgl. Abschnitt 4.2) Verwendung finden könnte.

Contacts-Kanal Über den Contacts-Kanal lassen sich alle bekannten Kontakte einer RA-Komponente abfragen. Als Ergebnis erhält der Anfragende die Kontaktdaten aller bekannten RA-Komponenten. Aus Gründen des Datenschutzes ist es explizit nicht vorgesehen, weiterführende Informationen über Kontakte (zum Beispiel die Inhalte von Nachrichten-Kanälen) weiterzugeben. Hierzu müssen die Kontakte von dem Anfragenden selbst kontaktiert werden.

Misc-Kanäle Die bisher beschriebenen Kanäle sind die Standard-Nachrichtenkanäle, die jede RA-Komponente ohne zusätzlichen Programmieraufwand anbieten kann. Es gibt allerdings die Möglichkeit, dem Angebot weitere Kanäle hinzuzufügen, um andere Arten von Information öffentlich anzubieten. Vorstellbar wären z.B. eigene Kanäle für Sensordaten, Programmaktualisierungen, Sportnachrichten, Mediendateien, Dokumente etc.

Abonnement-Anfragen

Um einen Nachrichtenkanal zu abonnieren, muss eine RA-Komponente eine Anfrage an die anbietende RA-Komponente verschicken. Diese Abonnementanfrage kann eine repräsentationsabhängige Suchanfrage (engl. *query*) enthalten, mittels derer die Kanalnachrichten schon vor dem Versand gefiltert werden können. Zum Beispiel könnte eine Abonnement-Anfrage für den Agents-Kanal eine Suchanfrage enthalten, die auf alle Agenten passt, deren Namen das Präfix *„XYZ“* aufweist. In diesem Fall würden alle Informationen über nicht derartige benannte Agenten aus den Kanalnachrichten gefiltert, und so die zu versendende Nachrichtenmenge reduziert werden können.

Außerdem kann eine Abonnement-Anfrage ein oder mehrere Beschränkungen (engl. *constraints*) enthalten, mittels derer sich zum Beispiel die Aktualisierungsintervalle beim Nachrichtenversand seitens des Abonnenten festlegen lassen. So kann zum Beispiel geregelt werden, dass für hundert gleichzeitig gestartete Agenten nicht hundert einzelne Nachrichten verschickt werden, sondern dass nach Eintreten eines Ereignisses, eine bestimmte Zeitspanne weitere Ereignisse gesammelt werden, bevor eine Nachricht verschickt wird.

Jede Abonnementanfrage sollte von dem Empfänger bestätigt werden, unabhängig davon, ob die Anfrage akzeptiert wurde oder nicht. In dieser Bestätigung (engl. *acknowledgement*) kann der Anbieter auch seinerseits Beschränkungen unterbringen, die zum Beispiel das Aktualisierungsintervall heraufsetzt, um das Datenübertragungsvolumen zu reduzieren.

4.5.2 Erstkontaktmechanismen

Wenn eine Ausführungsumgebung neu gestartet wird oder sich zum Beispiel auf einem mobilen Gerät in Reichweite eines neuen Netzwerkes begibt, muss die lokale RA-Komponente andere entfernte RA-Komponenten finden, mit denen sie Informationen über die Umwelt austauschen

und sich selbst bekannt machen kann. Um diese initialen Kontakte zu finden, gibt es eine Reihe verschiedener Möglichkeiten, die sich in aktive und passive Mechanismen unterteilen lassen. Die Verwendung aktiver Mechanismen erfordert den proaktiven Versand von Kontaktnachrichten, während eine RA-Komponente bei dem Einsatz passiver Mechanismen auf das Eintreffen bestimmter Nachrichten anderer RA-Komponenten wartet, was anfänglich keine Übertragung von Daten erfordert.

Ein weit verbreiteter Mechanismus zum aktiven und passiven Auffinden von Kontakten ist der Einsatz von Broadcast- bzw. Multicast-Nachrichten[10] (vgl. Abschnitte 2.3 und 2.4). Multicast-Nachrichten können in lokalen Netzwerken von einem Sender an mehrere Empfänger gleichzeitig verschickt werden, ohne dass der Sender die Adressen der Empfänger kennen muss. Hierzu adressiert der Sender seine Nachricht an eine fest vorgegebene Multicast-Adresse, welche gleichzeitig eine sogenannte Multicast-Gruppe repräsentiert. Interessenten können ihrerseits einer Multicast-Gruppe beitreten und bekommen fortan alle an diese Gruppe adressierten Nachrichten zugestellt. Um neue Kontakte zu finden, kann eine RA-Komponente ihre eigenen Kontaktdaten in einer Multicast-Nachricht verpacken und im Netzwerk versenden. Mitglieder der Multicast-Gruppe empfangen diese Nachricht und können von sich aus entscheiden, ob sie dem Sender wiederum mit ihren Kontaktdaten antworten möchten. Es ist darüber hinaus auch möglich, gleichzeitig in mehreren Multicast-Gruppen eingeschrieben zu sein, sodass beispielsweise verschiedene Discovery-Verfahren gleichzeitig unterstützt werden können.

Da zum einen die meisten Routing-Protokolle für ad-hoc Netzwerke den Versand von Multicast-Nachrichten nicht unterstützen, und zum anderen die Reichweite derartiger Nachrichten in Infrastrukturnetzwerken lokal begrenzt ist, müssen zusätzlich weitere Erstkontaktmechanismen unterstützt werden. Hierzu gehört auch die Herstellung eines Erstkontaktes mittels Unicast-Nachrichten[11]. Hierfür ist es jedoch notwendig, dass der Sender einer Kontaktnachricht die Adresse des Empfängers bereits im Vorwege kennt. Dafür funktioniert der Nachrichtenversand sowohl in ad-hoc Netzwerken als auch in Weitverkehrsnetzwerken.

Neben den allgemeinen Multicast- und Unicast-Mechanismen, gibt es auf technischer Seite noch weitere verfahrensabhängige Weisen, initiale Kontakte zu finden. In Abschnitt 2.4.2 wurde beispielsweise das *Bluetooth Service Discovery Protocol* beschrieben, welches speziell für den Kurzstreckenfunk Bluetooth entworfen wurde. Hierbei finden sich Geräte in Funkreichweite durch Abtasten des Bluetooth-eigenen Funkspektrums und können bei erfolgreicher Suche schließlich eine Kommunikationsbeziehung aufbauen.

Wurde unter Verwendung einer der oben aufgeführten Kontaktmechanismen ein erster Kontakt gefunden, so kann dieser nach ihm bekannten weiteren Kontakten befragt werden. Hierfür kann eine RA-Komponente einfach den *Contacts*-Nachrichtenkanal des gefundenen Kontaktes abonnieren und bekommt hierdurch Zugriff auf die Adressdaten aller diesem Kontakt bekannten weiteren Kontakte, an welche im Anschluss einzeln Unicast-Nachrichten verschickt werden können.

[10] Im Gegensatz zu einer Broadcast-Nachricht wird eine Multicast-Nachricht von einem Gerät nur dann empfangen, wenn es explizit durch Beitritt einer Multicast-Gruppe Interesse bekundet hat. Somit ist der Einsatz von Multicast gegenüber Broadcast vorzuziehen, um einerseits die Ressourcen der Geräte zu schonen, und andererseits verschiedene Discovery-Verfahren gleichzeitig einsetzen zu können.

[11] Unicast-Nachrichten werden von einem Sender an genau einen Empfänger adressiert.

4.5.3 Verbreitung von Umweltinformationen

Informationen in einer natürlichen Umwelt verbreiten sich, ohne dass Entitäten selbst aktiv werden müssen, zum Beispiel Licht und Schall (vgl. Abschnitt 2.2.4). In einer logischen Umwelt, wie der Ressourcenumwelt, müssen die Informationen jedoch aktiv unter den einzelnen Entitäten ausgetauscht werden, damit diese Kenntnis über Entitäten und Ereignisse in ihrer Umgebung erlangen. Hierzu wurden in den Abschnitten 2.3 und 2.4 eine Reihe sogenannter *Service Discovery*-Verfahren vorgestellt, wobei sich der Begriff der Service Discovery nicht zwangsläufig auf das Auffinden angebotener Dienste beschränkt, sondern bei den meisten Verfahren Informationen aller Art beinhaltet.

Aus der Anforderungsanalyse (vgl. Abschnitt 4.3) geht der Anspruch hervor, möglichst viele Verfahren zu unterstützen, das bedeutet, prinzipiell kein Verfahren grundlegend auszuschließen und stattdessen generische Schnittstellen zu bieten, sodass Adapter für beliebige Verfahren erstellt werden können. Der Grund für die Forderung einer breiten Unterstützung liegt in der Heterogenität und Dynamik der Umwelt, und da es kein Verfahren gibt, welches für jedweden Einsatzzweck gut geeignet ist (vgl. Abschnitte 2.3.7 und 2.4.7), werden in der Praxis eine Vielzahl unterschiedlicher Verfahren verwendet.

Daher kann eine Ausführungsumgebung idealerweise über entsprechende Adapter auf eine Vielzahl verschiedener Verfahren zurückgreifen, deren Einsatz sich nach den in einer Umgebung vorgefundenen Gegebenheiten richtet. Wenn zum Beispiel eine Ausführungsumgebung auf einem mobilen Gerät einem neuen, fremden Netzwerk beitritt, könnte es zur Herstellung eines Erstkontaktes alle bekannten verfahrenstypischen Kontaktmechanismen ausprobieren, um Informationen über andere Entitäten zu erhalten und sich selbst der neuen Umwelt bekannt zu machen. Sobald Antworten auf die initialen Kontaktanfragen eintreffen, kann die Ausführungsumgebung schließlich autonom eine Auswahl unterstützer Verfahren treffen, über die fortwährend der weitere Austausch von Informationen abgewickelt wird.

Die Wahl eines geeigneten Verfahrens hängt hierbei von einer Reihe unterschiedlicher Kriterien ab (vgl. Abschnitte 2.3.7 und 2.4.7), zum Beispiel die mögliche Reichweite einer Kommunikation, die Skalierbarkeit eines Verfahren oder der Protokollüberhang (engl. *protocol overhead*), den ein Verfahren mit sich bringt (beispielsweise zum Aufbau und Erhalt logischer Netzwerke). Da die autonome Entscheidung der Ausführungsumgebung für oder gegen ein Verfahren in einem bestimmten Kontext schwierig zu treffen ist, soll der Benutzer entsprechende Vorgaben machen können. So ließe sich der Ausführungsumgebung zum Beispiel vorschreiben, niemals ein datenintensives Protokoll zu verwenden, sobald das Gerät über eine teure UMTS-Verbindung einem Netzwerk beigetreten ist, oder ein Protokoll mit großer Reichweite (z.B. *JXTA*) zu verwenden, sobald ein Internetzugang verfügbar ist.

Doch nicht nur die Wahl eines Verfahrens soll abhängig von den Umweltbedingungen geschehen, es muss auch möglich sein, zur Laufzeit bestimmte Parameter, die die Ausführung eines Verfahrens beeinflussen, zu ändern. Beispielsweise könnte eine Ausführungsumgebung autonom entscheiden, dass bei Verringerung der Verbindungsqualität bestimmte Informationen nicht mehr an andere Ausführungsumgebungen übermittelt werden und die Häufigkeit der Aktualisierung reduziert wird.

Damit trotz der Unterstützung verschiedener Verfahren, die Umweltinformationen einheitlich zugreifbar sind, müssen alle Informationen, die von den Verfahren eingeholt oder versandt werden, in einer gemeinsamen Zugriffsstruktur, hier auch als *Umweltmodell* bezeichnet, vereinigt werden. Diese Zugriffsstruktur muss also Schnittstellen bieten, über die ein Verfahren sowohl Infor-

mationen abfragen als auch neue Informationen hinzufügen bzw. Informationen ändern kann. Da normalerweise jedes Verfahren seine eigenen Datenmodelle und Zugriffsstrukturen mit sich bringt, muss auch für jedes Verfahren ein entsprechender Adapter geschrieben werden, der zwischen der gemeinsamen Zugriffsstruktur und den individuellen Lösungen übersetzen kann.

4.5.4 Repräsentationssprachen

Mit der möglichen Erweiterbarkeit durch Discovery-Verfahren, geht die Forderung der Unterstützung unterschiedlicher Informationsrepräsentationen einher. Manche Discovery-Verfahren tauschen Informationen in proprietären Formaten aus, andere verwenden einen der vielen Repräsentationsstandards (vgl. Abschnitt 2.5). Wesentlich ist also, dass auch die Ausführungsumgebung mit allen Formaten umgehen kann, in denen Informationen durch die von ihr unterstützten Discovery-Verfahren repräsentiert werden.

Wenn eine Ausführungsumgebung über ein bestimmtes Discovery-Verfahren Informationen von einer anderen, entfernten Ausführungsumgebung empfängt, müssen diese Informationen in das lokale Umweltmodell, also die bereits erwähnte gemeinsame Zugriffsstruktur, integriert werden. Diese Aufgabe könnte von den jeweiligen Adaptern der Discovery-Verfahren übernommen werden, da diese ohnehin für jedes unterstützte Verfahren programmiert werden müssen. Eine andere, allgemeinere Lösung wäre die Erstellung eigener Adapter für jede unterstützte Repräsentationsform. Der Vorteil dieses Ansatzes besteht einerseits in der Wiederverwendbarkeit eines Adapters, falls zwei Discovery-Verfahren dieselbe Repräsentationsform gebrauchen, und andererseits in der Möglichkeit, Informationen des Umweltmodells auch anderweitig verwenden zu können, wenn sie in einem Format repräsentiert werden, welches nicht von den unterstützten Verfahren verwendet wird. So wäre es zum Beispiel möglich, die im Umweltmodell enthaltenen Dienstbeschreibungen mittels WSDL (siehe Abschnitt 2.5.4) zu repräsentieren und anderweitig zu verwenden, auch wenn kein Discovery-Verfahren unterstützt wird, welches WSDL-Repräsentationen verwendet. Ein solcher Repräsentationsadapter übernimmt also die Aufgabe, Informationen von einem bestimmten Format in das generische Format des Umweltmodells zu übersetzen und vice versa.

Da für jede Art von Repräsentationsform unterschiedliche Abfragesprachen existieren, müssen darüber hinaus auch Möglichkeiten geschaffen werden, die Abfragesprachen zu unterstützen. Eine Abfragesprache hat den Zweck bestimmte Informationen aus einer größeren Menge an Informationen zu extrahieren. Wenn sich zum Beispiel eine Ausführungsumgebung lediglich für angebotene Dienste in ihrer Umgebung interessiert, kann sie ihr spezifisches Interesse mittels einer Abfragesprache formulieren und an entfernte Ausführungsumgebungen übermitteln. Diese Ausführungsumgebungen wiederum setzen den entsprechenden Repräsentationsadapter ein, um die Informationen des Umweltmodells in das geforderte Format zu bringen, und wenden anschließend die Abfrage an, um die Ergebnisse zu filtern. Diese Ergebnismenge können sie dann an die anfragende Ausführungsumgebung als Antwort übermitteln.

4.5.5 Proprietäres Basisprotokoll

Um sicherzustellen, dass sich alle RA-Komponenten untereinander verständigen können, auch wenn sie eigentlich gänzlich unterschiedliche Discovery-Verfahren und Repräsentationssprachen zum Wahrnehmen und Beschreiben ihrer Umwelt verwenden, müssen alle Komponenten zumindest ein gemeinsames Basisprotokoll implementieren.

In den Abschnitten 2.3.7 und 2.4.7 wurde bereits festgestellt, dass sich keines der in dieser Arbeit betrachteten Discovery-Verfahren sowohl für Infrastruktur- als auch für ad-hoc Netzwerke eignet. Das Ergebnis des Abschnittes 2.5 war, dass die dort vorgestellten Repräsentationssprachen für Ressourcen, für einen einfachen Einsatz insbesondere auf mobilen Geräten zu mächtig sind. Eine Schlussfolgerung aus diesen Umständen ergibt, dass ein einfaches und effizientes Protokoll entworfen werden muss, welches als kleinster gemeinsamer Nenner für alle RA-Komponenten in heterogenen Umgebungen fungieren kann und den wesentlichen, im Folgenden aufgeführten Anforderungen gerecht wird[12].

- Portierbarkeit in die Java-Laufzeitumgebungen mobiler Geräte, welche aufgrund der geringen verfügbaren Ressourcen diversen Beschränkungen unterliegen.
- Umgang sowohl mit Unicast- als auch mit Multicast-Nachrichten zum Auffinden von Kontakten in einem Netzwerk. Auf diese Weise können sowohl Infrastruktur- als auch ad-hoc Netzwerk unterstützt werden.
- Einfacher Versand von Ressourceninformationen als serialisierte Java-Objekte bzw. als String-Repräsentationen, damit sichergestellt werden kann, dass auch beschränkte Java-Laufzeitumgebungen auf mobilen Geräten die Informationen verarbeiten können.
- Einfache Abfragesprache für Ressourceninformationen, um eine individuelle Auswahl interessanter Informationen treffen zu können.
- Hohes Maß an Konfigurierbarkeit seitens des Benutzers, um das Protokoll an unterschiedliche Umweltbedingungen manuell anpassen zu können.

In den Forderungen bereits aufgeführt ist die Unterstützung einer einfachen Repräsentationsform und Abfragesprache für Informationen aus dem Umweltmodell. Da jedoch die existierenden Standards zu mächtig sind und oft nicht von den eingeschränkten Ausführungsumgebungen mobiler Geräte unterstützt werden, gilt es eine ganz einfache, proprietäre Repräsentationsform und Abfragesprache zu verwenden. Hierfür wird der Einsatz von Zeichenketten sowie eine der *Object Query Language*[13] oder der *JSP Expression Language*[14] ähnliche Abfragesprache vorgeschlagen.

Des Weiteren soll das Protokoll, sofern dies möglich ist, Nachrichten über Sockets versenden, anstatt die Transportmechanismen der Ausführungsumgebung zu nutzen. Hierdurch bietet sich die Möglichkeit mittels spezieller Software-Sensoren die Übertragungsrate sowie die Antwortzeiten bei einer Interaktion zwischen RA-Komponenten zu messen (vgl. *ProNav*-Projekt, Abschnitt 4.8.2), was bei Verwendung der üblichen Transportmechanismen einer Plattform nicht unbedingt gegeben ist. Die aus den Messungen gewonnen Daten können zum einen für die Erstellung von Migrationsstrategien verwendet werden (vgl. *Kalong*-Mobilitätsmodell, Abschnitt 3.3.2), zum anderen auch zur ungefähren Positionsbestimmung eines mobilen Gerätes relativ zu anderen Geräten (vgl. Anwendungsbeispiel Kaufhausagent, Abschnitt 4.2.4). Außerdem ist eine möglichst zeitnahe Reaktion auf empfangene Nachrichten für die Funktion des *Heartbeat*-Nachrichtenkanals erforderlich, da eine verzögerte Verarbeitung fälschlich als Ausbleiben des Herzschlags erkannt werden könnte.

[12] Nähere Details zu diesem Protokoll finden sich in Abschnitt 5.2.

[13] Die *Object Query Language* ist ein von der Object Database Management Group (ODMG) vorgeschlagener Standard einer an die *Structured Query Language* (SQL) angelehnten Abfragesprache für objektorientierte Datenbanken [Berler u. a. 2000]

[14] Die *JSP Expression Language* wurde von Sun Microsystems entworfen, um einen einfachen, textbasierten Zugriff auf Objekte und deren Attribute zu ermöglichen [Delisle u. a. 2006, Kapitel 2].

4.5.6 Sparsamer Umgang mit Daten

Je mehr Informationen ein Umweltmodell enthält, desto mehr Hardware-Ressourcen werden benötigt, um dieses zu verwalten und die Inhalte an interessierte RA-Komponenten zu kommunizieren. Es ist daher notwendig, nur solche Informationen in ein Umweltmodell aufzunehmen, die von dem Benutzer bzw. der Ausführungsumgebung gefordert und benötigt werden. Auch sollten Informationen, die als ungültig oder veraltet erkannt werden, wieder aus dem Modell entfernt werden, beispielsweise wenn eine RA-Komponente den erwarteten Herzschlag einer entfernten RA-Komponente nicht empfängt.

Neben der allgemeinen Empfehlung der Datensparsamkeit, sollte die RA-Komponente aber auch weitergehende Mechanismen zur Reduzierung des Datenvolumens bereitstellen. Hierzu gehört beispielsweise die bereits erwähnte Unterstützung von Abfragesprachen für das Umweltmodell. Der Benutzer aus dem Nachrichtenagent-Anwendungsbeispiel (siehe Abschnitt 4.2.5) könnte seiner Ausführungsumgebung auf dem mobilen Gerät zum Beispiel die Vorgabe machen, dass sie entfernte Ausführungsumgebungen ausschließlich auf einen vorhandenen Internetzugang hin befragen soll, alle anderen Informationen sind überflüssig.

Eine weitere Möglichkeit bieten sogenannte *Nachrichten-Pools*, in denen Nachrichten an einen bestimmten Adressaten zeitweilig vor dem Versand zwischengespeichert werden, um ggf. zu einem späteren Zeitpunkt mit anderen Nachrichten an denselben Interessenten gemeinsam versendet zu werden. Anstatt beispielsweise einhundert einzelne Nachrichten an Interessenten zu versenden, sobald in einer Ausführungsumgebung einhundert Agenten gestartet werden, könnten diese Nachrichten bzw. deren Inhalte zusammengefasst und als eine einzige Nachricht versendet werden. Auch sollte es möglich sein, den Versand von Nachrichten, deren Inhalt bereits nach einer kurzen Zeit an Gültigkeit verliert, zu unterbinden. Wird zum Beispiel ein Agent gestartet, der bereits nach kurzer Zeit wieder beendet wird, müssen diese beiden Ereignisse nicht zwangsläufig an alle interessierten RA-Komponenten übermittelt werden.

4.5.7 Konfigurierbarkeit

Die RA-Komponente sollte sich vom Benutzer weitgehend konfigurieren und damit an unterschiedliche Einsatzzwecke anpassen lassen. Dies ist insbesondere aufgrund der Heterogenität der Infrastruktur sowie der Geräte erforderlich, um vor allem den Ressourcenverbrauch der RA-Komponente zu reduzieren. Beispielsweise sollte eine RA-Komponente, die auf einem mobilen Gerät in einem ad-hoc Netzwerk ausgeführt wird, weniger Ressourcen verbrauchen als eine RA-Komponente, die auf einem Server in einem Infrastrukturnetzwerk läuft.

Um dies zu ermöglichen, muss der Benutzer das Umweltmodell, das proprietäre Basisprotokoll, und die unterstützten Discovery-Verfahren und Repräsentationssprachen sehr genau an seine Bedürfnisse anpassen können. Für das Umweltmodell hieße das zum Beispiel, dass der Benutzer nur bestimmte Nachrichtenkanäle aktiviert oder mittels Abfragesprachen seine spezifischen Interessen an Umweltmodellen anderer RA-Komponenten formulieren können soll. Für das Basisprotokoll könnten Minimumzeiten für das Nachrichten-Pooling vorgegeben werden, die Sensoren zur Messung der Verbindungsqualitäten deaktiviert oder eine Kompression der Daten vor der Übertragung gefordert werden. Die Konfigurationsoptionen für einzelne Discovery-Verfahren und Repräsentationssprachen hängen individuell von der jeweils eingesetzten Implementation ab, sollten aber von dem Benutzer ebenso angepasst werden können. Grundlegend ist auch, dass die Konfiguration zur Laufzeit, also ohne Neustart der Ausführungsumgebung, angewendet werden können soll, damit zeitnah auf Änderungen der Umwelt reagiert werden kann.

4.6 Spezifikation des Mobilitätsmodells

Mit Hilfe der in Abschnitt 3.3.3 vorgestellten Mobilitätssprache *MoL* lassen sich alle Aspekte eines Migrationsvorgangs formal beschreiben [Braun 2003, Braun und Rossak 2005]. Eine solche Beschreibung ist in verschiedener Hinsicht hilfreich. Zum einen lassen sich hierdurch die Migrationsmechanismen unterschiedlicher Systeme miteinander vergleichen, zum anderen lassen sich mit der Sprache auch Anforderungen an eine neu zu implementierende Migrationskomponente formulieren (vgl. Abschnitt 4.7).

Um den Anforderungen an die Ausführungsumgebung hinsichtlich der kontextabhängigen, eigenverantwortlichen Migration, die bereits in Abschnitt 4.3 erarbeitet wurden, gerecht zu werden, bedarf es einiger Erweiterungen der Mobilitätssprache. Diese Erweiterungen werden zunächst erläutert, dann folgt die formale Anforderungsbeschreibung.

4.6.1 Erweiterung der Mobilitätssprache

Die Grammatik der Mobilitätssprache ist in der *Erweiterten Backus-Naur Form* gegeben. Für die Beschreibung der einzelnen Regeln werden folgende Metasymbole verwendet: + separiert eine Sequenz von Symbolen, Alternativen werden von eckigen Klammern umschlossen `[...|...]`, Wiederholungen mit mindestens einem Element werden durch geschweifte Klammern `{...}` gekennzeichnet und runde Klammern `(...)` markieren optionale Symbole.

```
<Migration>      ::= <Initiator> + <Mobility> + (<Safety>) +
                     <DestinationAddress> + <Effect> + <Error>
<Safety>         ::= 'MigrationSafety' + '=' + 'Backup' + '.' +
                     <BackupType> + <BackupLocation>
<BackupType>     ::= 'BackupType' + '=' + ['WeakBackup' +
                     (';' + 'StrongBackup') | 'StrongBackup'] + '.'
<BackupLocation> ::= 'BackupLocation' + '=' + ['CurrentAgency' +
                     (';' + 'RemoteAgency') | 'RemoteAgency'] + '.'
```

Die Regel für die Migration wird um ein neues, optionales Nichtterminal `Safety` ergänzt. Dieses Nichtterminal trägt der Forderung nach Eigenverantwortlichkeit eines Agenten für seine Migration Rechnung. Auch wenn der Agent nicht der Initiator eines Migrationsvorgangs ist, soll er die Möglichkeit haben, Vorkehrungen zu treffen, die einen endgültigen Verlust des Kontaktes zu seinem Benutzer verhindern könnten. Eine mögliche Vorkehrung wäre zum Beispiel das Anlegen einer Sicherheitskopie (engl. *backup*), die der Benutzer von sich aus zu einem späteren Zeitpunkt wieder reaktivieren kann, falls er keinen Zugriff auf das Original des Agenten bekommt. Das Anlegen einer Sicherheitskopie kann in Anlehnung an die Art der Migration in Form einer schwachen (`WeakBackup`) oder einer starken Kopie (`StrongBackup`) erfolgen, wobei das Mobilitätsmodell auch beide Möglichkeiten vorsehen kann. Die `BackupLocation`-Regel bestimmt die Ausführungsumgebung, in der die Sicherheitskopie hinterlegt werden soll. Dies kann entweder die aktuelle oder eine entfernte Ausführungsumgebung oder beides sein. So wäre es zum Beispiel möglich, dass ein Agent eine Kopie seiner selbst immer entfernt auf seiner Heimatplattform hinterlegt oder für den Fall, dass die Ausführungsumgebung das Hinterlegen von Kopien nicht ermöglicht, auf ein oder mehreren beliebigen entfernten Plattformen verteilt.

```
<Error>       ::= 'MigrationError' + '=' + ['ErrorMethod' | 'Exception'] +
                  ['+' + <Restarts>) | '.']
<Restarts>    ::= 'Restart' + '.' + 'Restart' + '=' +
                  ['WeakRestart' + (';' + 'StrongRestart') + '.' +
                  <WeakRestart> | 'StrongRestart']
<WeakRestart> ::= 'WeakRestart' + '=' +
                  {'FixedMethod' | 'ArbitraryMethod'} + '.'
```

Bei der Migration können eine Reihe von Fehlern an verschiedenen Stellen des Migrationsvorgangs auftreten. Wichtig ist hierbei die Unterscheidung, ob ein Fehler in der noch aktuellen Ausführungsumgebung oder bereits in der Ziel-Ausführungsumgebung auftritt. Ein Fehler in der aktuellen Ausführungsumgebung, zum Beispiel ein Fehler beim Serialisieren oder ein Verbindungsfehler zur entfernten Plattform, kann ggf. durch den Agenten selbst behandelt werden, dieser Fall ist bereits in der Mobilitätssprache abgedeckt. Ein Fehler in der Ziel-Ausführungsumgebung, zum Beispiel beim Deserialisieren aufgrund fehlenden Programmcodes oder beim Initialisieren und Starten des Agenten, ausgelöst durch zu geringe verfügbare Ressourcen, wird von der Mobilitätssprache nicht behandelt. Ein solcher Fehler würde zum Verlust des Agenten führen, da dieser zwar erfolgreich übertragen wurde und somit nicht mehr auf der Ursprungsplattform vorhanden ist, auf der Zielplattform jedoch nicht ausgeführt werden kann. Als Konsequenz könnte der Agent komplett wieder an die Ursprungsplattform zurückgeschickt und dort neu gestartet werden. Um dies zu vermeiden, wird die Mobilitätssprache dahingehend erweitert, dass bei einem Fehler, aufgetreten in einer beliebigen Phase des Migrationsvorgangs, umgehend eine ggf. hinterlegte Sicherheitskopie des Agenten reaktiviert wird. Falls der Agent eine starke Sicherheitskopie hinterlegt hat, kann ein starker Neustart (engl. *strong restart*) erfolgen, wobei die Ausführung des Agenten an der Stelle kurz vor der Migration wieder aufgenommen und der Fehler diesem entsprechend angezeigt wird. Im Falle eines schwachen Neustarts (engl. *weak restart*) wird die Ausführung bei einer bestimmten oder im Vorwege beliebig gewählten Methode wieder aufgenommen.

```
<Mobility> ::= 'Mobility' + '=' + [ 'Weak' + '.' + <Weak> |
               'Strong' + '.' + <Strong> |
               'Weak' + ';' + 'Strong' + '.' + <Weak> + <Strong>]
```

Die Mobilitätssprache erlaubt lediglich entweder eine schwache oder eine starke Migration zu definieren. Es ist jedoch für eine Ausführungsumgebung durchaus sinnvoll, beide Arten der Migration zu unterstützen. Die starke Migration ist für den Programmierer eines Agenten deutlich einfacher zu handhaben, da die Ausführungsumgebung in die Pflicht genommen wird, den Agenten samt seines Ausführungszustandes komplett zu sichern und wieder herzustellen, ohne dass der Programmierer hierfür Sorge tragen muss. Es ist jedoch nicht in allen Fällen sinnvoll einen Agenten stark migrieren zu lassen[15], da einerseits die hierfür notwendigen Erweiterungen des Programmcodes des Agenten die Ausführung verlangsamen, und andererseits das Transfervolumen bei der Migration erhöht wird.

Die Entscheidung, ob ein Agent stark migrieren können sollte, liegt schließlich beim Programmierer. Ein Agent, der die Möglichkeit zur starken Migration hat, kann jedoch auch zur Laufzeit, abhängig von seinem Kontext, eine schwache Migration vorziehen. In diesem Fall muss lediglich sein Objektzustand transferiert werden, was zu einer Verringerung des Transfervolumens und

[15] In [Braun und Rossak 2005, Seiten 85/86] wird der allgemeine Nutzen der starken Migration weitergehend diskutiert.

somit der Transferdauer führt. Beispielsweise könnte ein Agent bei drohendem Verbindungsverlust zu seinem Ziel, eine schwache Migration veranlassen, um die Chance zu erhöhen, wenigstens seinen Objektzustand in die Zielumgebung zu transferieren. Dort angekommen, kann er die Ausführungsumgebung schließlich veranlassen, eine beliebige Methode auszuführen, die es dem Agenten erlaubt, seine Ausführung weiter fortzusetzen.

```
<MigrationProtocol> ::= 'MigrationProtocol' + '=' +
                        ['Synchronous' + ('+' + 'FailureAtomic') +
                        (';' + 'Asynchronous') | 'Asynchronous'] + '.'
```

Ähnlich der oben erwähnten Erweiterung der `Mobility`-Regel, wird auch die Regel des Migrationsprotokolls ergänzt. Die Mobilitätssprache erlaubt nicht die Unterstützung sowohl einer synchronen als auch einer asynchronen Migration, sondern sieht diese beiden Verfahren als Alternativen an. Die Erweiterung gestattet nun beide Migrationsvarianten anzubieten. Der Vorteil eines asynchronen Protokolls ist dessen höhere Leistung hinsichtlich der zu übertragenden Datenmenge und der gesamten Übertragungszeit. Das Protokoll ist jedoch auch weniger zuverlässig, da die Ziel-Ausführungsumgebung den Empfang eines Agenten nicht bestätigt. Ein Protokoll wird als `FailureAtomic` bezeichnet, falls das Protokoll garantiert, dass der komplette Agent entweder vollständig oder gar nicht übertragen wird [Braun und Rossak 2005]. Die genaue Semantik ist aus der Literatur jedoch nicht ersichtlich. Falls sich `FailureAtomic` lediglich auf die Übertragung bezieht, entspräche dies in etwa einem synchronen Protokoll, da ein asynchrones Protokoll keine Garantie für eine erfolgreiche Übertragung bieten kann. Daher soll `FailureAtomic` in dieser Arbeit als *transaktional* angesehen werden und sich über alle Phasen des Migrationsprozesses erstrecken. Das bedeutet, dass ein Protokoll, welches die Migration als Transaktion versteht, nicht nur die vollständige Übertragung eines Agenten zusichert, sondern auch das erfolgreiche Initialisieren und Starten des Agenten in der Ziel-Ausführungsumgebung garantiert. Falls in einer beliebigen Phase der Migration ein Fehler auftritt, soll der komplette Vorgang als fehlerhaft abgebrochen und geeignete Wiederherstellungsmaßnahmen seitens der Ursprungsplattform ergriffen werden (vgl. Regel `Error`).

4.6.2 Beschreibung des Mobilitätsmodells

Im Folgenden werden die für diese Arbeit interessanten Aspekte des Mobilitätsmodells beschrieben. Die im Kontext weniger interessanten Details, zum Beispiel die Adressierung von Migrationszielen und die Datenraumverwaltung des Agenten, werden nicht näher erörtert. Die Beschreibung richtet sich nach der in Abschnitt 3.3.3 vorgestellten Grammatik der Mobilitätssprache sowie den im vorigen Abschnitt eingeführten Erweiterungen der Grammatik. Sinn dieser Beschreibung ist die formale Spezifikation von Anforderungen an eine Ausführungsumgebung und insbesondere an eine Migrationskomponente, welche im folgenden Abschnitt vorgestellt wird, zur Umsetzung des Konzeptes der autonomen, eigenverantwortlichen Migration von Agenten.

```
(1) CreateAt            = CurrentAgency; RemoteAgency.
(2) MigrationInitiator = Agent; OtherAgent withVeto; Agency withVeto; Owner.
(3) Mobility            = Weak; Strong.
(4) WeakMobility        = ArbitraryMethod + Command.
(5) StrongMobility      = SourceCodeTransformation.
(6) MigrationEffect     = Move; Copy; Cloning;
```

```
(7) MigrationSafety    = Backup.
(8) BackupType         = WeakBackup; StrongBackup;
(9) BackupLocation     = CurrentAgency; RemoteAgency.
(10) MigrationError    = Exception + Restart.
(11) Restart           = WeakRestart; StrongRestart.
(12) WeakRestart       = ArbitraryMethod.
(13) MigrationProtocol = Synchronous + FailureAtomic; Asynchronous.
```

In der ersten Zeile wird die Anforderung formuliert, dass ein Agent entweder in der Ausführungsumgebung, in der das Migrationskommando initiiert wurde, gestartet werden kann, oder aber gleich in einer entfernten Ausführungsumgebung. Das entfernte Starten ist zum Beispiel dann sinnvoll, wenn die aktuelle Ausführungsumgebung nicht ausreichend Ressourcen zur Verfügung hat, um einen Agenten zu starten oder ein neu zu startender Agent ohnehin direkt nach seinem Start in eine andere Ausführungsumgebung migrieren soll.

Das Initiieren einer Migration kann von verschiedenen Seiten veranlasst werden (Zeile 2). Entweder der Agent selbst ruft ein entsprechendes Kommando auf oder ein anderer Agent derselben Ausführungsumgebung, die Ausführungsumgebung selbst oder der Benutzer des Agenten. Falls ein anderer Agent die Migration initiiert, soll der zu migrierende Agent ein Vetorecht haben und die Aufforderung ablehnen können. Gleiches gilt bei der Ausführungsumgebung als Initiator, wobei ein Veto seitens des Agenten in diesem Fall nicht zwangsläufig von der Ausführungsumgebung berücksichtigt werden muss (vgl. Abschnitt 4.3.3). Veranlasst der Benutzer die Migration, soll der Agent keinen Einspruch erheben können und wird von der Ausführungsumgebung zwangsläufig migriert.

Die Unterstützung sowohl der schwachen als auch der starken Migration wird in den Zeilen 3 bis 5 gefordert. Auch wenn der Nutzen der starken Migration in der Literatur diskutiert wird (vgl. [Braun und Rossak 2005]), soll es letztlich dem Programmierer überlassen werden, diese Möglichkeit einzusetzen. Die starke Migration soll nicht durch Änderungen an der virtuellen Maschine, sondern durch Erweiterungen des Programmcodes eines Agenten realisiert werden (Zeile 5). Bei der schwachen Migration soll mit Aufruf des Migrationskommandos ein Methodenname übergeben werden können, welcher die auf der entfernten Plattform aufzurufende Methode im Programmcode des Agenten repräsentiert (Zeile 4). So kann der Programmierer in einfacher Weise Kontrolle über den Programmablauf vor und nach einer Migration behalten.

Die Möglichkeit einen Agenten in entfernte Ausführungsumgebungen zu kopieren oder einen Agenten im Netzwerk zu verteilen, wird durch den Migrationseffekt beschrieben (Zeile 6). Dieser sieht unter anderem das hinsichtlich einer Migration normale Verschieben eines Agenten vor. Hierbei wird ein migrierender Agent von der Ursprungsplattform entfernt, auf eine entfernte Plattform transferiert und dort schließlich wieder gestartet, sodass zu jedem Zeitpunkt immer nur ein Exemplar des Agenten aktiv ist. Der Punkt, an dem die Ausführung startet, richtet sich nach der Art der Mobilität (entweder schwach oder stark). Anstatt einen Agenten zu verschieben, kann es in manchen Szenarien jedoch durchaus sinnvoll sein, mehrere Instanzen eines Agenten zuzulassen (vgl. Anwendungsszenario *Verteilte Berechnungen* sowie Abschnitt 4.8.1, *ASCML*). Bei dem Kopieren respektive dem Klonen, wird der Agent nicht auf seiner Ursprungsplattform beendet. Das Kopieren bewirkt, dass der Programmcode des Agenten in eine entfernte Ausführungsumgebung übertragen und dort neu ausgeführt wird, während das Klonen eine Kopie sowohl des Programmcodes als auch des Ausführungszustandes an die entfernte Plattform transferiert und den Agenten dann an einer bestimmten Stelle im Programmablauf startet.

Um der Forderung nach Eigenverantwortlichkeit eines Agenten für seinen Migrationsvorgang Rechnung zu tragen, wird in den Zeilen 7 bis 9 die Unterstützung von Sicherheitskopien beschrieben. Diese Kopien sollen es ermöglichen, den Agenten in einem bestimmten Zustand wieder herzustellen, falls der Zugriff auf das Original nicht mehr möglich ist. Ähnlich den Arten der Migration, wird bei den Sicherheitskopien ebenfalls zwischen einer schwachen und einer starken Kopie unterschieden. Eine schwache Kopie enthält lediglich den Objektzustand eines Agenten sowie den Namen der Methode, die bei Reaktivierung von der Ausführungsumgebung aufgerufen werden soll. Eine starke Kopie hingegen enthält den kompletten Zustand eines Agenten inklusive des Aufrufstapels und aller lokalen Variablen, sodass die Ausführung an der Stelle fortgesetzt werden kann, an der das Erstellen der Sicherheitskopie ursprünglich veranlasst wurde. Die Sicherheitskopie muss nicht zwangsläufig auf der aktuellen Plattform des Agenten hinterlegt werden. Es soll zusätzlich möglich sein, ein oder mehrere entfernte Ausführungsumgebungen anzugeben, an die die Kopie verschickt wird (Zeile 9). Im Gegensatz zur Migration, wird diese Kopie jedoch nicht entfernt gestartet, sondern lediglich verwahrt.

Die Behandlung von Fehlern, die während des Migrationsprozesses auftreten können, beschreiben die Zeilen 10 bis 12. Die Reaktion auf ein Problem im normalen Migrationsprozess soll von einem Agenten bzw. der Ausführungsumgebung durch individuelle Ausnahmebehandlungen (engl. *exception handling*) erlaubt werden. Das bedeutet, dass eine Ausführungsumgebung bzw. ein Agent explizit einen Fehler angezeigt bekommt und explizit darauf reagieren muss. Dies betrifft nicht nur Fehler, die sich auf der Ursprungsplattform ereignen, sondern auch Fehler, die in den letzten Phasen des Migrationsvorgangs auf der entfernten Plattform auftreten können. Die entfernte Plattform muss in diesem Fall das Auftreten und den Grund des Fehlers an die Ursprungsplattform zurück kommunizieren. Diese kann dann auf das Problem reagieren, indem sie beispielsweise bei Verlust eines Agenten, eine möglicherweise angelegte Sicherheitskopie reaktiviert. Abhängig davon, ob es sich um eine schwache oder starke Kopie handelt, kann dann entsprechend ein schwacher bzw. starker Neustart von der Ausführungsumgebung veranlasst werden (Zeile 11).

Der letzte Punkt betrifft einerseits die Sicherheit und andererseits die Leistung einer Migration (Zeile 13). Das zur Kommunikation zwischen der Ursprungs- und der Ziel-Ausführungsumgebung verwendete Migrationsprotokoll, soll in zwei Varianten von den Ausführungsumgebungen abgewickelt werden können. Ein asynchrones Protokoll ist schneller, aber auch weniger verlässlich als ein synchrones Protokoll, denn die Zielplattform sendet bei erfolgreichem Empfang des Agenten keine Bestätigung an die Ursprungsplattform, sodass diese weder über den Erfolg noch den Misserfolg der Migration in Kenntnis gesetzt wird. Anders verhält es sich bei dem synchronen Protokoll. Hier wartet die Ursprungsplattform auf eine Empfangsbestätigung der Zielplattform. Erhält sie diese nach Ablauf einer Frist nicht, kann sie die Migration vollständig abbrechen oder einen erneuten Versuch unternehmen. Die Option *FailureAtomic* sorgt zusätzlich dafür, dass nicht nur die Übertragung des Agenten, sondern auch dessen Deserialisierung und Start in der entfernten Ausführungsumgebung fehlerfrei verlaufen müssen, andernfalls gilt der Vorgang als nicht erfolgreich.

Die Beschreibung des Mobilitätsmodells findet im folgenden Abschnitt weitere Verwendung. Hier wird eine Komponente vorgestellt, die den Prozess der Migration mit allen damit zusammenhängenden Anforderungen abwickelt.

4.7 Migrationskomponente

Für alle in den vorhergehenden Abschnitten aufgeführten Ansprüche an die Migration von Agenten, bedarf es einer Komponente, die diese Anforderungen umsetzen kann und eine Schnittstelle für den Aufruf von Migrationsfunktionen bereitstellt. In diesem Abschnitt sollen die Aufgaben dieser Komponente im Detail erläutert werden, ohne jedoch Bezug auf eine konkrete Umsetzung zu nehmen.

4.7.1 Versenden von Agenten

Eine der grundlegenden Aufgaben einer Migrationskomponente ist das Versenden von Agenten. Entsprechend des im vorigen Abschnitt beschriebenen Mobilitätsmodells, kann der Versand entweder von einem Agenten selbst, einem anderen Agenten, der Plattform und letztlich auch vom Benutzer durch Aufruf einer entsprechenden Funktion mit dem Migrationsziel als Parameter initiiert werden. Mit diesem Aufruf beginnt der Migrationsvorgang, welcher in sechs verschiedene, aufeinander folgende Phasen unterteilt werden kann [Braun und Rossak 2005]. In der ersten Phase muss die Migrationskomponente die Ausführung des zu migrierenden Agenten anhalten, das bedeutet, dass der Prozess (engl. *thread*), in dem der Agent ausgeführt wird, sowie alle von diesem erstellten Subprozesse, unterbrochen werden müssen[16]. Dies ist erforderlich, damit in der zweiten Phase der Migration die Daten des Agenten (d.h. die Variablenbelegungen) sowie ggf. dessen Ausführungszustand (d.h. Aufrufstapel, lokale Variablen und der Befehlszeiger) serialisiert werden können. Die serialisierte Repräsentation des Agenten kann anschließend in der dritten Phase der Migration an die Ziel-Ausführungsumgebung versandt werden. Die Details des Transfers folgen später in diesem Abschnitt.

4.7.2 Empfangen von Agenten

Der Empfang eines Agenten umfasst die letzten drei Phasen des oben angesprochenen Migrationsprozesses, beginnt mit der Verarbeitung der empfangenen Migrationsnachricht und ist in erster Linie abhängig vom verwendeten Migrationsprotokoll. Die Verarbeitung beginnt mit der Überprüfung, ob der empfangene Agent überhaupt akzeptiert werden soll. Die Basis hierfür stellen die Metainformationen über den Benutzer des Agenten und die versendende Ausführungsumgebung dar, welche in der Migrationsnachricht enthalten sind. An dieser Stelle können Sicherheitsrichtlinien, welche vom Benutzer der Ausführungsumgebung vorgegeben werden können, angewandt und Agenten, die von unbekannten oder nicht vertrauenswürdigen Ausführungsumgebungen kommen, abgelehnt werden. Eine erfolgreiche Überprüfung führt zur nächsten Phase des Migrationsprozesses. In diesem Schritt wird der empfangene Agent deserialisiert. Alle Variablenbelegungen des Agenten sowie ggf. dessen Ausführungszustand werden wieder hergestellt. Das Ergebnis dieses Schrittes sollte eine exakte Kopie des Agenten vor dem Aufruf des Migrationsbefehls auf der Ursprungsplattform sein. In der letzten Phase muss die Migrationskomponente einen neuen Prozess für die Ausführung des Agenten erstellen und den Agenten schließlich starten.

[16] Da die gesamte Datenraumverwaltung (vgl. Abschnitt 3.1.2) im Rahmen dieser Arbeit nicht behandelt werden kann, wird der Einfachheit halber angenommen, dass ein Agent lediglich in einem Prozess ausgeführt wird und keine weiteren Subprozesse erstellen kann.

Für das Deserialisieren und Starten des Agenten wird natürlich dessen Programmcode benötigt. An dieser Stelle wird angenommen, dass die empfangende Ausführungsumgebung den Programmcode entweder vorliegen hat oder automatisch nachladen kann. Die Beschreibung der genauen Mechanismen, wie der Programmcode zur Verfügung gestellt wird, erfolgt zu einem späteren Punkt in diesem Abschnitt.

4.7.3 Sicherheitskopien

Eine Möglichkeit, sich vor dem Verlust eines Agenten zu schützen, ist das Anlegen von Sicherheitskopien. Ein Agent soll von sich aus entscheiden können, eine Kopie seiner selbst anlegen zu lassen und die Migrationskomponente mit der Ausführung beauftragen können. Entsprechend der Beschreibung des Mobilitätsmodells aus Abschnitt 4.6.2, gibt es zwei Formen von Kopien: schwache und starke Sicherheitskopien. Schwache Kopien enthalten lediglich den Objektzustand eines Agenten, also alle Variablenbelegungen, starke Kopien zusätzlich noch den Ausführungszustand. Außerdem sollte eine Sicherheitskopie weitergehende Metainformationen, wie zum Beispiel den Zeitpunkt der Erstellung, enthalten.

Das Anlegen einer Sicherheitskopie umfasst in etwa dieselben Phasen wie das Versenden eines Agenten bei der Migration und kann daher auf denselben Mechanismen aufbauen. Hierzu gehören das (zeitweilige) Unterbrechen der Ausführung (Phase 1) sowie das Serialisieren des Agenten (Phase 2). Im Anschluss an das Serialisieren migriert der Agent jedoch nicht, sondern wird auf der aktuellen Ausführungsumgebung weiter ausgeführt. Die serialisierte Repräsentation kann entweder persistent gemacht werden oder bei Bedarf auch an entfernte Ausführungsumgebungen versandt werden (Phase 3 und 4). Wird eine Ausführungsumgebung zu einem späteren Zeitpunkt aufgefordert eine Sicherheitskopie wieder zu reaktivieren, kann sie sich der Mechanismen der fünften und sechsten Phase des Migrationsprozesses bedienen. In diesen Phasen wird die Sicherheitskopie wieder deserialisiert und die Ausführung des Agenten anschließend wieder aufgenommen.

4.7.4 Verschieben, Kopieren und Klonen

Im Normalfall wird ein migrierender Agent von einer Ausführungsumgebung in eine andere Ausführungsumgebung verschoben, wobei zu jedem Zeitpunkt ausschließlich eine Instanz des Agenten vorhanden ist. Den Anforderungen entsprechend soll es darüber hinaus zusätzlich möglich sein, anstatt einen Agenten zu verschieben, diesen auch entfernt zu kopieren oder zu klonen, sodass in diesen Fällen mehrere Instanzen eines Agenten vorhanden sind. Beim Kopieren eines Agenten soll eine frische Instanz, also ohne besondere Variablenvorbelegungen, in einer entfernten Ausführungsumgebung neu gestartet werden. Hierfür muss die Migrationskomponente lediglich eine Aufforderung zum Starten eines neuen Agenten an die Ziel-Ausführungsumgebung übermitteln. Beim Klonen soll eine exakte Kopie des Agenten, also inklusive aller aktuellen Variablenbelegungen sowie ggf. dem Ausführungszustand, in einer entfernten Ausführungsumgebung erstellt und gestartet werden. Die Mechanismen hierfür sind identisch mit denen, die bei der starken Migration eines Agenten bzw. dem Anlegen einer starken Sicherheitskopie verwendet werden, nur dass der migrierende Agent auf seiner Ursprungsplattform auch nach dem Transfer weiterhin ausgeführt wird.

4.7.5 Verwaltungsoperationen

Das Verschieben, Kopieren und Klonen von Agenten erfordert eine Reihe von Verwaltungsoperationen, um die Konsistenz eines Multiagentensystems zu wahren. Beispielsweise müssen die beteiligten Migrationskomponenten beim Verschieben eines Agenten, diesen bei dem *Agent Management System* (AMS) der Ursprungsplattform abmelden und bei dem AMS der Zielplattform wieder neu anmelden. Außerdem bedarf es Mechanismen, um Nachrichten, adressiert an einen bereits verschobenen Agenten, diesem in die neue Ausführungsumgebung hinterher zu schicken.

Speziell beim Kopieren bzw. Klonen von Agenten ergibt sich darüber hinaus das Problem der Benennung der Instanzen. Die *FIPA*[17] spezifiziert in [FIPA 2002a, FIPA 2004], dass Namen von Agenten global eindeutig sein müssen und schlägt vor, diese in der Form von `<SymbolischerName>@<NameDerUrspungsplattform>` zu vergeben. Zum einen wirft das die Frage auf, ob hiermit die Ursprungsplattform des Originals gemeint ist oder die Ursprungsplattform, auf der die neue Instanz erzeugt wurde? Zum anderen führt diese Benennung zu Konflikten sobald mehr als eine Kopie erzeugt wird. Eine Lösung für dieses Problem soll an dieser Stelle nicht gegeben werden, da sie vielmehr abhängig von der jeweiligen Implementation der Ausführungsumgebung ist.

4.7.6 Entfernte Befehlsausführung

Eine Migrationskomponente soll andere, entfernte Migrationskomponenten mit der Ausführung bestimmter Operationen beauftragen können. Hierzu gehören, neben dem bereits erwähnten Starten von Agentenkopien, beispielsweise auch das Reaktivieren von Sicherheitskopien, das Löschen eines Agenten oder einer Sicherheitskopie sowie die Migration eines Agenten. Zum Beispiel könnte ein Benutzer veranlassen, dass sein Agent wieder auf seine Heimatplattform zurück migrieren soll, und diesen Befehl über ein oder mehrere Migrationskomponenten hinweg im Netzwerk verbreiten. Die Migrationskomponente der aktuellen Ausführungsumgebung des Agenten muss daraufhin den Migrationsprozess für den Agenten einleiten. Der Sicherheit wegen sollte die Ausführung eines Befehls mit der Autorisierung des Befehlsgebers einhergehen, um Missbrauch zu unterbinden.

4.7.7 Transfermechanismen

Die Migration eines Agenten besteht häufig nicht nur aus dem Versand der Daten bzw. des Ausführungszustandes des Agenten. Es müssen unter Umständen auch Teile des Programmcodes transferiert werden, die für die Deserialisierung oder das Ausführen des Agenten in der entfernten Ausführungsumgebung notwendig sind. Wie bereits in den Abschnitten 3.1.2, 3.2.5 und 3.3.2 erläutert, gibt es verschiedene Strategien, um dies zu realisieren. Man unterscheidet hierbei im Wesentlichen zwischen *push*- und *pull*- Strategien, und kann zusätzlich eine Reihe von Effizienzoptimierungen, z.B. durch Aufsetzen von Codeservern, Codespeichern etc., durchführen. Diese Optimierungen sind abhängig vom Zustand der Umwelt eines Agenten (z.B. Verbindungsqualitäten des Netzwerkes) sowie von der geplanten Reiseroute (engl. *itinery*) und werden entweder von dem Agenten selbst vorgeschlagen oder automatisch von der Migrationskomponente durchgeführt.

[17] *Foundation of Intelligent Physical Agents*

4.7.8 Fehlerbehandlung

Falls während eines Migrationsvorgangs Fehler auftreten, müssen diese von der Migrationskomponente erkannt und es muss dann entsprechend darauf reagiert werden. Für das Erkennen von Fehlern müssen an kritischen Stellen des Migrationsvorgangs entsprechende Ausnahmebehandlungen (engl. *exception handling*) vorgesehen werden, sodass Fehler, die bereits auf den unteren Schichten der Ausführungsumgebung auftreten (z.B. Verbindungsfehler), aufgefangen werden können. Sobald ein Fehler erkannt wurde, soll die Migrationskomponente selbst versuchen, den Fehler zu beheben, sofern dies möglich und sinnvoll erscheint (z.B. durch erneuten Verbindungsaufbau). Wenn der Fehler schließlich nicht zu beseitigen ist, muss die Migrationskomponente dem Initiator der Migration diesen Fehler anzeigen. Abhängig von der Art des Fehlers und der Migrationsphase in der der Fehler aufgetreten ist, soll die Migrationskomponente von sich aus weitergehende Wiederherstellungsmaßnahmen treffen können. Wenn es beispielsweise beim Versenden eines bereits serialisierten Agenten zu einem permanenten Verbindungsabbruch kommt, soll die Migrationskomponente den zu migrierenden Agenten wieder deserialisieren und starten sowie dem Agenten den Fehler anzeigen.

Ein wenig anders verhält es sich mit Fehlern, die in den letzten drei Phasen des Migrationsvorgangs in der Ziel-Ausführungsumgebung auftreten. Vorstellbar wäre zu Beispiel, dass der serialisierte Agent nicht vollständig oder beschädigt empfangen wird, die Annahme bzw. Ausführung des Agenten aufgrund von Sicherheitsrichtlinien verweigert wird, die Deserialisierung aufgrund fehlenden Programmcodes misslingt, zu wenig Ressourcen zum Starten des Agenten vorhanden sind etc. In einem solchen Fall muss die empfangene Migrationskomponente den erkannten Fehler, sofern der Fehler nicht selbst behoben werden kann, an die sendende Migrationskomponente zurück kommunizieren, welche daraufhin die oben erwähnten Ausnahmebehandlungen durchführen und ggf. eine Sicherheitskopie des Agenten reaktivieren sollte. Kann der Fehler zum Beispiel aufgrund eines Verbindungsabbruchs nicht kommuniziert werden, so muss der serialisierte Agent als Sicherheitskopie gespeichert werden, andernfalls gilt er als verloren.

4.7.9 Konfigurierbarkeit

Ähnlich wie die Komponente, die für die Resource Awareness zuständig ist, sollte sich auch die Migrationskomponente möglichst weitgehend vom Benutzer konfigurieren lassen. Die Möglichkeiten der Konfiguration umfassen insbesondere die unterstützten Arten der Migration (schwach bzw. stark), Migrationsprotokolle (synchron, asynchron), die Möglichkeiten Sicherheitskopien zu hinterlegen, aber insbesondere auch Sicherungsmaßnahmen. Zu diesen Maßnahmen gehören beispielsweise die Verweigerung des Empfangs von Agenten fremder Plattformen, Unterstützung von Authentifizierungstechniken, Angabe maximaler Aufenthaltsdauer für Agenten, Zugriff der Agenten auf lokale Ressourcen (z.B. über eine *Sandbox*) etc. Auch für die Konfiguration der Migrationskomponente gilt, dass die Einstellungen zwar tiefgreifend möglich sein sollten, der Benutzer aber als Erleichterung eine Menge vorgefertigter Profile angeboten bekommt, die seine Auswahl erleichtern und nicht das ganzheitliche Verständnis aller Einstellungen erfordert.

4.8 Verwandte Arbeiten

Im Folgenden sollen mit dem *Agent Society Configuration Manager and Launcher* (ASCML) und der *Proactive Navigation* (ProNav) zwei Projekte vorgestellt werden, die im Kontext dieser Arbeit als besonders interessant und in gewisser Weise verwandt erachtet werden. Eine Reihe weiterer Arbeiten, die jeweils Teilaspekte des in dieser Arbeit entwickelten Konzeptes abdecken, wurden bereits in den Abschnitten referenziert, in denen auf diese Arbeiten Bezug genommen wurde. Arbeiten, die denselben Fokus wie diese Arbeit haben, konnten in der Literatur nicht gesichtet werden.

4.8.1 ASCML

Der *Agent Society Configuration Manager and Launcher* (ASCML) ist ein Deployment[18]-Werkzeug für Multiagentensysteme [Braubach u. a. 2005, Bellifemine u. a. 2007]. Der Begriff des Deployments bezeichnet nach [OMG 2003] die Prozesse zwischen der Anschaffung und der Ausführung von Software, wozu unter anderem die Installation, Konfiguration und das Starten der Anwendung gehören. In verteilten Systemen und speziell in verteilten Multiagentensystemen sind diese Prozesse nicht trivial, da unter Umständen eine Vielzahl an Ausführungsumgebungen beteiligt sind und ggf. Abhängigkeiten zwischen einzelnen Komponenten einer Software bzw. einer Multiagentenanwendung berücksichtigt werden müssen.

Der ASCML soll hierbei den Benutzer einer verteilten Anwendung unterstützen und das transparente Verteilen und Starten der einzelnen Bestandteile einer Software übernehmen. Diese Problematik soll anhand eines Beispiels verdeutlicht werden. Angenommen, eine Anwendung bestünde aus drei Agenten A, B und C, die in einem Netzwerk auf jeweils unterschiedlichen Knoten laufen sollen. Die Anwendung erfordert es, dass A vor B und B vor C gestartet werden muss, da die Agenten voneinander abhängig sind. Anstatt dass der Benutzer die jeweiligen Agenten nun manuell in ihren Ausführungsumgebungen startet, verwendet er den ASCML, um von einem zentralen Knoten aus die Agenten zu konfigurieren, deren entfernten Start in den jeweiligen Ausführungsumgebungen zu veranlassen und anschließend deren Zustand zu überwachen.

Um dieses Ziel zu erreichen, muss der Benutzer eine genaue Kenntnis der Anwendung besitzen. Ihm obliegt es, eine sogenannte Agentengesellschaft (engl. *agent society*) zu erstellen, welche die Anwendung repräsentiert. Für die Agenten dieser Gesellschaft lassen sich dann Parameter und Abhängigkeiten zu bestimmten Ressourcen (andere Agenten, Dienste etc.) definieren, sowie die intendierte Ausführungsumgebung angeben. Der Start einer solchen Gesellschaft führt schließlich zum Versand von Startaufforderungen an entfernte Knoten, die ihrerseits für die Auflösung der Abhängigkeiten eines Agenten sorgen können und den Agenten ausführen.

Gemeinsamkeiten und Unterschiede

Zwischen dem ASCML und dem in dieser Arbeit vorgestellten Konzept lässt sich insbesondere ein gemeinsamer Anwendungszweck erkennen, obgleich dieser unterschiedlich gewichtet und ausgeprägt ist. Grob lässt sich der Anwendungszweck des ASCML als das verteilte Starten einer agentenbasierten Anwendung zusammenfassen. Auch das hier vorgestellte Konzept sieht eine Verteilung von Agenten vor, wobei ebenfalls der Benutzer von der manuellen Verteilung entlastet werden soll. Allerdings unterstützt dieses Konzept andere Phasen des Deployment-Prozesses, beruht auf der Mobilität von Agenten, stellt deren Autonomie in den Vordergrund und unterstützt

[18] zu deutsch: Aufstellung, Bereitstellung, Entsendung

dynamische Umgebungen. Während der ASCML insbesondere die Konfiguration sowie das verteilte Starten einer Anwendung unterstützt (Phase 3 und 5 nach [OMG 2003]), fokussiert das in dieser Arbeit vorgestellte Konzept die Phase der Startvorbereitung (engl. *preparation phase*, Phase 4), in der die Verteilung des Programmcodes auf die einzelnen Ausführungsumgebungen mittels mobiler Agenten vorgenommen wird. Dies wird vom ASCML nicht oder nur rudimentär unterstützt und beruht auf dem Nachladen von Klassen über das Netzwerk unter Verwendung des Java-Classloaders, was wenig effizient ist (vgl. Abschnitt 3.2.4).

Weitere Unterschiede finden sich in der Rolle des Benutzers. Der ASCML verlangt für die manuelle Konfiguration einer Anwendung ein tiefes Verständnis der Anwendungsarchitektur von dem Benutzer (und entlastet somit den Programmierer), erlaubt aber auch die Kontrolle der Mechanismen seitens des Benutzers. Im hier vorgestellten Konzept soll der Benutzer keinerlei Kenntnis von der Beschaffenheit einer Anwendung haben müssen, was durch eine höhere Autonomie seitens der Agenten sowie höhere Anforderungen an den Programmierer und die Ausführungsumgebung kompensiert werden muss. Zum Beispiel muss der Programmierer Anforderungen an eine Ausführungsumgebung für die Agenten definieren, erst dann können sich diese zur Laufzeit autonom (unter Zuhilfenahme ihres Umweltmodells) für eine geeignete Ausführungsumgebung entscheiden und dorthin migrieren.

4.8.2 ProNav

Das *ProNav*-Rahmenwerk[19] wurde ebenso wie das in Abschnitt 3.3.2 vorgestellte *Kalong*-Mobilitätsmodell an der Universität Jena entwickelt. Es fokussiert insbesondere das Rundreiseproblem (engl. *itinery problem*) mobiler Agenten und beschreibt ein Rahmenwerk, welches den Migrationsprozess verbessern und die Autonomie sowie die Adaptivität eines Agenten hinsichtlich der Planung und Durchführung einer Rundreise erhöhen soll [Erfurth und Rossak 2002].

Grundlegend bei dem im Folgenden beschriebenen Konzept ist die Annahme, dass ein mobiler Agent typischerweise zur Bearbeitung einer Aufgabe mehr als eine Ausführungsumgebung in einem Netzwerk besucht. Demnach braucht der Agent eine ihm bekannte Reiseroute, die er Schritt für Schritt abarbeitet. Dabei kann er in jeder Ausführungsumgebung, die er besucht, aufgrund der dort verfügbaren Ressourcen ein oder mehrere Teilaufgaben ausführen, bis er am Ende seiner Rundreise wieder auf dem Gerät seines Benutzers ankommt und diesem das Ergebnis der Bearbeitung präsentieren kann.

ProNav adressiert eine relativ dynamische Umwelt, in der unter Umständen eine Vielzahl von Ressourcen vorhanden sein können. Um mit diesen Anforderungen umzugehen, führt ProNav eine logische Ebene ein, die ein Netzwerk in dezentrale und selbstorganisierende Domänen unterteilt (vgl. *JXTA*, Abschnitt 2.3.6). Eine solche Domäne besteht aus allen Entitäten eines Subnetzes und stellt einen abgeschlossenen Raum der Informationsverbreitung dar. Das bedeutet, Entitäten einer Domäne tauschen untereinander Informationen aus, nicht aber über die Grenzen der Domäne hinweg.

Eine Domäne wird durch einen sogenannten *Domain Manager* betreut, welcher kontinuierlich durch ein Auswahlverfahren unter den Teilnehmern einer Domäne neu bestimmt bzw. bestärkt wird. Der Domain Manager stellt über einen sogenannten *Domain Master* die Verbindung zwischen einzelnen Domänen her und sorgt auf dieser hierarchisch höheren Ebene für einen Austausch von zusammengefassten Domäneninformationen. Auf diese Weise hat jeder Teilnehmer einer Domäne einerseits einen umfassenden Blick auf seine Domäne und anderer-

[19] *ProNav* steht für Proaktive Navigation

seits Zugriff auf eine Zusammenfassung von Informationen anderer Domänen [Döhler u. a. 2004, Erfurth und Rossak 2002]. Um in einem solchen logischen Netzwerk eine nahezu optimale Strategie für die Rundreise eines Agenten zu finden, sieht die ProNav-Architektur drei wesentliche Komponenten vor: 1) die dynamische Domänenkarte (engl. *dynamic domain maps*), 2) einen Routenplaner und 3) einen Migrationsplaner (siehe Abbildung 4.5).

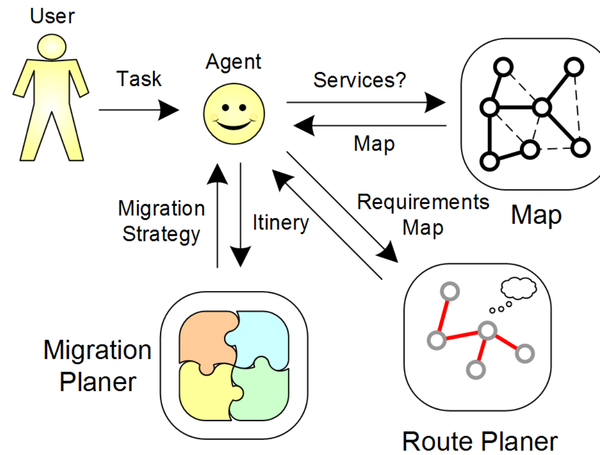

Abbildung 4.5: ProNav Infrastrukturkomponenten (nach [Erfurth und Rossak 2002])

Dynamische Domänenkarte Jedes Gerät des Netzwerkes führt eine eigene dynamische Domänenkarte, welche Informationen über andere verfügbare Ressourcen innerhalb sowie über die Grenzen der Domäne hinaus, enthält. Aufgrund der potenziell hohen Anzahl an verfügbaren Ressourcen ist es jedoch nicht sinnvoll, alle Informationen über alle Ressourcen auf jedem Gerät vorzuhalten. Stattdessen wird die Domänenkarte in detaillierte Informationen über lokale Geräte derselben Domäne, und vage Informationen über andere entfernte Domänen, unterteilt. Die Informationen über Geräte derselben Domäne bestehen aus den Eigenschaften eines Gerätes, angebotene Dienste und vor allem Verbindungsqualitäten zu diesen Geräten. Diese Informationen nutzt ein Agent, um Ressourcen aufzufinden und um Zugriff auf die Verbindungsqualitäten zu diesen Ressourcen zu erhalten.

Die Verbindungsqualitäten werden durch Sensoren, die jede Ausführungsumgebung bereitstellt, gemessen. Verschiedene Arten von Sensoren können die Auslastung und Antwortzeiten von Knoten, die Datenrate einer Netzwerkverbindung etc. messen, indem sie beim Versand der Ressourceninformationen Leistungsmessungen auf Ebene der Middleware durchführen. Außerdem ist eine Vorhersage der zukünftigen Entwicklung dieser Daten auf Basis von Informationen vergangener Messungen möglich [Erfurth 2004].

Routenplaner Mit Hilfe der dynamischen Domänenkarte kann der Routenplaner entsprechend den Anforderungen eines Agenten einen Weg durch das Netzwerk planen. Ziel dieser Planung ist die Berechnung eines kürzesten Pfades, auf dem der Agent alle von ihm benötigten Ressourcen besuchen kann. Grundlage dieser Berechnung sind hierbei die Angaben der Domänenkarte über die Verbindungsqualitäten. Das Resultat stellt schließlich eine geordnete Liste mit Adressen entfernter Ausführungsumgebungen dar, die eine vorgeschlagene Reiseroute repräsentiert. Um die Autonomie des Agenten zu wahren, ist es diesem jedoch selbst überlassen, den Vorschlag anzunehmen oder ggf. nach eigenen Interessen abzuändern. Auch ist es einem Agenten jederzeit und

in jeder Ausführungsumgebung möglich, auf Basis aktuellerer Daten eine erneute Berechnung durchführen zu lassen, um die Route ggf. an neue Umstände anzupassen.

Migrationsplaner Hat der Agent sich schließlich für eine Reiseroute entschieden, übergibt er die Feinabstimmung an den Migrationsplaner. Dieser hat eine eher technische Sicht auf den Prozess der Migration und versucht die Menge an zu übertragenden Daten für jeden einzelnen Migrationsvorgang zu minimieren, indem er individuelle Migrationsstrategien für einen Agenten erstellt. Hier kommt das Kalong-Mobilitätsmodell zum Einsatz, welches in Abschnitt 3.3.2 detailliert erklärt wurde.

Gemeinsamkeiten und Unterschiede

Verglichen mit dem in dieser Arbeit vorgestellten Konzept, lassen sich sowohl einige Gemeinsamkeiten als auch einige Unterschiede feststellen. Insbesondere ist die Motivation bzw. das Ziel von ProNav ein anderes. Während diese Arbeit die autonome, eigenverantwortliche Migration und Verteilung von Agenten auf Basis des Wissens über die Umwelt und darin auftretender Ereignisse anstrebt, konzentriert sich ProNav auf das Erstellen einer näherungsweise optimalen Reiseroute unter Ausnutzung von Umweltinformationen. Etwaige Umweltereignisse, die einen Migrationsvorgang auslösen könnten, zum Beispiel das Auffinden neuer Ressourcen, das Beenden einer Ausführungsumgebung etc., werden von ProNav nicht modelliert. Dafür versucht ProNav insbesondere die Verbindungsqualitäten, als einen besonderen Aspekt der Umwelt, über verschiedenartige Sensoren zu erfassen und Vorhersagen über deren Entwicklung zu erstellen, was durchaus auch vielversprechende Möglichkeiten zur Weiterentwicklung des in dieser Arbeit vorgestellten Konzeptes bietet.

Weitere Unterschiede lassen sich in der Art der ausgetauschten Informationen ausmachen. Während das Umweltmodell eines Agenten in dieser Arbeit möglichst umfassende Informationen über alle erreichbaren Ressourcen bieten soll, konzentriert sich ProNav lediglich auf einen Ausschnitt dieser, nämlich allgemeine Eigenschaften von Geräten sowie detaillierte Informationen über angebotene Dienste und Verbindungsqualitäten. Gemeinsamkeiten finden sich hingegen wieder in dem Aufbau eines logischen Netzwerkes zur Begrenzung der Entfernung, über die Informationen ausgetauscht werden. Lediglich die Art, wie das logische Netzwerk aufgebaut und verwaltet wird, unterscheidet sich. Während sich ProNav hierfür auf die Unterteilung des physikalischen Netzwerkes in verschiedene logische Domänen stützt, erlaubt das in dieser Arbeit vorgestellte Konzept eine flexible Wahl des logischen Netzwerkes, abhängig von hierfür speziell vorgesehenen Adaptern. So ist es durchaus vorstellbar, dass dieselbe Ausführungsumgebung sowohl an einem *JXTA*-Netzwerk (vgl. Abschnitt 2.3.6), an einem *CARD*-Netzwerk (vgl. Abschnitt 2.4.6), an einer ProNav-Domäne (vgl. Abschnitt 4.8.2) als auch komplett ohne logisches Netzwerk an einem Bluetooth-Netz (vgl. Abschnitt 2.4.2) teilnimmt. Die Auswahl hierfür obliegt in erster Linie dem Benutzer sowie den von der Ausführungsumgebung vorgefundenen Umweltgegebenheiten.

4.9 Zusammenfassung

In diesem Kapitel wurden zwei Middleware-Komponenten konzipiert, die die Grundlage für die kontextabhängige und eigenverantwortliche Migration bilden. Die erste Komponente ist für die Wahrnehmung der Umwelt zuständig und stellt die Basis für den Kontext bereit, in dessen Abhängigkeit ein Agent eine Migrationsentscheidung treffen und die Art der Migration den Gegebenheiten der Umwelt anpassen kann. Die zweite Komponente zeichnet sich für alle Phasen des Migrationsvorgangs verantwortlich und bietet darüber hinaus Sicherungsmechanismen für migrierende Agenten, welche die Eigenverantwortlichkeit eines Agenten für seinen Weg durch das Netzwerk und schließlich zurück zu seinem Benutzer unterstützen.

Grundlage für die Aufstellung von Anforderungen bildeten fünf verschiedene Anwendungsszenarien, die jeweils unterschiedliche Aspekte des Konzeptes beleuchteten und einen möglichen Einsatz in der Praxis durch entsprechende Beispiele rechtfertigen sollen. Anhand dieser Anwendungsszenarien wurden Anforderungen in vier verschiedenen Klassen identifiziert. Die ersten beiden Klassen bildeten hierbei die Anforderungen an den Benutzer und die Agenten bzw. Programmierer, welche hinsichtlich der Benutzbarkeit so gering wie möglich gehalten wurden. Die dritte und größte Klasse betrifft die Anforderungen an die Ausführungsumgebung, welche den Agenten bzw. Programmierern möglichst einfach zu handhabende Schnittstellen für alle angeboten Funktionen zur Verfügung stellen, und damit die zugrunde liegende Komplexität verstecken sollte. In der letzten Klasse finden sich Anforderungen an die Sicherheit, die für einen praktischen Einsatz des Konzeptes unerlässlich sind, im Rahmen dieser Arbeit jedoch nicht weiter behandelt werden können.

Im Anschluss an die Anforderungsanalyse wurde das Modell einer Ressourcenumwelt spezifiziert, welches die im Kontext der Migration interessanten Entitäten und Ereignisse umfasst und die Basis der sogenannten Resource Awareness-Komponente darstellt. Diese Komponente ist, wie bereits oben erwähnt, für die Wahrnehmung der Umwelt, und damit für die Erstellung und Aktualisierung des Umweltmodells, zuständig. Um der Heterogenität und Dynamik der Umwelt gerecht zu werden, soll diese Komponente generische Mechanismen bieten, an die beliebige Discovery-Verfahren und Repräsentationssprachen für Ressourcen adaptiert und zur Laufzeit den Gegebenheiten der Umwelt angepasst werden können.

Im darauf folgenden Teil wurde mit der bereits vorgestellten Mobilitätssprache *MoL* das Mobilitätsmodell für die Migrationskomponente spezifiziert. Hierfür musste die Grammatik der Sprache erweitert werden, sodass die zusätzlichen Sicherungsmaßnahmen für die Migration in die Spezifikation einfließen konnten. Ausgehend von dieser Spezifikation wurden die Migrationskomponente und ihre Aufgaben im Einzelnen beschrieben. Hierzu gehören die wesentlichen Basisfunktionalitäten, wie das Versenden und Empfangen von Agenten, aber auch weitergehende Funktionen, wie zum Beispiel das Anlegen und Reaktivieren von Sicherheitskopien sowie die Behandlung von Fehlern. Zum Abschluss wurden mit *ProNav* und dem *ASCML* zwei Projekte vorgestellt, welche, verglichen mit dieser Arbeit, zwar unterschiedliche Ziele verfolgen, aber in Teilen ähnliche Ansätze zeigen. Insbesondere ProNav, welches ebenfalls eine kontextabhängige Migration verfolgt, bietet interessante Vergleichspunkte.

Im folgenden Kapitel soll die Implementation eines Prototypen vorgestellt werden, der das Konzept der kontextabhängigen und eigenverantwortlichen Migration in Form der in diesem Kapitel bereits beschriebenen Middleware-Komponenten umsetzt.

5 Prototypische Umsetzung des Konzeptes

In diesem Kapitel soll die prototypische Umsetzung zweier Komponenten beschrieben werden, welche Agenten die kontextabhängige und eigenverantwortliche Migration in heterogenen Umgebungen erlauben. Die Aufgaben der Komponenten sowie die Modelle, auf denen sie basieren, wurden bereits im vorigen Kapitel identifiziert und spezifiziert. Nachfolgend sollen die konkreten Anforderungen, etwaige Entwurfsentscheidungen sowie die Funktionsweise dieser Komponenten vorgestellt werden.

Die Wahl einer Ausführungsumgebung für die Komponenten ist die erste maßgebliche Entscheidung, durch die der weitere Entwurf grundlegend beeinflusst wurde. Sie wirkt sich auf die vorgestellten Architekturen sowie das Zusammenspiel der Komponenten untereinander aus. Um die Funktionsweise der Komponenten anzudeuten, werden neben den Anforderung und der Architektur für jede Komponente jeweils einige Kernpunkte der Ausführung vorgestellt. Den Abschluss dieses Kapitels bildet eine Zusammenfassung, in der ein Fazit über den abschließenden Entwicklungsstand der Prototypen gezogen und fehlende sowie mangelhaft unterstützte Funktionalitäten angesprochen werden.

5.1 Ausführungsumgebung

Für die Realisierung des in Kapitel 4 vorgestellten Konzeptes der kontextabhängigen, eigenverantwortlichen Migration, wurde das *Jadex*-Rahmenwerk (engl. *framework*) ausgewählt. Jadex ist ein an der Universität Hamburg entwickeltes Rahmenwerk für BDI-Agenten[1], welches die Programmierung und Ausführung intelligenter Software-Agenten erlaubt und unterstützt [Braubach 2007, Pokahr 2007]. Vielfältige objektive Gründe sprechen für diese Wahl, unter anderem:

- Der Quelltext dieses Rahmenwerkes ist frei verfügbar, unter der *LGPL*-Lizenz[2] veröffentlicht und wird fortlaufend weiterentwickelt.
- Die *Jadex Reasoning Engine* kann einerseits als eigenständige Agentenplattform laufen, anderseits aber auch als eine Abstraktionsschicht (als *rational agent layer* bezeichnet) zur Ausführung rational handelnder Agenten auf anderen Agentenplattformen, z.B. *JADE* [Bellifemine u. a. 2007] oder *DIET* [Marrow u. a. 2001], genutzt werden. In [Harbeck 2004] wurde außerdem gezeigt, dass Jadex grundsätzlich auch auf mobilen Geräten lauffähig ist.
- Das Rahmenwerk wurde in der Programmiersprache *Java* entwickelt und unterstützt eine Reihe von Standards, die das System grundsätzlich interoperabel mit anderen Technologien machen.

[1] Das *Belief-Desire-Intention*-Modell (BDI) nach [Bratman 1987] ist ein aus der Philosophie stammendes Modell zur Erklärung rationaler Denkweisen. Dieses Modell wird auch Grundlage für den Entwurf rational handelnder Agenten verwendet, welche auf Basis ihres Wissens (Beliefs) eigene Ziele bzw. Wünsche (Desires) verfolgen und diese mit Hilfe von Plänen (Intentions) zu erreichen versuchen.

[2] Die *GNU Lesser General Public License* (LGPL) ist eine von der *Free Software Foundation* entwickelte freie Lizenz. Eine derartig lizenzierte Software darf frei genutzt, weitergegeben und verändert werden. Detaillierte Informationen hierzu finden sich in [Free Software Foundation 2007].

Vor Beginn der eigentlichen prototypischen Implementation des in dieser Arbeit vorgestellten Konzeptes wurde die Jadex Agentenplattform in der Version 0.96beta1 für die Verwendung auf mobilen Geräten angepasst. Im Zuge dessen wurde, aufbauend auf der Arbeit von [Harbeck 2004], insbesondere die sogenannte *Standalone*-Plattform von Jadex in die J2ME[3]-Umgebung portiert. Bei der im Folgenden beschriebenen Implementation eines Prototyps wurden daher die Beschränkungen dieser Zielumgebung berücksichtigt, sodass auch der Prototyp auf mobilen Geräten, die das *J2ME Personal Profile 1.0* [Sun 2007b] unterstützen, lauffähig ist. Dies umfasst weiterhin die Lauffähigkeit auf allen von Jadex unterstützten Agentenplattformen (z.B. JADE und DIET) mit Einschränkungen in der J2EE[4]-Umgebung.

5.2 Resource Awareness-Komponente

Die Aufgabe der Resource Awareness-Komponente (RA-Komponente) ist das Sammeln und Verbreiten von Umweltinformationen (vgl. Abschnitt 4.5). Zu diesen Informationen gehören die Beschreibungen der in der Umwelt vorkommenden Entitäten sowie die Beschreibungen der aufgetretenen Ereignisse. Diese Informationen müssen in ein Modell der Umwelt integriert und als eine observierbare und manipulierbare Zugriffsstruktur repräsentiert werden. In Abschnitt 4.5.1 wurde bereits die Organisationsform dieser Zugriffsstruktur vorgestellt, die auch vom Prototypen implementiert wird. Hiernach sind die Umweltinformationen in Form von Nachrichtenkanälen organisiert, die einzeln und unter Verwendung von Anfragesprachen abonniert werden können.

Um Informationen zu sammeln, werden Discovery-Verfahren (vgl. Abschnitte 2.3 und 2.4) eingesetzt, welche mittels spezifischer Protokolle den Austausch von Informationen zwischen Anbietern und Interessenten regeln. Die RA-Komponente muss die von den Discovery-Verfahren erhaltenen Informationen einerseits in ihr Umweltmodell integrieren und andererseits an andere Interessenten (z.B. entfernte RA-Komponenten oder lokale Agenten) über ihre Nachrichtenkanäle weiterleiten. Interessenten müssen sich hierfür bei der RA-Komponente registrieren und werden anschließend fortlaufend mit entsprechenden Informationen versorgt.

5.2.1 Anforderungen

In Abschnitt 4.5 wurden bereits die wesentlichen Aufgaben der RA-Komponente sowie die damit einhergehenden allgemeinen Anforderungen erarbeitet. Auf dieser Grundlage aufbauend, sollen im Folgenden nun die konkreten Anforderungen an den Prototypen und dessen Funktionalität vorgestellt werden. Im Einzelnen soll die RA-Komponente Folgendes leisten:

Bereitstellung umfangreicher System- und Umgebungsinformationen Um allgemein Entscheidungen und insbesondere Migrationsentscheidungen treffen zu können, bedarf es möglichst detaillierter Informationen über die Domäne bzw. deren Zustand. Die Wahrnehmung der Ressourcen in der Umwelt eines Agenten soll es diesem erlauben, den aktuellen Zustand der Umwelt bei der Auswahl seiner Aktionen zu berücksichtigen. Die Menge der Aktionen, die ein Agent in seiner Umwelt ausführen kann (z.B. das Versenden von Nachrichten, das Migrieren von einer Ausführungsumgebung in eine andere) ist zwar begrenzt, aber die Entscheidung für die Auswahl einer Aktion kann vielfältige Gründe haben. Da diese Gründe je nach Einsatzkontext individuell verschieden und schwerlich allgemein vorab

[3] *Java 2 Micro Edition*, speziell das *Personal Profile 1.0*
[4] *Java 2 Enterprise Edition*

bestimmt werden können, soll zunächst eine möglichst breite Auswahl auszutauschender System- und Umweltinformationen getroffen werden. In Abschnitt 4.4 wurden bereits Informationen von allgemeinem Interesse identifiziert. Zusätzlich soll es jedoch möglich sein, den Fundus an angebotener und nachzufragender Informationen nach Belieben zu erweitern, sodass auch anwendungsabhängige Informationen in das Umweltmodell integriert werden können.

Unterstützung hybrider Netzwerke Wie bereits in Kapitel 2 festgestellt, unterstützen existierende Discovery-Verfahren in den meisten Fällen lediglich entweder Infrastruktur- oder ad-hoc Netzwerke[5]. Denn die unterschiedlichen Anforderungen, die sich aus den spezifischen Eigenschaften dieser Netzwerkarten ergeben, erfordern speziell angepasste Protokolle und Mechanismen. Dies schränkt jedoch die Kooperationsmöglichkeiten zwischen Geräten in unterschiedlichen Netzwerkarten ein. Der zu entwickelnde Prototyp soll diese Einschränkung umgehen, indem er unter anderem die Verwendung unterschiedlicher Discovery-Verfahren erlauben und auszutauschende Informationen in verschiedenen Repräsentationsformen unterstützen soll. Die Art und Weise wie Umweltinformationen zwischen einzelnen Endgeräten ausgetauscht werden und die damit einhergehenden Anforderungen an die Hardware-Ausstattung der Geräte und die Kommunikationsinfrastruktur des Netzwerkes hängen auf diese Weise ausschließlich von dem Anwendungskontext und den Ansprüchen des Benutzers ab. Zusätzlich soll der Prototyp selbständig seine Erreichbarkeit überwachen und mögliche Adressänderungen (z.B. bei Neueintritt in ein ad-hoc Netzwerk) seinen Kontakten mitteilen können, damit die Konnektivität nach Möglichkeit erhalten bleibt.

Unterstützung unterschiedlicher Discovery-Verfahren Die Diskussion in den Abschnitten 2.3.7 und 2.4.7 hat aufgezeigt, dass die Wahl eines geeigneten Discovery-Verfahrens für einen bestimmten Einsatzkontext von einer Vielzahl an Kriterien abhängt. Ein optimales Verfahren für heterogene Umgebungen gibt es nicht. Aus diesem Grund und wegen der bereits weiten Verbreitung einiger Discovery-Verfahren, soll die Architektur des Prototyps offen für etwaige Protokolle sein. Zwar soll ein eigenes, einfaches und proprietäres Basisprotokoll (vgl. Abschnitt 4.5.5) implementiert werden, um die grundlegende Interoperabilität zwischen Geräten sicherzustellen, aber die Option, weitere Verfahren zu adaptieren, soll grundsätzlich durch Entwicklung entsprechender Adapterkomponenten möglich sein.

Unterstützung unterschiedlicher Beschreibungs- und Abfragesprachen Mit der potenziellen Erweiterbarkeit durch unterschiedliche Discovery-Verfahren geht die Unterstützung verschiedener Informationsrepräsentationen einher. Ein Teil der betrachteten Verfahren tauscht Informationen in proprietären Formaten aus, andere verwenden einen der vielen Repräsentationsstandards. Die Wahl eines geeigneten Formates ist durch den Einsatzkontext und die Fähigkeiten des Gerätes bestimmt. Informationen sollen allerdings nicht einfach nur repräsentiert, sondern auch gezielt angefragt und gefiltert werden können, um zum Beispiel den Ressourcenverbrauch zu reduzieren. Daraus erwächst die Anforderung an den Prototypen, dass dieser möglichst vielfältige Repräsentationen und Anfragesprachen verarbeiten können soll.

Unterstützung mobiler Geräte Die Energie-, Hard- und Software-Ressourcen mobiler Geräte sind im Vergleich zu stationären Geräten beschränkt. Abgesehen von dem Ressourcenverbrauch (Netzwerk, Prozessor, Speicher) der RA-Komponente zur Laufzeit, steht deren ge-

[5] Eine Ausnahme hiervon bildet zum Beispiel das in Abschnitt 2.4.5 vorgestellte *Scalable Service Discovery*-Verfahren.

nereller Einsatz auf mobilen Geräten im Vordergrund. Da der Prototyp, wie auch bereits die Jadex-Agentenplattform, in der Programmiersprache Java entwickelt werden soll, müssen die Einschränkungen der Java-Laufzeitumgebung für mobile Systeme berücksichtigt werden. Das bedeutet, bei der Implementation darf nur eine vorgegebene Teilmenge der normalen Java-Bibliotheken verwendet werden, welche durch die Spezifikation des *J2ME Personal Profile 1.0* [Sun 2007b] festgelegt ist.

Hohes Maß an Konfigurierbarkeit Da die RA-Komponente in heterogenen Umgebungen eingesetzt werden und vielfältige Möglichkeiten zur Erweiterung bieten soll, ist ein hohes Maß an individueller Konfigurierbarkeit erforderlich. Die wesentlichen Mechanismen zur Verarbeitung und Verbreitung von Informationen sollen von dem Benutzer nach eigenen Ansprüchen und ohne Eingriff in den Programmcode einzustellen sein. Auch soll die Möglichkeit vorgesehen werden, dass die RA-Komponente eigenmächtig Einstellungen an der Konfiguration vornehmen kann, um sich so dynamisch und ohne Benutzerinteraktion optimal an verändernde Umweltbedingungen anzupassen.

Einfache Benutzbarkeit Der Einsatz der RA-Komponente seitens des Benutzers hängt nicht zuletzt davon ab, ob der Benutzer von den Diensten der Komponente bei seiner Arbeit unterstützt wird und die Verwendung keinen Mehraufwand für ihn bedeutet. Ein anfänglich hoher Konfigurationsaufwand soll durch Verwendung parametrisierbarer Profile vermieden werden, sodass die Einstiegshürde für normale Benutzer möglichst tief angesetzt wird und diese, ggf. unter suboptimaler Ausnutzung der Kapazitäten, von den gebotenen Funktionen profitieren. Versierte Benutzer hingegen sollen weiterhin die Möglichkeit haben, wie bereits vorangehend erwähnt, die Komponente sowie die unterstützten Discovery-Verfahren und Repräsentationsformen im Detail zu konfigurieren.

Einfache und transparente Integration in bestehende Systemumgebung Da die Wahrnehmung der Umwelt keinen Selbstzweck darstellt, müssen die Ausführungsumgebung bzw. die Agenten möglichst einfach und, sofern möglich, transparent auf die Umweltinformationen zugreifen können. Die hierfür notwendigen Schnittstellen des Umweltmodells müssen daher einerseits einfach, andererseits aber auch komfortabel angesprochen werden können. Um einen transparenten Zugriff zu ermöglichen, könnten zum Beispiel die *gelben Seiten* einer Agentenplattform (auch *Directory Facilitator* genannt), Informationen über entfernt angebotene Dienste aufnehmen und so den lokal laufenden Agenten auf Anfrage zur Verfügung stellen. Im Rahmen dessen spricht man auch von einer Föderation der Verzeichnisse, die einen wechselseitigen Austausch von Dienstinformationen erfordert.

Die oben beschriebenen Anforderungen sollen durch eine prototypische Implementation als Agent für die Jadex-Agentenplattform umgesetzt werden. Die folgenden Abschnitte beschreiben die Architektur der Komponente und greifen die wichtigsten Merkmale dieser Umsetzung heraus.

5.2.2 Realisierung als Agent

Es gibt unterschiedliche Möglichkeiten, die RA-Komponente zu realisieren und in einem Rahmenwerk zugreifbar zu machen. In [Satyanarayanan u. a. 1995] wird die Stellung bezogen, dass das Betriebssystem eines Gerätes die Umwelt wahrnehmen und die Informationen für lokale Anwendungen zugreifbar machen soll. Auf diese Weise können bestimmte Daten, wie zum Beispiel

die Signalqualität bei Funkverbindungen oder die Auslastung des Prozessors, bereits in den unteren Schichten des Betriebssystems erhoben werden. Anwendungen, die Interesse an derartigen Informationen haben, kommunizieren direkt mit entsprechenden Bibliotheken des Betriebssystems. Der Vorteil dieses Ansatzes liegt zum einen darin, dass jedwede Art von Anwendung sowie auch das Betriebssystem selbst diese Informationen nutzen kann, zum anderen sind bestimmte Informationen, z.B. die Auslastung der Hardware (z.B. Prozessor, Netzwerkschnittstellen etc.), nur auf Ebene des Betriebssystems zugreifbar und nicht zum Beispiel in der virtuellen Maschine von Java. Der größte Nachteil hingegen ist, dass in heterogenen Umgebungen eine Vielzahl unterschiedlicher Betriebssysteme, angepasst auf die jeweiligen Geräteklassen, eingesetzt werden und der Prototyp somit zwangsläufig für verschiedene Systeme umgesetzt werden müsste.

Ein ähnlicher Ansatz wird in [Chen und Kotz 2000] beschrieben. In diesem Ansatz wird eine Middleware-Schicht zwischen dem Betriebssystem und den Anwendungen entworfen, deren Aufgabe die Wahrnehmung des Kontextes bzw. der Umwelt ist. Es wird argumentiert, dass hierdurch der Entwicklungsaufwand reduziert und ein entsprechendes System auch zur Verwendung für andere Anwendungen generalisiert werden kann. Für den im Rahmen dieser Arbeit zu entwickelnden Prototypen hieße das, dass dieser beispielsweise als Komponente bzw. Dienst einer Agentenplattform realisiert werden könnte, wie es auch in [Pirker u. a. 2004] vorgeschlagen wird. Da die Agentenplattform selbst eine Middleware darstellt, also eine Ebene zwischen dem Betriebssystem und den Anwendungen, könnten somit alle Anwendungen, die auf der Agentenplattform laufen, auf die RA-Komponente bzw. den RA-Dienst zugreifen. Außerdem kann der Vorgang des Auffindens von Ressourcen als Teil des Umfeldes eines Agenten angesehen werden und sollte daher, genau wie das in der abstrakten Architektur von FIPA spezifizierte Nachrichtentransportsystem [FIPA 2002a], als Komponente der Plattform realisiert werden. Jedoch würde die Umsetzung als Dienst, Änderungen bzw. Erweiterungen der Implementation der Plattform nach sich ziehen, um den Dienst über Programmierschnittstellen aufrufen zu können [Lawrence 2002a].

Um die Entwicklung des Prototypen von der fortlaufenden Weiterentwicklung des Jadex-Rahmenwerkes weitestgehend zu entkoppeln, wird der Prototyp daher auf Anwendungsebene als Agent implementiert. Dies ist auch im Gegensatz zu der Modellierung als Plattformkomponente ein natürlicheres Modell einer Entität, die Strategien für die Bearbeitung von Aufgaben besitzt [Lawrence 2002a]. Hierdurch kann die RA-Komponente die anderen von der Plattform angebotenen Dienste, zum Beispiel das Versenden von Nachrichten oder Deliberationsmechanismen, in Anspruch nehmen, kann eigenständig entwickelt und getestet werden und ist darüber hinaus auch prinzipiell auf den von Jadex unterstützten Agentenplattformen *JADE* und *DIET* verwendbar. Der Nachteile dieses Ansatzes ist jedoch, dass sowohl lokale als auch entfernte Agenten mit der RA-Komponente über den Austausch von Nachrichten anstatt über direkte Methodenaufrufe kommunizieren müssen, was zum einen weniger performant und zum anderen weniger komfortabel ist.

5.2.3 Architektur

In Anlehnung an den Bezeichner für die *gelben Seiten* einer Agentenplattform, dem *Directory Facilitator*, wird die RA-Komponente im Folgenden als *Resource Facilitator*-Agent (RF) bezeichnet (engl. *facilitator*, zu dt. Moderator). Der RF hat im Wesentlichen die Aufgabe, lokal vorhandene Ressourceninformationen zu erfassen, registrierte Discovery-Verfahren auszuführen, zwischen verschiedenen Repräsentationsformaten zu übersetzen, Kontakte und mit diesen verknüpfte Um-

weltinformationen zu verwalten und die Gesamtheit der vorhandenen Informationen der Ausführungsumgebung bzw. den registrierten Interessenten zur Verfügung zu stellen. Die wichtigsten Komponenten des Resource Facilitator sowie deren Zusammenspiel sind schematisch in Abbildung 5.1 dargestellt. Die Architektur ergab sich aus der Identifikation der wichtigsten Objekte, die innerhalb des Systems zwischen den einzelnen Komponenten ausgetauscht werden. Die einzelnen Teilkomponenten werden im Folgenden näher erläutert.

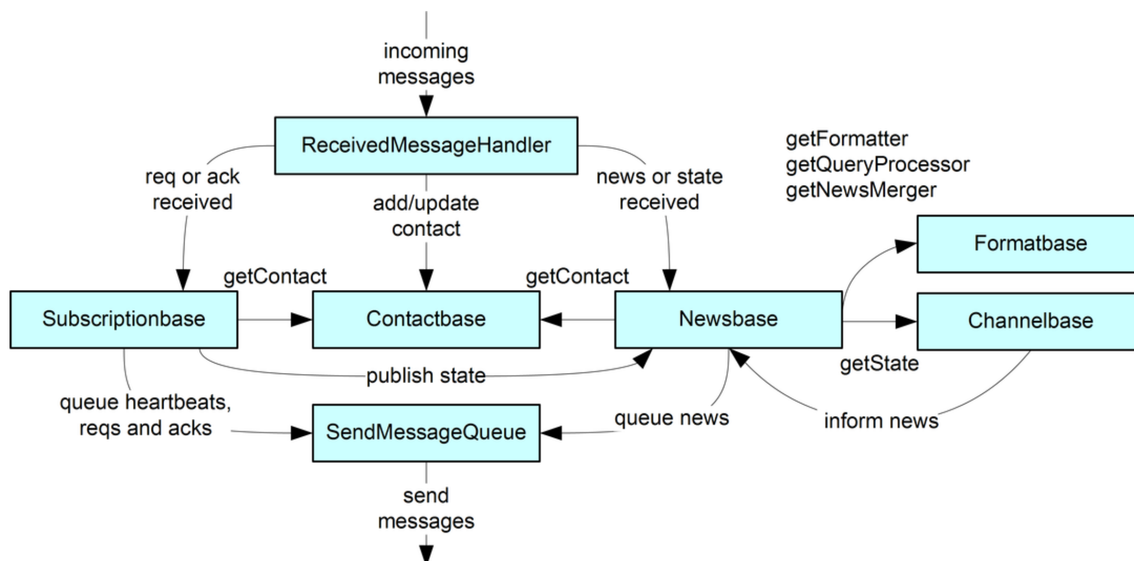

Abbildung 5.1: RF: Zusammenspiel der Komponenten (eigene Darstellung)

ReceivedMessageHandler An diese Komponente werden alle an den RF adressierten Nachrichten übergeben. Die Inhalte einer Nachricht werden extrahiert und an die jeweils zuständigen Komponenten weitergeleitet. Inhalte können zum Beispiel sogenannte *SubscriptionRequests*, mit Hilfe derer man bestimmte Nachrichtenkanäle abonnieren kann, oder *SubscriptionAcknowledgements* sein, die ein Abonnement als bestätigt bzw. zurückgewiesen auszeichnen. Diese werde an die sogenannte *Subscriptionbase* weitergeleitet. Informationen über Zustandsbeschreibungen eines Kanals (*ChannelState*) oder Meldungen über Zustandsänderungen (*ChannelNews*) hingegen werden an die sogenannte *Newsbase* übergeben, wo sie ihren jeweiligen Urhebern zugeordnet werden. Jede ankommende Nachricht enthält darüber hinaus die aktuellen Kontaktdaten des Senders, welche mit den lokal gespeicherten Kontaktdaten aus der *Contactbase* abgeglichen werden, um auf die Dynamik der Umgebung reagieren zu können.

Contactbase In der Contactbase werden die Kontakte eines RFs gespeichert und verwaltet. Als Kontakt bezeichnet man einen entfernten RF, den man bei Bedarf zu kontaktieren versuchen kann. Jeder entfernte RF, mit dem Informationen ausgetauscht werden, wird als Kontakt in der Contactbase vermerkt (und auch persistent gemacht). Zusätzlich kann ein RF andere RFs nach deren Kontakten befragen und erhält von diesen alle notwendigen Adressdaten um selbst eine Verbindung zu dem jeweiligen Kontakt aufzubauen. Außerdem werden in der Contactbase neben den reinen Kontaktdaten auch alle Informationen, die man zur Laufzeit von diesem Kontakt erhalten hat (ChannelStates und -News) sowie Statistiken über den Nachrichtenaustausch gespeichert, sodass jederzeit der aktuelle Zustand dieses Kontaktes festgestellt werden kann.

Subscriptionbase Die Subscriptionbase bekommt von dem ReceivedMessageHandler die empfangenen SubscriptionRequests (Abonnement-Aufforderung) und SubscriptionAcknowledgements (Abonnement-Bestätigungen) übergeben. Wird eine Aufforderung übergeben, so entscheidet die Subscriptionbase anhand festgelegter Regeln, ob diese positiv oder negativ bestätigt werden soll und veranlasst ggf. die Newsbase, den aktuellen Zustand des abonnierten Kanals an den Anfragenden zu übermitteln. Bei einer übergebenen Bestätigung führt die Subscriptionbase einen Abgleich mit den versendeten Aufforderungen durch, aktualisiert ggf. Abonnement-Parameter und vermerkt den Status des Abonnements für den entsprechenden Kontakt.

Newsbase Die Zustände und Neuigkeiten aller Nachrichtenkanälen werden in der Newsbase vorgehalten. Empfängt die Newsbase Neuigkeiten, seien es Neuigkeiten lokaler Kanäle von der *Channelbase* oder Neuigkeiten abonnierter Kanäle über den ReceivedMessageHandler, durchlaufen diese unter Umständen mehrere Verarbeitungsschritte in denen sie zusammengefasst, gefiltert und ggf. in andere Repräsentationen überführt werden. Im Fall lokal aufgetretener Kanalneuigkeiten werden diese an die *SendMessageQueue* weitergeleitet, welche die Informationen an registrierte Abonnenten verschickt. Andernfalls wird aus der Contactbase der Urheber der empfangenen Nachricht abgefragt und dessen entsprechender Kanalzustand mit den empfangenen Neuigkeiten zusammengeführt.

Channelbase An der Channelbase registrierten sich alle lokalen Nachrichtenkanäle und offerieren Informationen, welche von Interessenten angefordert werden können. Ein Nachrichtenkanal besteht aus einem *ChannelInformationProvider*, einem *ChannelInformationHandler* und kanalspezifischen Datenobjekten, die den Zustand bzw. Neuigkeiten des Kanals repräsentieren. Die ChannelInformationProvider liefern Informationen über lokale Ressourcen, zum Beispiel die lokal laufenden Agenten oder die lokal bekannten Kontakte. Die ChannelInformationHandler hingegen verarbeiten Informationen, die über abonnierte Nachrichtenkanäle eingetroffen sind, melden also zum Beispiel neu angebotene Dienste eines entfernten Systems an registrierte Interessenten.

Formatbase Die Formatbase verwaltet die für einzelne Kanäle registrierten Repräsentationsformen. Um Zustandsinformationen und Neuigkeiten an Abonnenten zu versenden bzw. empfangene Informationen zu verarbeiten, muss für jeden Kanal mindestens eine Repräsentationsform registriert sein. Zu einer Repräsentationsform gehört ein *Formatter*, der kanalspezifische Informationsobjekte in eine allgemeine Repräsentation überführt, ein *QueryProcessor*, der Anfragen (engl. *queries*) auf der allgemeinen Repräsentation ausführen kann und somit speziell angefragte Informationen der einzelnen Abonnenten filtert und ein *NewsMerger*, der zwei verschiedene Neuigkeiten zusammenführen und redundante Informationen entfernen kann.

SendMessageQueue Alle zu versendenden Nachrichtenobjekte werden vor ihrem Versand in eine Warteschlange eingereiht. Auf diese Weise können SubscriptionRequests, -Acknowledgements, Kanalinformationen und Heartbeats für jeden einzelnen Empfänger gebündelt und als eine einzige Nachricht versandt werden. Dieser Mechanismus bietet die Möglichkeit das allgemeine Nachrichtenaufkommen durch Vermeidung redundanter Informationen bei dem Versand von Nachrichten zu reduzieren und kann dem Verlust von Nachrichten während kurzer offline-Zeiten (vor allem z.B. in ad-hoc Netzen) vorbeugen.

Discoverybase Die Discoverybase ist nicht in dem Diagramm enthalten, hat jedoch ähnliche Bezüge zu anderen Komponenten wie der ReceivedMessageHandler. Die Discoverybase enthält Informationen zu den einzelnen Discovery-Verfahren, initialisiert und startet diese. Der Prototyp enthält lediglich grundlegende Funktionen zum Verbreiten von Umweltinformationen, die an die Methoden von JXTA (vgl. Abschnitt 2.3.6) angelehnt sind. Hierzu gehören das Auffinden von Ressourcen in lokalen Netzwerken über Uni- und Multicast-Suche sowie das Anfragen bei bereits bekannten Kontakten in Weitverkehrsnetzen.

5.2.4 Funktionsweise

Im Folgenden soll die Funktionsweise der internen Mechanismen des Prototypen angedeutet werden. An dieser Stelle können jedoch nur exemplarisch einige Aspekte, die von besonderem Interesse sind, betrachtet werden. Eine komplette Anleitung würde den Rahmen dieser Arbeit übersteigen und es sei auf die weiterführende Dokumentation des Prototypen sowie dessen Programmcode verwiesen.

Abonnieren eines Nachrichtenkanals

Um Informationen über die Umwelt zu erhalten oder um bei Eintreten bestimmter Ereignisse benachrichtigt zu werden, muss ein Agent entsprechende Nachrichtenkanäle abonnieren. Hierzu verschickt er eine Aufforderung an den anbietenden Resource Facilitator (RF). Die Aufforderung muss die Kontaktinformationen des Agenten, den Namen aller zu abonnierenden Kanäle sowie für jeden Kanal optional weitere Parameter enthalten. Über diese Parameter können zum Beispiel Anfragen (engl. *queries*) für spezifische Kanalinformationen formuliert, etwaige Beschränkungen (engl. *constraints*) der Ergebnismenge sowie unterstützte Repräsentationsformate vorgegeben werden.

Bei der Abonnierung eines Kanals werden zwei Fälle unterschieden: 1) der Abonnement ist ein RF-Agent und 2) der Abonnement ist ein normaler Agent. Abonniert ein RF-Agent einen Nachrichtenkanal eines anderen, entfernten RF-Agenten, so gehen diese beiden Agenten eine Kontaktbeziehung miteinander ein. Das bedeutet, sie können gegenseitig Kanalinformationen abonnieren. In dem zweiten Fall verhält es sich anders. Mit einem normalen Agenten geht ein RF-Agent keine Kontaktbeziehung, sondern lediglich eine einfache Abonnement-Beziehung ein. Dies führt dazu, dass die Kontaktdaten des Agenten nicht persistent gemacht werden und der RF-Agent nicht versucht, Nachrichtenkanäle dieses normalen Agenten zu abonnieren. Außerdem kommuniziert ein normaler Agent im Regelfall mit dem lokal auf der Plattform laufenden RF-Agenten. Ein Abonnement von Nachrichtenkanälen entfernter RF-Agenten ist zwar möglich, sollte jedoch dem lokalen RF vorbehalten bleiben, damit die ausgetauschten Informationen allen Agenten auf der Plattform zugänglich gemacht werden können.

Die Kommunikation, die bei der Abonnierung eines Kanals stattfindet, ist in Abbildung 5.2 dargestellt, sie unterscheidet nicht zwischen beiden oben genannten Fällen. Im ersten Schritt schickt der Agent eine Abonnement-Aufforderung (engl. *subscription request*) für einen bestimmten Kanal sowie seine eigenen Kontaktdaten und optional ein oder mehrere Abonnement-Parameter an den RF-Agenten (Schritt 1). Der RF-Agent entscheidet, ob er die Aufforderung annehmen will, kann die Abonnement-Parameter ändern (z.B. den Benachrichtigungszeitraum erhöhen um Ressourcen zu sparen) und schickt schließlich eine Empfangsbestätigung (engl. *subscription acknowledgement*) zurück an den Abonnenten (Schritt 2). Diese Empfangsbestätigung enthält den Status des Abonnements (z.B. ob die Aufforderung angenommen oder verweigert wurde, ob der Agent

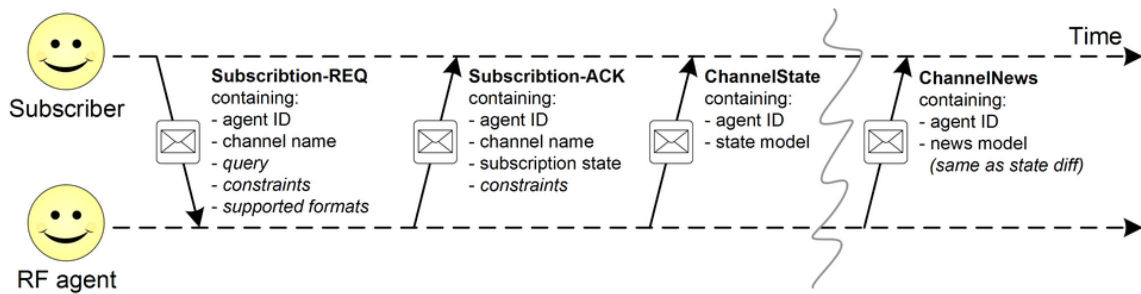

Abbildung 5.2: RF: Abonnieren eines Nachrichtenkanals (eigene Darstellung)

bereits für diesen Kanal registriert war etc.), unter Umständen eine Menge von Abonnement-Parametern, die von dem Anbieter vorgegeben werden bzw. überschrieben werden, sowie die aktuellen Kontaktdaten des RF-Agenten.

Wurde das Abonnement angenommen, versendet der RF-Agent zusätzlich den aktuellen Zustand des abonnierten Nachrichtenkanals (Schritt 3), zum Beispiel alle aktuell ausgeführten Agenten im Falle des *Agents*-Kanals oder eine Liste derzeit angebotener Dienste im Falle des *Services*-Kanals. Von diesem Zeitpunkt an, schickt der RF-Agent, sobald ein Ereignis eintritt, welches den Zustand des Kanals ändert, nur noch die Änderungen an den Abonnenten (Schritt 4), zum Beispiel den Namen eines gerade gestarteten Agenten oder Informationen über einen neuen Dienst. Es obliegt dann dem Agenten diese Änderungen mit den Zustandsinformationen des entsprechenden Kanals lokal zusammenzuführen. Ein Abonnement gilt dann als gekündigt, wenn der RF-Agent eine entsprechende Kündigungsnachricht erhält, bei der Zustellung einer beliebigen Nachricht an den Abonnenten ein Fehler auftritt (z.B. wenn der Agent nicht mehr erreichbar ist) oder eine festgelegte Anzahl erwarteter Heartbeats nicht empfangen wird. Auf diese Weise können Abonnements, die nicht mehr aktuell sind, aus dem System entfernt werden.

Um eine Übersicht über alle Abonnenten zu haben, verwaltet ein RF-Agent diese in einer internen Ordnungsstruktur, die als *Subscriptionbase* bezeichnet wird. In dieser Struktur sind die Kontaktdaten aller Abonnenten samt ihrer Abonnement-Parameter, nach Kanälen geordnet, enthalten. Ist der Abonnent ein anderer RF-Agent, so wird dieser zusätzlich in einer separaten Ordnungsstruktur, der *Contactbase*, vermerkt. Um bekannte Kontakte nicht wieder zu vergessen, wird diese Struktur beim Beenden des RF-Agenten persistent gemacht und beim nächsten Start erneut geladen. Nach einem Neustart hat ein RF-Agent initial keine Verbindung zu anderen RF-Agenten und entsprechend eine eingeschränkte Sicht auf die Umwelt und auf verfügbare Ressourcen. Er kann in diesem Fall jedoch einzelnen Kontakten seiner initialen Contactbase proaktiv Abonnement-Aufforderungen schicken, um so sein Umweltmodell zu aktualisieren. Um eine Auswahl der zu adressierenden Kontakte zu treffen, kann er sich an Metadaten der vergangenen Kommunikation orientieren, die z.B. die Häufigkeit ausgetauschter Nachrichten, den Zeitpunkt der letzten Kommunikation sowie die Anzahl aufgetretener Fehler beim Nachrichtenversand dokumentieren.

Um die Kommunikation eines Agenten mit einem RF-Agenten für den Programmierer so einfach wie möglich zu machen, wurde eine Komponente (eine sogenannte *Capability*) für Jadex implementiert, die von den Details des Abonnement-Vorgangs abstrahiert. Der Programmierer muss diese Komponente lediglich in die Beschreibung seines Agenten (*Agent Definition File*) einbinden und angeben, welcher Plan die Kanalinformationen handhaben soll. Dann kann durch Aktivieren vordefinierte Ziele (engl. *goals*) einen Abonnement-Vorgang angestoßen werden.

Kommunikation von Ereignissen

Beim Auftreten eines Ereignisses gilt es, den geänderten Zustand der Umwelt an registrierte Interessenten zu übermitteln, damit diese ihrerseits ihr Umweltmodell aktualisieren können. Wenn ein solches Ereignis in einer Ausführungsumgebung auftritt, muss dieses daher von speziellen Sensoren wahrgenommen und zur weiteren Bearbeitung an entsprechende Teilkomponenten des RF weitergeleitet werden. Derartige Sensoren sind spezifisch für jeden Nachrichtenkanal und werden als *ChannelInformationProvider* (CIP) bezeichnet. Der Prototyp implementiert bereits eine Reihe solcher CIPs, zum Beispiel für den *Agents*-Kanals, welcher Informationen des *Agent Management System* (AMS) abfragt, und für den *Services*-Kanal, der mit dem lokalen *Directory Facilitator* (DF) interagiert.

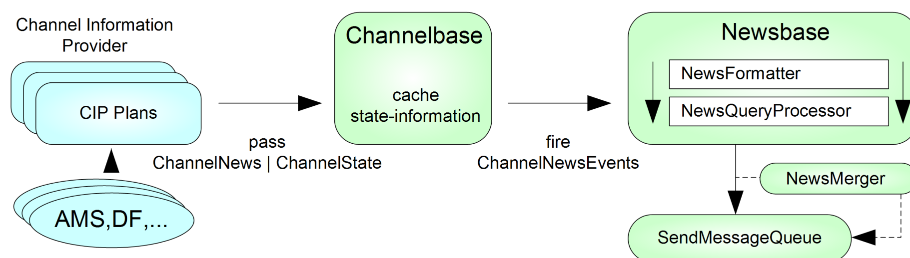

Abbildung 5.3: RF: Kommunikation von Ereignissen (eigene Darstellung)

Beim Starten eines solchen CIP liest dieser den initialen Zustand des Kanals aus und übergibt ihn an die Channelbase, welche den Kanalzustand zwischenspeichert. Im Falle des AMS-CIP wäre dies zum Beispiel eine Liste aller aktuell auf der Plattform laufenden Agenten. Tritt nun ein Ereignis auf, zum Beispiel das Starten eines neuen Agenten, so wird sowohl der neue Kanalzustand als auch die Differenz zum vorhergehenden Zustand (auch als Neuigkeit bezeichnet) an die Channelbase übergeben und von dort an die Newsbase weitergeleitet. Die Newsbase fragt daraufhin bei der Contactbase alle Kontakte ab, die Informationen des entsprechenden Kanals abonniert haben und beginnt die Neuigkeiten zu verarbeiten.

Da ein Abonnement Angaben zu bevorzugten Repräsentationsformaten seitens des Abonnenten enthalten kann, müssen in einem ersten Verarbeitungsschritt die Kanalinformationen in entsprechende Repräsentationen, beispielsweise in das RDF-Format (vgl. Abschnitt 2.5.1), überführt werden. Falls ein Abonnent eine spezifische Anfrage (engl. *queries*) im Format der entsprechenden Repräsentationsform angegeben hat, wird diese in einem nächsten Schritt auf den Daten ausgeführt und ein individuelles Ergebnisobjekt erstellt, welches die gefilterten Informationen enthält.

Schließlich wird im Nachrichtenausgang (der *SendMessageQueue*) für jeden einzelnen Empfänger überprüft, ob bereits Nachrichten an diesen für den Versand vorliegen. Ist dies nicht der Fall, kann das Ergebnisobjekt, in einer Nachricht verpackt, direkt an den Nachrichtenausgang für den späteren Versand übergeben werden. Im anderen Fall jedoch überprüft die Newsbase, ob die bereits zum Versand vorliegende Nachricht ältere Informationen desselben Kanals enthält. In diesem Fall werden in einem dritten Verarbeitungsschritt die alten und die neuen Informationen durch den *NewsMerger* zusammengeführt und erst dann wiederholt in den Nachrichtenausgang eingereiht. Dieser letzte Schritt kann jedoch von einem Abonnenten durch Angabe entsprechender Parameter in der Abonnement-Aufforderung unterdrückt werden. Der Sinn dieses Verarbeitungsschrittes ist das Eliminieren redundanter oder veralteter Informationen, zum Beispiel wenn ein neuer Agent gestartet und sofort wieder beendet wurde.

Auffinden von Kontakten

Jeder RF-Agent kennt die lokal in seiner Ausführungsumgebung verfügbaren Ressourcen. Um jedoch Informationen über andere Ressourcen seiner Umwelt zu erhalten, muss er sich mit entfernten RF-Agenten über die ihnen bekannten Ressourcen verständigen. Hierfür ist es initial notwendig, andere RF-Agenten entfernter Agentenplattformen im Netzwerk zu finden. In Abschnitt 4.5 wurden bereits mögliche Erstkontaktmechanismen vorgestellt, deren Anwendung anhand eines Beispielszenarios im Folgenden beschrieben werden soll.

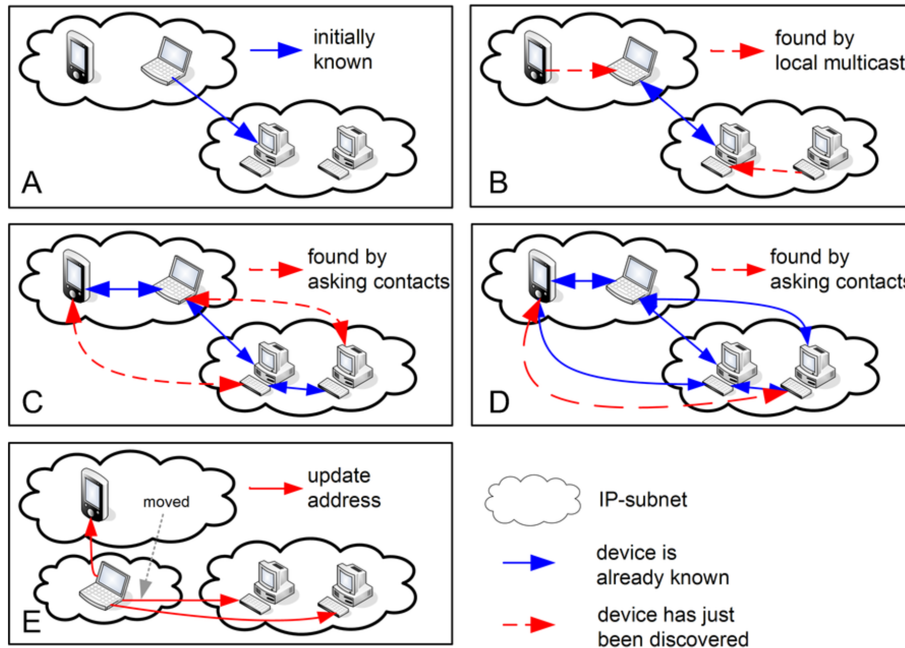

Abbildung 5.4: RF: Auffinden von Kontakten (eigene Darstellung)

In diesem Szenario, dargestellt in Abbildung 5.4, befinden sich vier Geräte in zwei unterschiedlichen IP-Subnetzen. Auf allen Geräten laufen Agentenplattformen mit jeweils einem RF-Agenten. Das Ziel ist, dass sich alle RF-Agenten untereinander kennen um Informationen austauschen zu können. Da es keine zentrale Registrierung für RF-Agenten gibt und diese in verschiedenen Subnetzen beheimatet sind[6], ist es nicht möglich die RF-Agenten gänzlich ohne Vorabinformationen miteinander bekannt zu machen. Die einzige Voraussetzung in diesem Szenario ist daher, dass das Notebook des einen Subnetzes die Adresse eines Computers aus dem anderen Subnetz kennt und somit eine Brücke schlagen kann (Abbildung A). In einem nächsten Schritt versendet mindestens ein Gerät aus jedem Subnetz eine Multicast-Kontaktanfrage und findet somit das jeweils andere Gerät des Subnetzes (Abbildung B).

Bis zu diesem Punkt kennt jedes Gerät lediglich seine Nachbarn, die in einer 1-Sprung (engl. *one hop*) Nachrichtendistanz liegen. Durch Abonnement des *Contacts*-Kanals kann ein RF-Agent von seinen Nachbarn Informationen über deren Kontakte anfordern. Mit diesen Daten können RF-Agenten weitere Kontaktnachrichten verschicken, um entfernte RF-Agenten auch über mehrere Sprünge hinweg zu kontaktieren. In Abbildung C ist dies über eine Distanz von zwei und in Abbildung D über eine Entfernung von drei Sprüngen dargestellt. Auf diese Weise haben alle RF-Agenten Kenntnis von jeweils allen anderen RF-Agenten erlangt.

[6]Der Versand von Multicast-Kontaktnachrichten funktioniert technisch bedingt nur innerhalb desselben Subnetzes, weshalb vollautomatische Kontaktmechanismen nur innerhalb eines solchen einsetzbar sind.

In Abbildung E ist der Fall dargestellt, dass sich die Adresse eines Gerätes geändert hat, entweder weil der Benutzer sich mit seinem mobilen Gerät explizit in einem anderen Netzwerk angemeldet oder weil das Gerät nach einem Verbindungsabbruch eine neue Adresse zugewiesen bekommen hat. Um auf einen solchen Fall entsprechend reagieren zu können, überwacht jeder RF-Agent kontinuierlich die Netzwerkschnittstellen des Gerätes um die Änderung von Adressen wahrzunehmen. Hat sich eine Adresse geändert, so versendet er Nachrichten mit den aktualisierten Informationen an alle seine Kontakte, damit das logische Netzwerk zwischen den Kontakten intakt bleibt. Dieser Mechanismus ist insbesondere in ad-hoc Netzwerken von Vorteil, da sich in diesen sowohl die Adressen der Geräte als auch die Topologie des Netzwerkes häufig ändern kann.

5.2.5 Zusammenfassung

Die RA-Komponente ist für die Wahrnehmung der Umwelt, das Erstellen und Aktualisieren eines Umweltmodells sowie für das Verbreiten von Umweltinformationen verantwortlich. Die Anforderungen, die an eine entsprechende Komponente formuliert wurden, zum Beispiel die Bereitstellung umfangreicher System- und Umgebungsinformationen sowie die Unterstützung heterogener Hard- und Software-Infrastrukturen, resultierten in der prototypischen Realisierung dieser Komponente als ein Agent für das Jadex-Rahmenwerk.

Die Architektur dieser Komponente wurde entsprechend den Anforderungen generisch gestaltet und sieht eine Reihe von Schlüsselrollen zur Verwaltung unterschiedlicher Arten von Informationen vor. Aufgrund dieser Architektur sollte es später möglich sein, einzelne Teilkomponenten einfach auszutauschen, um das System insgesamt besser an die weiteren Anforderungen, die sich aus einem praktischen Einsatz ableiten lassen, anzupassen. Das Zusammenspiel dieser Schlüsselrollen untereinander wurde anhand ausgewählter Beispiele exemplarisch beschrieben. Für eine weitergehende Betrachtung der Funktionsweise sei auf die Quellen und die Dokumentation des Prototypen verwiesen.

5.3 Migrationskomponente

Die Migrationskomponente zeichnet sich für alle Belange der Migration verantwortlich. Neben dem Vorgang der Migration selbst, bietet sie darüber hinaus Dienste zur Transaktionskontrolle, zum Anlegen und Reaktivieren von Sicherheitskopien sowie für die Zustellung von Nachrichten an migrierte Agenten. Um dies zu realisieren, bedient sich die Migrationskomponente im Wesentlichen der Plattformdienste sowie Funktionen, die von der Java-Laufzeitumgebung zur Verfügung gestellt werden. Lediglich für die starke Migration und das Anlegen starker Sicherheitskopien wird auf zusätzliche Bibliotheken zurückgegriffen.

Ebenso wie der Resource Facilitator, wird auch der Prototyp der Migrationskomponente als ein Anwendungsagent, im Folgenden auch als *Migration Manager* (MM) bezeichnet, und nicht als ein Dienst der Plattform realisiert (vgl. Abschnitt 5.2.2). Auf diese Weise ist die Entwicklung und der Test des Agenten nicht an die fortschreitende Entwicklung der Plattform gekoppelt. Im folgenden Abschnitt werden Anforderungen an den Prototypen formuliert. Auf diesen basiert die Realisierung des Agenten, dessen Architektur und Funktionen auszugsweise im Anschluss daran vorgestellt werden.

5.3.1 Anforderungen

Die allgemeinen Aufgaben einer Migrationskomponente wurden bereits in Abschnitt 4.7 erarbeitet. Im Folgenden sollen nun darauf aufbauend die konkreten technischen Anforderungen an den zu implementierenden Prototypen der Komponente aufgeführt werden.

Unterstützung verschiedener Arten der Mobilität Um möglichst vielen Ansprüchen gerecht zu werden, sollen sowohl die schwache als auch die starke Mobilität von der Migrationskomponente unterstützt werden. Für die schwache Migration soll es darüber hinaus die Wahl zwischen einer einfachen Objektserialisierung oder der Verwendung des *Kalong*-Mobilitätsmodells (vgl. Abschnitt 3.3.2) geben, welches eine effizientere Migration insbesondere bei mehreren vorab bekannten Migrationsschritten verspricht. Außerdem sollen die Effekte einer Migration frei wählbar sein, sodass Agenten bei einer Migration entweder verschoben, kopiert oder geklont werden können.

Unterstützung verschiedener Initiatoren Die Migration soll von verschiedenen Seiten aus angestoßen werden können. Einerseits soll natürlich ein Agent selbst seine Migration veranlassen können, aber auch der Benutzer oder die Ausführungsumgebung sollen die Möglichkeit bekommen, einen Agenten zwangsläufig migrieren zu lassen. Dies kann in unterschiedlichen Fällen von Nutzen sein: 1) ein Agent ist unerwünscht, weil er beispielsweise zu viele Ressourcen verbraucht oder die Sicherheit gefährdet, 2) der Benutzer möchte aus etwaigen Gründen einen Agenten explizit in eine andere Ausführungsumgebung migrieren lassen und 3) die Ausführungsumgebung wird aufgrund eines Fehlers oder auf Anweisung des Benutzers beendet. In [Braun und Rossak 2005, Seite 85] wird argumentiert, dass eine starke Migration nur von einem Agenten initiiert werden kann. Es wird daher bei der Umsetzung gefordert, einen Mechanismus zu entwickeln, der diese Einschränkung umgeht. Hinzu kommt, dass auch andere Agenten die Möglichkeit bekommen sollen, eine Migration lokal oder entfernt laufender Agenten anzuweisen. Obwohl insbesondere dieser Punkt Risiken birgt, ließ sich auf diese Weise zum Beispiel die Kontrolle eines Agenten über von ihm erzeugte Duplikate realisieren.

Intelligente Ausnahme- und Fehlerbehandlung Die einzelnen Phasen der Migration bieten eine Vielzahl möglicher Fehlerquellen. Diese sind zum einen technisch durch den Transfer eines Agenten über möglicherweise unzuverlässige Übertragungskanäle bedingt, entstehen aber auch entweder durch Fehler im Programmcode (z.B. nicht-serialisierbare Klassen) oder mangelnde Rechte (z.B. nicht autorisierte Ausführung). Die Migrationskomponente soll versuchen, bestimmte Fehler, zum Beispiel einen Verbindungsabbruch bei einem Transfer, durch wiederholte Versuche automatisch zu umgehen. Fehler, die jedoch nicht behebbar sind, soll sie entsprechend anzeigen und ggf. Maßnahmen zur Wiederherstellung eines fehlerfreien Zustandes ergreifen. Das Anzeigen eines Fehlers kann zum Beispiel über eine Ausnahme (engl. *exception*) oder den Aufruf einer vordefinierten Fehlermethode geschehen. Zur Herstellung eines ehemals fehlerfreien Zustandes können ggf. vorab hinterlegte Sicherheitskopien automatisch reaktiviert werden. Die zu ergreifenden Maßnahmen richten sich nach den Umständen des Fehlers und vor allem nach der Migrationsphase, in der ein Fehler aufgetreten ist. Tritt ein Fehler in den letzten drei Phasen der Migration in der empfangenden Ausführungsumgebung auf (vgl. Abbildung 3.10), so soll der Fehler zurück an die versendende Migrationskomponente kommuniziert und von dieser entsprechend bearbeitet werden (z.B. Neustart des Agenten und Aufruf einer Fehlermethode).

Entfernte Befehlsausführung Migrationskomponenten verschiedener Ausführungsumgebungen sollen sich untereinander verständigen und die entfernte Ausführung von Aktionen veranlassen können. Hierzu müssen sie über Nachrichten Art und Inhalte der Aktionen kommunizieren. Anstatt einen Agenten zu migrieren, kann eine Migrationskomponente zum Beispiel das entfernte Starten eines Agenten veranlassen, vorausgesetzt, der benötigte Programmcode ist in der entfernten Ausführungsumgebung bereits vorhanden. Auch soll eine Sicherheitskopie aus der Ferne wieder reaktiviert, in andere Ausführungsplattformen verschoben oder gelöscht werden können. Und schließlich soll ein Benutzer aus der Ferne die Migration eines Agenten zurück auf seine Ursprungsplattform veranlassen können.

Hohe Ausführungspriorität Eine Migration kann unter Umständen ein zeitkritischer Vorgang sein. Vergeht zu viel Zeit zwischen der Initiierung und der Durchführung einer Migration, kann es passieren, dass das Migrationsziel evtl. nicht mehr erreichbar ist, dies gilt insbesondere für ad-hoc Netzwerke. Daher ist eine zeitnahe Ausführung der Migration erforderlich. Um dies bestmöglich zu gewährleisten, soll der Ausführungsprozess der Migrationskomponente eine hohe Priorität besitzen. Hierdurch ist jedoch nur eine geringe Gesamtbelastung des Systems zu erwarten, da die Komponente nur bei Bedarf für kurze Zeit aktiv wird.

Einfache und komplexe Schnittstellen Um einem Programmierer bequemen Zugriff auf die Funktionen der Migrationskomponente zu erlauben, sollen die Schnittstellen möglichst einfach gehalten werden. Zwar sollen die einzelnen Parameter eines Migrationsvorgangs, zum Beispiel die Art der Migration, etwaige Sicherungsmaßnahmen und das Verhalten im Fehlerfall, abhängig von den Umweltbedingungen vom Programmierer vorgegeben werden können, es müssen aber auch einfache Standards angeboten werden. Derartige Standards können in Form von Migrationsprofilen einfach vom Programmierer verwendet werden, falls keine besonderen Anforderungen an die Verlässlichkeit und Effizienz eines Vorgangs bestehen.

Nach der im Folgenden vorgestellten Architektur der Komponente wird die Umsetzung der hier dargelegten Anforderungen in Abschnitt 5.3.3 anhand der Beschreibung verschiedener Kernfunktionen der Komponente detailliert erläutert. Programmcodebeispiele werden außerdem die Benutzung der Schnittstellen veranschaulichen.

5.3.2 Architektur

Anders als der Resource Facilitator weist der Migration Manager keine komplexe Architektur auf. Vielmehr setzt sich der Agent aus einer Reihe von Zielen und Plänen zusammen, die bei Eintreten bestimmter Ereignisse aktiviert werden. Beispielsweise stellt die Aufforderung zu einer Migration oder zur Übermittlung einer Nachricht an einen entfernten Agenten ein solches Ereignis dar. Diese Aufforderungen werden als Nachrichten an den Migration Manager verschickt. Bei Eintreffen einer Nachricht wird durch die Mechanismen der Plattform ein Ereignis generiert und ein entsprechender Plan für die Ausführung ausgewählt.

Daten, die von verschiedenen Plänen aus observiert und manipuliert werden, zum Beispiel das Migrationsprotokoll oder die Sicherheitskopien von Agenten, werden in der Wissensbasis des Migration Managers gespeichert und sind so von allen Plänen aus zugreifbar. Um die Kapselung und Wiederverwendbarkeit einzelner Module zu ermöglichen, sind thematisch zusammengehörige Pläne, Ziele, Ereignisse und Fakten in sogenannten *Capabilities* zusammengefasst.

5.3.3 Funktionsweise

Dieser Abschnitt beschreibt auszugsweise die Funktionen des Migration Managers. Hierbei werden einige wesentlich Funktionen im Detail beschrieben, um die generelle Funktionsweise des Agenten zu verdeutlichen. Auf eine komplette Beschreibung aller Funktionen wird an dieser Stelle verzichtet und stattdessen auf die weitergehende Dokumentation und den Programmcode des Prototypen verwiesen.

Initiieren einer Migration

Eine wesentliche Anforderungen an die Migrationskomponente bestand darin, dass eine Migration von einem Benutzer, einer Ausführungsumgebung oder einem Agenten sowohl lokal als auch entfernt initiiert werden können sollte. Unter Umständen sollte ein Agent außerdem ein Vetorecht besitzen, um beispielsweise nicht-autorisierten Migrationsaufforderungen selbständig widersprechen zu können. Als zusätzliche Anforderungen sollten darüber hinaus keine wesentlichen Änderungen an dem Programmcode des Rahmenwerkes vorgenommen werden müssen, das heißt, die Programmierschnittstelle, zum Beispiel eines Plans, sollte nicht geändert werden. Somit ist es nicht möglich, wie bei anderen Rahmenwerken, eine Methode `doMove(...)` oder `doCloneRemotely(...)` anzubieten, die von jedem Plan aus aufgerufen werden kann.

```
1
2  // create the migration goal and set neccessary parameter
3  IGoal goal = createGoal("migrate");
4  goal.getParameter("destination").setValue("134.100.5.165");
5
6  // dispatch the goal to start migration
7  try {
8    dispatchTopLevelGoalWait(goal);
9    System.out.println("Agent␣migrated␣successfully");
10 }
11 catch(GoalFailureException exc) {
12   System.err.println("Migration␣failed");
13 }
```

Listing 5.1: Erstellen eines Migrationsziels

Stattdessen wird die Migration in Jadex als ein *MigrationGoal* (dt. Migrationsziel) definiert und über eine Menge an Plänen realisiert. Ein Agent, der migrieren möchte, muss ein solches MigrationGoal erzeugen und aktivieren, wodurch der Vorgang schließlich angestoßen wird. Listing 5.1 zeigt die Erstellung eines MigrationGoals. Das Ziel benötigt mindestens einen zusätzlichen Parameter, welcher die Adresse des Zielorts angibt (Zeile 4). Mit dem Aufruf von `dispatchTopLevelGoalAndWait` (Zeile 8) wird die aktuelle Ausführung des Plans unterbrochen und das neue Ziel als Option dem internen Schlussfolgerungsprozess des Agenten hinzugefügt. Sobald das Ziel aktiviert wird, versendet der Agent eine Aufforderung an den lokalen Migration Manager, welcher daraufhin die erste Migrationsphase einleitet und die Ausführung des Agenten aussetzt (engl. *suspend*). Tritt ein Fehler während der Migration auf, so schlägt das MigrationGoal fehl und eine entsprechende Ausnahme wird dem Agenten mittels einer *GoalFailureException* angezeigt (Zeile 11).

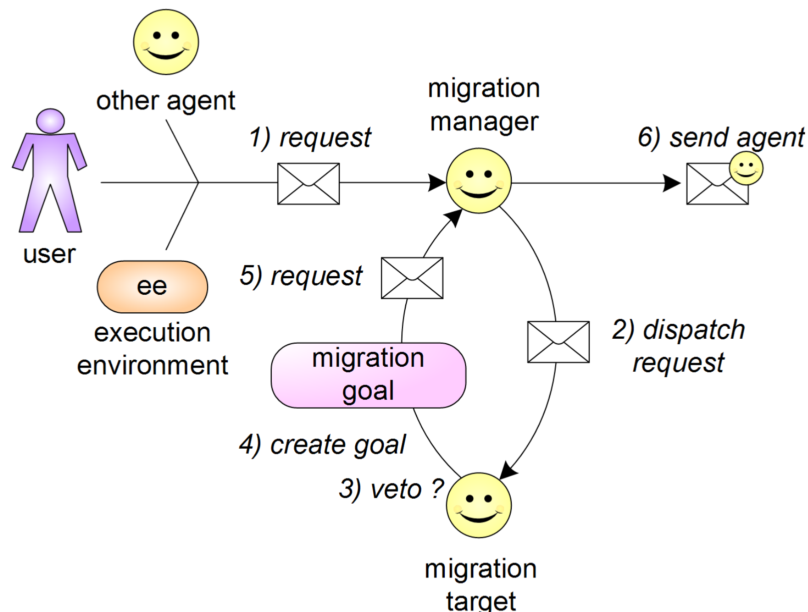

Abbildung 5.5: MM: Initiierung einer Migration (eigene Darstellung)

In Abbildung 5.5 ist dieses Vorgehen in den Schritten 4 bis 6 dargestellt. Die ersten drei Schritte beziehen sich auf den Fall, dass eine Migration nicht von dem Agenten, sondern von einem Benutzer, der Ausführungsumgebung oder einem anderen Agenten eingeleitet wird. Hierbei müssen diese eine Aufforderung an den Migration Manager schicken (Schritt 1), welcher die Nachricht an den zu migrierenden Agenten weiterleitet (Schritt 2). Dieser hat die Möglichkeit sein Vetorecht auszuüben und die Migration zu verweigern (Schritt 3). Nutzt er dieses Recht nicht, schließen sich die oben bereits erwähnten Schritte 4 bis 6 an. Natürlich ist es auch möglich einen zu migrierenden Agenten direkt zu adressieren, anstatt den Umweg über den Migration Manager zu nehmen. Durch den Umweg ist es jedoch möglich, Authentifizierungs- und Autorisierungsmechanismen an zentraler Stelle im Migration Manager einzubauen.

```
1
2  AgentIdentifier agentToCopy = new AgentIdentifier("copyme@myplatform",
3                                        new String[] {"134.100.1.001"});
4
5  // create the migration goal and set parameters
6  IGoal goal = createGoal("migrate");
7  goal.getParameter("destination").setValue(MigrationUtil.getLocalPublicIP());
8  goal.getParameter("effect").setValue(MigrationUtil.COPY);
9  goal.getParameter("profile").setValue(MigrationUtil.FAST_PROFILE);
10 goal.getParameter("agent").setValue(agentToCopy);
11
12 // dispatch the goal to start migration
13 try {
14   dispatchTopLevelGoalAndWait(goal);
15 }
16 catch(GoalFailureException exc) {
17   System.err.println(exc.getMessage());
18 }
```
Listing 5.2: Erstellen eines Migrationsziels mit zusätzlichen Parametern

Für ein MigrationGoal können neben dem erforderlichen Parameter für den Zielort noch eine Reihe weiterer, optionaler Parameter angegeben werden (siehe Listing 5.2). So ist es zum Beispiel möglich, den Effekt einer Migration (Verschieben, Kopieren oder Klonen des Agenten) zu spezifizieren oder die Art der Migration vorzugeben (Objektserialisierung, Verwenden der *Kalong*-Komponente oder eine starke Migration). Diese Angaben können jedoch auch durch einen Parameter für ein bestimmtes Migrationsprofil abgekürzt werden. Ein Migrationsprofil beinhaltet bereits eine Reihe von Angaben, zum Beispiel Art und Effekt einer Migration und ob die Migration zuverlässig oder schnell abgewickelt werden soll. Soll ein Agent nicht verschoben, sondern kopiert oder geklont werden, so kann außerdem ein Name vorgegeben werden, nach dem die Agentenkopie beim Starten benannt wird.

Der Parameter `agent` erlaubt die Angabe einer Agenten-ID (AID), welche einen Agenten durch seinen Namen und seine Adresse beschreibt. Die Angabe dieses Parameters ist immer dann erforderlich, wenn der zu migrierende Agent nicht selber der Initiator der Migration ist. Der Programmcode in Listing 5.2 soll zum Beispiel einen entfernt laufenden Agenten dazu veranlassen, eine Kopie seiner selbst auf die lokale Plattform zu übertragen. Wurde die Anweisung `dispatchTopLevelGoalAndWait` (Zeile 14) erfolgreich ausgeführt, so wurde eine frische Kopie des angegeben Agenten auf die lokale Plattform transferiert und gestartet. Andernfalls schlägt das Ziel fehl und eine Ausnahmebehandlung kann durchgeführt werden (Zeile 16).

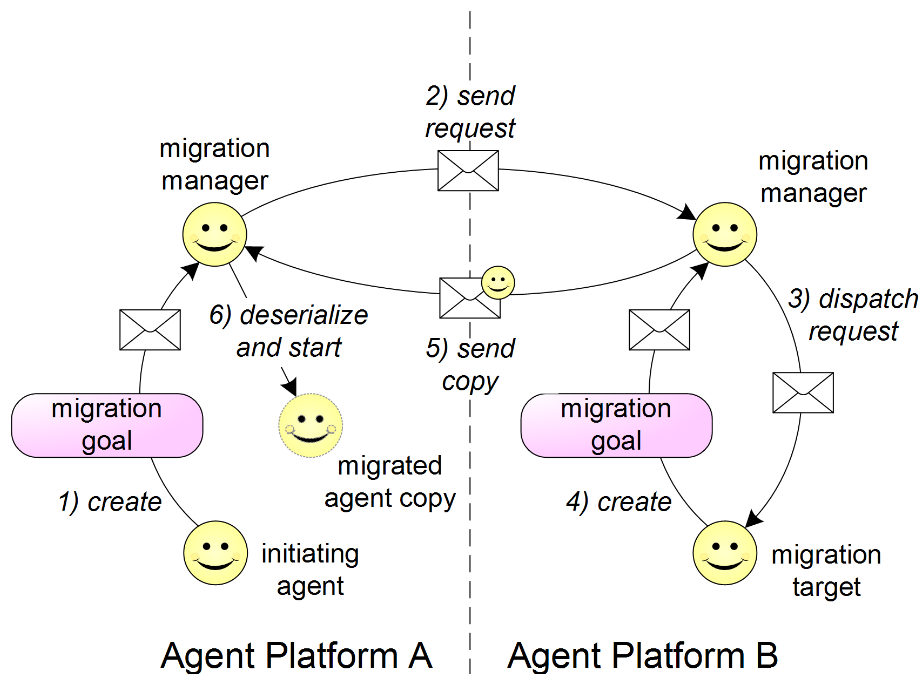

Abbildung 5.6: MM: Aufforderung zur entfernten Migration (eigene Darstellung)

In Abbildung 5.6 ist die Kommunikation unter den Beteiligten für obiges Szenario dargestellt. In diesem Beispiel initialisiert und erstellt ein Agent auf der *Agentenplattform A* ein MigrationGoal (Schritt 1), durch welches eine Nachricht an den lokalen Migration Manager versandt wird. Dieser leitet sie in Schritt 2 an den Migration Manager des zu migrierenden Agenten weiter. Der Migration Manager der Agentenplattform B überprüft bei Empfang der Nachricht ggf. die Authentizität und Berechtigung des Absenders und leitet die Aufforderung anschließend an den zu

migrierenden Agenten weiter (Schritt 3). Der Empfang dieser weitergeleiteten Nachricht führt automatisch zur Instanziierung eines MigrationGoals für den Agenten (Schritt 4). Mit Aktivierung dieses Ziels wird eine Nachricht an den lokalen Migration Manager geschickt, welcher daraufhin eine Kopie des Agenten erstellt und diese als Inhalt einer Nachricht an den Migration Manager auf Agentenplattform A zurückschickt (Schritt 5). Dieser hat schließlich die Aufgabe den Agenten aus der Nachricht zu entpacken, zu deserialisieren und schließlich zu starten (Schritt 7).

Die Gründe, die Kommunikation über die Migration Manager abzuwickeln, anstatt den Initiator und den zu migrierenden Agenten direkt miteinander kommunizieren zu lassen, sind vielfältig. Zum einen bieten die Migration Manager eine einheitliche Schnittstelle, über die sowohl Agenten, Benutzer und die Ausführungsumgebung eine Migration veranlassen können. Des Weiteren sind sie für das eventuelle Anlegen von Sicherheitskopien der zu versendenden Agenten verantwortlich und überwachen alle Phasen der Migration, sind somit also für die Behandlung von Fehlern zuständig. Soll ein Agent bei der Migration verschoben werden, anstatt wie im Beispiel kopiert, agieren die Migration Manager auch als sogenannte *Adressauflöser* (engl. *resolver*), um den migrierten Agenten an sie adressierte Nachrichten hinterher zu schicken. Außerdem lassen sich etwaige Sicherheitsmechanismen in die Migration Manager implementieren, die zum Beispiel die Authentizität und Autorisierung des Versenders überprüfen können. Und schließlich sind sie auch für das Deserialisieren eines empfangenen Agenten verantwortlich, starten diesen und melden ihn beim lokalen Agent Management System (AMS) an.

Realisierung der starken Migration

Bei einer starken Migration werden sowohl die Daten als auch der Ausführungszustand eines Agenten von einer Ausführungsumgebung in eine andere transferiert. Wie bereits in Abschnitt 3.1.5 erläutert, ist die starke Migration in Java nicht ohne Weiteres zu realisieren, da Java keine Schnittstellen für den Zugriff auf den Ausführungszustand eines Programmes bietet. Um diesen Einschränkungen beizukommen, haben Projekte wie *Javaflow* [Curdt 2007] und *Rife/continuation* [Bevin 2007] Methoden entwickelt, die während der Ausführung eines Programmes den Ausführungszustand respektive die Belegungen der lokalen Variablen, den Aufrufstapel sowie den Befehlszeiger in jedem Programmschritt protokollieren. Ein solches Protokoll wird in diesem Kontext auch als *Continuation* (dt. Weiterführung) bezeichnet. Der Vorteil dieses Ansatzes ist, dass keine Änderungen an der virtuellen Maschine von Java vorgenommen werden müssen, da der Bytecode einer Anwendung um zusätzliche Protokollanweisungen ergänzt wird. Dies hat jedoch den Nachteil, dass die Größe des Bytecodes wächst und somit einerseits mehr Daten bei einer Migration übertragen werden müssen und andererseits die Ausführungsgeschwindigkeit effektiv sinkt, da zusätzliche Protokollanweisungen ausgeführt werden müssen.

Für die Umsetzung der starken Migration wurde das Javaflow-Rahmenwerk dem Rife/continuation-Projekt vorgezogen. Gründe hierfür waren vor allem die bessere Dokumentation des Rahmenwerkes sowie dessen subjektiv einfachere Schnittstellen. Hinzu kommt, dass der Einsatz von Javaflow auf mobilen Geräten bereits erfolgreich getestet werden konnte. Um einem Agenten die starke Migration zu erlauben, müssen in der derzeitigen Version des Prototypen zwei Änderungen an der Ausführungsumgebung vorgenommen werden. Zum einen muss der Bytecode aller Klassen, die in Verbindung mit einem Agenten stehen, einmalig um die oben erwähnten Protokollanweisungen ergänzt werden[7], zum anderen muss die Einstellung für die

[7]Es ist zu erwarten, dass dieser Schritt in zukünftigen Versionen durch einen speziellen *Classloader* von Javaflow übernommen und somit automatisiert werden kann.

Ausführungskomponente von Plänen angepasst werden. Eine solche Ausführungskomponente übernimmt die Abarbeitung von Planschritten und ist somit ein guter Ansatzpunkt für die Erzeugung von Continuation-Objekten.

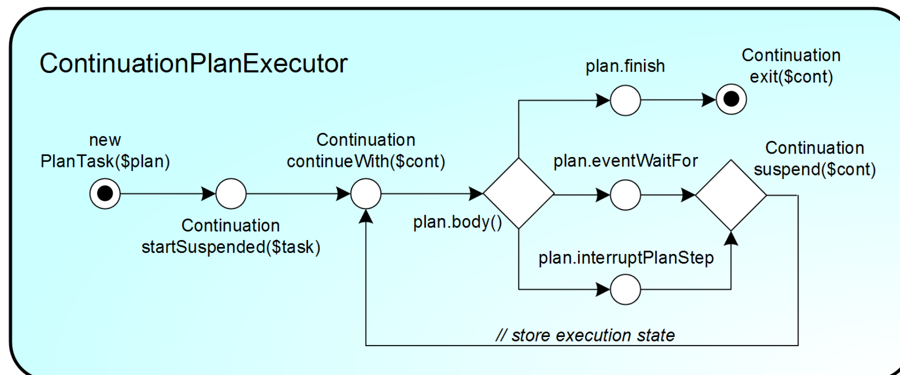

Abbildung 5.7: MM: Schematische Darstellung der Continuation-Ausführungskomponente (eigene Darstellung)

Gelangt ein neuer Plan zur Ausführung, so erstellt die Ausführungskomponente ein entsprechendes Planobjekt (als *PlanExecutionTask* bezeichnet) und speichert dieses in einer Liste von Plänen, die aktuell ausgeführt werden (siehe Abbildung 5.7). Für diesen neuen Plan wird außerdem ein neues Continuation-Objekt erzeugt, in welchem fortan der Ausführungszustand protokolliert wird, und ebenfalls in einer Liste gespeichert. Im Anschluss daran gelangt das Planobjekt durch Aufruf der planeigenen `body()`-Methode zur Ausführung. Die Abarbeitung der `body()`-Methode wird solange fortgesetzt, bis 1) die Methode erfolgreich durchlaufen wurde, 2) der Plan durch Aufruf einer `waitFor...()`-Methode angehalten wird um auf ein Ereignis zu warten oder 3) der Jadex-Interpreter die Ausführung unterbricht um Konsequenzen der Ausführung, zum Beispiel Änderungen an der Wissensbasis des Agenten, die sich auf die Agenda eines Agenten auswirken, zu verarbeiten.

In jedem dieser Fälle wird das Continuation-Objekt in der Ausführungskomponente aktualisiert und enthält somit den für einen Plan gültigen Ausführungszustand. Migriert nun ein Agent von einer Ausführungsumgebung in eine andere, so wird ebenfalls die Ausführungskomponente zusammen mit ihren Plan- und Continuation-Listen übertragen. Auf der entfernten Plattform kann dann die Ausführung der Pläne mit der zuletzt aktualisierten Continuation wieder fortgesetzt werden. Da die Continuation-Objekte jedoch nicht bei jeder einzelnen Anweisung aktualisiert werden, sondern lediglich bei Unterbrechung eines Plans, kann es unter Umständen passieren, dass ein Agent nicht exakt an der Programmstelle wieder fortgesetzt wird, an der die Migration initiiert wurde. Dies betrifft jedoch nur Fälle, in denen die Migration nicht von dem Agenten selbst initiiert wurde.

Fehlerbehandlung bei der Migration

In den einzelnen Phasen der Migration können eine Reihe von Fehlern auftreten. Je nach Ursache eines Fehlers kann auf unterschiedliche Weise darauf reagiert werden. Entweder man wiederholt eine fehlgeschlagene Aktion, bricht die Aktion ab und meldet das Fehlschlagen an den Initiator der Migration oder man versucht den Fehler anderweitig zu beheben. Im Folgenden sollen für jede der sechs Migrationsphasen (vgl. Abbildung 3.10) mögliche Fehlerquellen, soweit möglich

potenzielle Lösungsansätze sowie die Effekte eines Fehlers vorgestellt werden. Die Auflistung deckt jedoch nur einen Teil aller möglichen Fehler ab und soll lediglich als Grundlage dafür dienen, die Fehlerbehandlung in der Zukunft kontinuierlich zu verbessern.

Phase 1 In der ersten Phase der Migration wird der Migrationsprozess durch ein Kommando gestartet und die Ausführung des Agenten unterbrochen. Auch wenn ein Agent nicht unbedingt der Initiator der Migration sein muss, ruft er dieses Kommando immer von sich aus auf, indem er ein entsprechendes MigrationGoal erzeugt. Bei Aktivierung dieses Ziels wird eine Nachricht an den lokalen Migration Manager der Agentenplattform geschickt, welcher daraufhin die Ausführung des Agenten unterbricht. Das Ziel bricht mit einem Fehler ab, wenn entweder keine Zieladresse für die Migration als Parameter angegeben wurde oder der lokale Migration Manager nicht erreichbar ist. Im ersten Fall muss die Migration abgebrochen und dem Initiator eine entsprechende Fehlernachricht zugestellt werden. Im zweiten Fall könnte nach Ablauf eines Zeitintervalls ein erneuter Versuch unternommen werden, den Migration Manager zu kontaktieren. Schlägt auch dies fehl, wird der Migrationsvorgang endgültig abgebrochen.

Phase 2 Nachdem die Ausführung des Agenten erfolgreich angehalten wurde, erfasst der Migration Manager den Datenraum und ggf. den Ausführungszustand eines Agenten und versucht diese zu serialisieren. Dieser Vorgang kann beispielsweise fehl schlagen, falls nicht alle Klassen des Agenten die *Serializable*-Schnittstelle von Java implementieren oder zu wenig Ressourcen zum Speichern der serialisierten Repräsentation zur Verfügung stehen. In diesen Fällen wird der Migrationsvorgang abgebrochen und die Ausführung des Agenten wieder aufgenommen. Das Fehlschlagen der Migration wird in dem MigrationGoal über eine *GoalFailureException* als Ausnahme angezeigt und schließlich eine Fehlernachricht an den Initiator versendet.

Phase 3 In der dritten Phase soll die serialisierte Form des Agenten von der Ursprungsplattform an die Zielplattform übertragen werden. Für die Übertragung des Agenten wird ein Migrationsprotokoll eingesetzt, welches je nach Implementation einen mehrfachen Austausch von Nachrichten zwischen den Plattformen erfordert [Braun und Rossak 2005]. In der prototypischen Implementation der Migrationskomponente findet jedoch ein sehr einfaches Protokoll Verwendung, welches nur aus maximal zwei Nachrichten besteht: 1) der Nachricht, die den serialisierten Agenten enthält und 2) einer optionalen Empfangsbestätigung, die nur bei verlässlicher Migration versendet wird. Je nach Protokoll können an unterschiedlichen Stellen des Ablaufs unterschiedliche Fehler auftreten und Fehlerbehandlungen eingeleitet werden. An dieser Stelle werden nur allgemeine Verbindungsprobleme berücksichtigt, bei denen eine Verbindung zur Zielplattform gar nicht erst aufgebaut werden kann oder während einer Übertragung abbricht. In beiden Fällen kann der Migration Manager durch wiederholtes Senden der Nachricht versuchen, den Fehler zu beheben. Schlägt dies mehrfach fehl, muss der Migrationsvorgang abgebrochen werden. Die Ausführung des Agenten wird in diesem Fall einfach auf der lokalen Plattform fortgesetzt, die serialisierte Form verworfen und der Fehler im MigrationGoal und bei den Initiatoren entsprechend angezeigt.

Phase 4 Die vierte Phase, in der der Agent von der Zielplattform empfangen wird, läuft parallel zur dritten Phase ab. Die möglichen Fehler werden also von der sendenden Plattform bereits abgefangen. Einzig die Überprüfung der Authentizität des sendenden Migration Managers und dessen Berechtigung einen Agenten auf die Zielplattform übertragen zu dürfen, können Fehler

verursachen, welche dem Sender angezeigt und von diesem an die Initiatoren weitergeleitet werden müssen.

Phase 5 Nach erfolgreichem Empfang des Agenten, soll dieser in der fünften Phase auf der Zielplattform deserialisiert werden. Hier können insbesondere Fehler auftreten, falls die serialisierte Form des Agenten nicht vollständig oder fehlerhaft über unzuverlässige Übertragungsprotokolle (z.B. das *UDP*-Protokoll) übertragen wurde. In diesem Fall muss der Fehler dem sendenden Migration Manager mitgeteilt werden, damit dieser den Agenten erneut übertragen kann. Ein anderes Problem stellt unter Umständen der für die Deserialisierung benötigte Programmcode dar. Java verlangt schon bei der Deserialisierung das Vorhandensein sämtlichen Programmcodes aller Klassen, unabhängig davon, ob diese später verwendet werden. Fehlende Klassen müssen an dieser Stelle über eine pull-Strategie von der Ursprungsplattform nachgeladen werden. Bricht jedoch beim Transfer der Klassen die Kommunikationsverbindung zwischen den Beteiligten ab, kann die Deserialisierung nicht fortgesetzt werden. Dieses Problem kann unter Umständen durch wiederholte Verbindungsversuche oder durch Laden der benötigten Klassen von anderen Plattformen (vgl. das Codebasen-Konzept von *Kalong*, Abschnitt 3.3.2) behoben werden. Weitere Probleme ergeben sich, falls die Migration verlässlich durchgeführt werden sollte und keine erneute Verbindung zwischen den Beteiligten hergestellt werden konnte. Unter diesen Umständen wartet der sendende Migration Manager vergeblich auf die abschließende Bestätigung, dass der migrierte Agent erfolgreich auf der Zielplattform gestartet wurde. In diesem Fall muss die Ausführung des Agenten auf der Ursprungsplattform wieder aufgenommen, sowohl der Agent als auch der Initiator der Migration benachrichtigt und die serialisierte Form des Agenten auf der Zielplattform wieder gelöscht werden.

Phase 6 Der Migrationsvorgang endet mit dem Starten des deserialisierten Agenten auf der Zielplattform. Für den Fall, dass bereits ein Agent mit demselben Namen auf der Plattform läuft (z.B. eine andere Kopie des Agenten von derselben Ursprungsplattform) muss der migrierte Agent vor dem Starten umbenannt werden, wobei der Migration Manager ein vordefiniertes Schema zur Erzeugung eines neuen Namens verwendet. Ein nicht behebbarer Fehler tritt auf, wenn die Ausführungsumgebung nicht über ausreichend Ressourcen zur Erzeugung des Agenten verfügt. In diesem Fall wird der Migrationsvorgang abgebrochen, der deserialisierte Agent wieder aus dem Speicher entfernt und eine Nachricht an den sendenden Migration Manager verschickt.

Die Fehlerbehandlung in allen Phasen dient grundsätzlich dem Ziel, einen Agenten nicht zu verlieren und die Konsistenz des Gesamtsystems durch Vermeidung unerwünschter Duplikate zu wahren. Bei einer verlässlichen Migration ist dies auch zu jedem Zeitpunkt durch Verwendung eines transaktionalen Migrationsprotokolls garantiert. Wurde statt der verlässlichen jedoch eine schnelle Migration gewählt, so wird der Migrationsvorgang asynchron ausgeführt (vgl. Abschnitt 3.1.2). Der empfangende Migration Manager bestätigt in diesem Fall nicht den erfolgreichen Start des Agenten und der sendende Migration Manager entfernt den Agenten von der Plattform, sobald die dritte respektive vierte Migrationsphase erfolgreich durchlaufen wurde. Tritt ein nicht zu behebender Fehler in der fünften oder sechsten Phase der Migration auf, ist der Agent verloren. Lediglich durch Reaktivierung ggf. angelegter Sicherheitskopien kann der vollständige Verlust des Agenten vermieden werden.

Zurückbeordern entfernter Agenten

Da sich Agenten autonom im Netzwerk bewegen können, muss dem Benutzer die Möglichkeit gegeben werden, jederzeit Zugriff auf seine Agenten zu erhalten, also einen entfernt laufenden Agenten wieder in die aktuelle Ausführungsumgebung des Benutzer zurück zu beordern. Da der Benutzer jedoch nicht zwangsläufig den momentanen Aufenthaltsort eines Agenten kennt, kann eine entsprechende Migrationsaufforderung unter Umständen nicht ohne Weiteres an den Agenten direkt überstellt werden.

Um eine Nachricht an einen Agenten zu übersenden, adressiert der Benutzer diese zwar an den Agenten, ohne jedoch das Adressfeld der Nachricht auszufüllen. Stattdessen trägt er in das sogenannte *Resolver*-Feld der Nachricht die Adresse seines lokalen Migration Manager ein, welcher einen Namensauflösungdienst (engl. *name resolution service*) bietet. Ein solcher Dienst kann den Namen eines Empfängers mit dessen aktueller Adresse zusammenführen und wird eingesetzt, sobald die Adresse des Empfängers unbekannt oder nicht erreichbar ist [FIPA 2002c]. Da ein Migration Manager ein Protokoll darüber führt, welcher Agent wann und wohin migriert ist, kann er, falls nötig, die Nachricht entsprechend der Route des Agenten zum nächsten Migration Manager leiten oder sie dem Agenten ggf. lokal zustellen. Empfängt der Agent schließlich die Nachricht, so initiiert er, wie vorangehend beschrieben, die Migration zurück auf seine Heimatplattform.

Kann eine Nachricht einem Agent nicht zugestellt werden, weil der Migrationspfad durch Ausscheiden einer Plattform respektive eines Migration Managers unterbrochen wurde, so kann ein Migration Manager den lokalen Resource Facilitator zu Hilfe nehmen. Hierzu stellt er zunächst eine Suchanfrage an den *Agents*-Nachrichtenkanal, welcher die Namen der laufenden Agenten auf benachbarten Plattformen enthält. Wurde der gesuchte Agent nicht gefunden, so kann er die Migrationsaufforderung an die Migration Manager aller bekannten Kontakte versenden, wobei er seinen eigenen Namen sowie die Namen aller Empfänger an die Nachricht hängt, um zirkulären Nachrichtenversand zu vermeiden. Konnte der gesuchte Agent schließlich im Netzwerk nicht aufgefunden werden, wird die Migrationsaufforderung letztendlich den Migrationspfad des Agenten entlang, zurück gesendet. Hierbei hängt jede Ausführungsumgebung, die vormals eine Sicherheitskopie des Agenten angelegt hat, einen entsprechenden Vermerk an die Nachricht.

Der Vorgang endet, sobald entweder der Agent erfolgreich auf seine Heimatplattform zurück migriert ist und dort gestartet wurde oder die Migrationsaufforderung wieder auf der Heimatplattform empfangen wurde. Falls mindestens ein Migration Manager das Vorhandensein einer Sicherheitskopie in der Nachricht vermerkt hat, kann der Benutzer in einem zweiten Schritt die Reaktivierung dieser Kopie und die anschließende Migration des Agenten fordern.

5.3.4 Zusammenfassung

Die Migrationskomponente ist für die Realisierung der schwachen und starken Migration sowie aller damit zusammenhängenden Funktionen verantwortlich. Sie zeichnet sich für für alle Phasen des Migrationsvorgangs, von der Initialisierung der Migration bis zum entfernten Starten eines Agenten verantwortlich und überwacht jede Phase um auftretende Fehler erkennen, umgehen und anzeigen zu können. Außerdem bietet sie zusätzliche Funktionen zum Anlegen und Reaktivieren von Sicherheitskopien sowie zum Weiterleiten von Nachrichten an migrierte Agenten.

Die Umsetzung der Komponente erfolgte als ein Anwendungsagent für das Jadex-Rahmenwerk. Eine Reihe von Anforderungen, die zu Beginn des Abschnittes aufgestellt wurden, bilden die Grundlage der Implementation. Die erforderlichen Funktionen wurden auf Ziele bzw. Pläne des sogenannten Migration Manager-Agenten abgebildet, die bei Eintreten bestimmter Ereignisse automatisch von der Plattform aktiviert werden. Solche Ereignisse sind beispielsweise das Empfangen bestimmter Nachrichten, die den Migration Manager zur Ausführung einer Aktion veranlassen sollen. Anhand von ausgewählten Mechanismen, die sich aus der Beschreibung von Anforderungen ergaben, wurde die Funktionsweise des Agenten, zum Beispiel das Initiieren einer Migration oder die Realisierung der starken Migration, erläutert. Für eine Beschreibung der Gesamtfunktionalität sei an dieser Stelle auf die weiterführende Dokumentation sowie den Quelltext der Komponente verwiesen.

5.4 Zusammenfassung

In diesem Kapitel wurde die Umsetzung des Konzeptes der kontextabhängigen und eigenverantwortlichen Migration vorgestellt. Die prototypische Realisierung erfolgte in Form zweier Komponenten für die Jadex-Agentenplattform, welche als Middleware zur Ausführung von BDI-Agenten fungiert und bereits im Vorwege zur Verwendung in dieser Arbeit in die J2ME-Umgebung portiert wurde.

Die erste beschriebene Komponente dient im Wesentlichen der Wahrnehmung der Umwelt in heterogenen Umgebungen. Diese sogenannte Ressource Awareness-Komponente (auch als *Resource Facilitator* bezeichnet) hat die Aufgabe, ein Umweltmodell zu erstellen, Informationen aus diesem Modell etwaigen Interessenten auf Anfrage zur Verfügung zu stellen und diese bei Eintreten von Ereignissen in der Umwelt entsprechend zu benachrichtigen. Für die Erstellung und Aktualisierung des Umweltmodells kann sich die Komponente verschiedener Discovery-Verfahren bedienen, die, abhängig vom jeweiligen Kontext, dynamisch eingesetzt werden kann. Um dies zu gewährleisten, bietet die Komponente eine Reihe generischer Schnittstellen an, über die ein Verfahren sowie dessen verwendete Repräsentationsform für Umweltinformationen, an die internen Mechanismen adaptiert werden können. Um die Kommunikation zwischen Agenten und der Ressource Awareness-Komponente auch bei Unterstützung unterschiedlicher Verfahren sicherzustellen, wurde zudem ein proprietäres Basisverfahren entwickelt, welches die grundlegenden Funktionen eines Discovery-Verfahrens sowie eine einfache Repräsentationsform für Ressourcen bietet.

Für die Organisation des Umweltmodells wurde die Metapher der Nachrichtenkanäle gewählt, welche Informationen thematisch gliedern und von einem Benutzer intuitiv verstanden werden können. Eine Reihe von Standard-Nachrichtenkanälen (*System-*, *Agents-*, *Services-*, *Contacts-* und *Heartbeat-*Kanal) wurden im Rahmen der Implementierung ebenfalls realisiert, wobei weitere, anwendungsspezifische Kanäle später einfach ergänzt werden können. Um Informationen zu be-

ziehen, müssen Agenten diese Kanäle abonnieren, wofür ebenfalls ein entsprechendes Protokoll entwickelt wurde.

Der Prototyp kann demzufolge bereits eine vorgegebene Menge an Umweltinformationen anbieten, welche über das proprietäre Basisverfahren unter den Agenten ausgetauscht werden können. Auch können Interessenten unter Verwendung einer einfachen Abfragesprache, Kanalinformationen im Vorwege filtern und mittels unterschiedlicher Parameter die Menge und die Versandart von Nachrichten beeinflussen. Unter Berücksichtigung der Einschränkungen der J2ME-Umgebung ist der Prototyp zudem auf mobilen Geräten lauffähig. Die Unterstützung für mobile ad-hoc Netzwerke resultiert zum einen aus der Fähigkeit, unterschiedliche Discovery-Verfahren und Repräsentationsformen zur Verbreitung und Aufbereitung von Informationen zu verwenden. Des Weiteren aus der Möglichkeit, unter Verwendung des proprietären Basisverfahrens, die Menge auszutauschender Daten durch vielfältige Parameter optimal auf die Umgebung einzustellen. Und zum anderen aus der Fähigkeit des Prototypen Veränderungen der Netzwerkkonnektivität festzustellen, da die Netzwerkschnittstellen von diesem kontinuierlich überwacht werden.

Die zweite Komponente, die für Jadex realisiert wurde, ist ein sogenannter *Migration Manager*-Agent. Dieser bietet eine Reihe unterschiedlicher Möglichkeiten der Migration von Agenten. Den Anforderungen entsprechend, unterstützt diese Komponente sowohl die schwache als auch die starke Migration, unterschiedliche Effekte wie das Verschieben, Kopieren oder Klonen eines Agenten sowie verschiedene Initiatoren für einen Migrationsvorgang, die eine Migration sowohl lokal als auch entfernt auslösen können. Unter anderem dadurch, dass ein Migrationsvorgang letztendlich immer von einem Agenten angestoßen wird, jedoch durch Versand von Migrationsaufforderungen beliebige Initiatoren zulässt, kann entgegen der Argumentation von [Braun und Rossak 2005] auch eine starke Migration bei beliebigen Initiatoren geboten werden. Des Weiteren kann der Migration Manager auf Anforderung Sicherheitskopien migrierender Agenten erstellen und später bei Bedarf wieder reaktivieren.

Um während eines Migrationsvorgangs auftretende Fehler aufzudecken, überwacht ein Migration Manager alle Phasen eines Migrationsvorgangs. Je nach Ursache eines Fehlers kann er durch unterschiedliche Maßnahmen versuchen, das Problem zu umgehen und zu beheben, sofern dies möglich erscheint. Andernfalls zeigt er den Fehler sowohl dem migrierenden Agenten als auch dem Initiator entweder über eine Ausnahme (engl. *exception*) oder eine Fehlernachricht an.

Außerdem fungiert ein Migration Manager als Adressauflösungsdienst (engl. *resolver*) und kann migrierten Agenten an sie adressierte Nachrichten hinterher schicken. Einerseits ist dies für die Aufrechterhaltung der Kommunikation zwischen mobilen Agenten notwendig, andererseits erlaubt dieser Mechanismus aber auch zum Beispiel die Zustellung von Migrationsaufforderungen an entfernte Agenten, deren autonom gewählter Aufenthaltsort dem Absender nicht zwangsläufig bekannt sein muss. Auf diese Weise kann ein Benutzer beispielsweise seine entfernt laufenden Agenten auf seine aktuelle Plattform zurück beordern.

Aus dem Zusammenspiel der beiden realisierten Agenten resultiert schließlich die Möglichkeit eines Agenten, kontextabhängig und eigenverantwortlich in heterogenen Umgebungen zu migrieren. Die Abhängigkeit vom Kontext bezieht sich hierbei einerseits auf den Zeitpunkt und andererseits auf die Art einer Migration. Die Ressource Awareness-Komponente kann Agenten bei Eintreten bestimmter Ereignisse informieren. Diese können daraufhin eine Migration, deren Mechanismen auf die aktuellen Gegebenheiten der Umwelt angepasst werden können, veranlassen. Die Eigenverantwortlichkeit einer Migration basiert ebenfalls auf der Kenntnis der Umwelt. Diese Forderung soll Agenten in die Pflicht nehmen, eigenständig Vorkehrungen zu veranlassen,

die einen vollständigen Verlust des Kontaktes zu ihrem Benutzer nach Möglichkeit vermeiden. Hierzu gehören beispielsweise das Anlegen von Sicherheitskopien, sobald die Agenten in ein adhoc Netzwerk migrieren oder eine schnelle und effiziente Migration zurück auf das Gerät ihres Benutzers, sobald dieses aus dem Netzwerk auszuscheiden droht.

In der weiteren Entwicklung der Prototypen gilt es nachzuweisen, dass die realisierten Mechanismen, Modelle und Schnittstellen den Anforderungen aus der Praxis gerecht werden. Hierzu müssen neben den IP-basierten Protokollen vor allem auch weitere gängige Protokolle (z.B. *Bluetooth*) unterstützt werden. Der Ressource Awareness-Komponente müssten Adapter für verbreitete Discovery-Verfahren (z.B. *JXTA* und *Jini*) sowie für bedeutsame Repräsentationsformen wie *RDF* und *OWL* hinzugefügt werden, um die in der Praxis weit verbreitete Software-Infrastruktur nutzen zu können. Für die Weiterentwicklung des Prototypen gilt es, die Behandlung von Fehlern weitergehend zu optimieren und vor allem die Anbindung an die Kalong-Komponente zu implementieren, welche bei Veröffentlichung dieser Arbeit noch nicht verfügbar war.

Ein letzter wichtiger Punkt betrifft die Sicherheit des Systems. Diesbezüglich wurden bereits in Abschnitt 4.3.4 etwaige Bedenken formuliert, für die praxistaugliche Lösungsmöglichkeiten gefunden und in die Komponenten integriert werden müssen. Hierzu gehören vor allem Sicherheitsmechanismen zur Authentifizierung und Autorisierung sowie die verschlüsselte Übertragung von Daten.

6 Diskussion, Zusammenfassung und Ausblick

In diesem letzten Kapitel werden die Ergebnisse und möglichen Folgen dieser Arbeit zusammengefasst. Es werden die Risiken und Chancen, die sich aus einer Umsetzung und einen möglichen Einsatz des Konzeptes ergeben, angedeutet und in einem Ausblick mögliche technische und konzeptionelle Erweiterungen des Konzeptes vorgestellt sowie eine Anwendungsvision präsentiert.

6.1 Diskussion

In diesem Abschnitt sollen die Risiken und Chancen der kontextabhängigen, eigenverantwortlichen Migration und Verteilung mobiler Agenten diskutiert werden, denn insbesondere die Aspekte der Autonomie und der Verantwortung bedürfen in dem gegebenen Kontext einer kritischen Betrachtung. Der wesentliche Hintergrund ist, dass Technikanwendungen niemals nur ihr Arbeitsziel erfüllen, sondern darüber hinaus weitere Nebenwirkungen, unter anderem auf die natürliche und soziale Umwelt, haben. Hierzu werden im Folgenden Risiken und Chancen des Konzeptes angedeutet, zu denen jedoch nicht weiter Stellung bezogen werden soll, da sie weitergehender, interdisziplinärer Untersuchung bedürfen.

6.1.1 Risiken

Wenn einem Software-Artefakt autonomes Handeln zugebilligt wird, geht dies einher mit der Verringerung der Kontrolle seitens des Benutzers. Dies soll im Wesentlichen zu einer Entlastung des Benutzers führen, bedeutet aber auch, dass dem Benutzer die Abläufe im Hintergrund nicht mehr unbedingt zugänglich sind und dieser in gewisser Hinsicht entmündigt wird. Auch stellt sich die Frage, wer die Verantwortung für einen Agenten, der eigenverantwortlich handeln soll, übernimmt, falls dieser seine Kompetenzen nicht ausüben kann oder überschreitet und somit Schaden anrichtet? Und wer haftet beispielsweise, falls ein fremder Agenten auf dem eigenen System Informationen anbietet oder nachfragt, die gegen geltendes Recht verstoßen[1]?

Des Weiteren ist zu überlegen, ob Benutzer generell überhaupt bereit sind, einem Agenten persönliche Informationen anzuvertrauen, wenn sich dieser zur Bearbeitung von Aufgaben in fremde und möglicherweise nicht vertrauenswürdige Ausführungsumgebungen begibt. Denn einerseits besteht die Gefahr, dass Informationen des Benutzers unberechtigt ausgelesen oder manipuliert werden, andererseits kann nicht unter allen Umständen sichergestellt werden, dass die Integrität eines Agenten während der Ausführung auf fremden System nicht kompromittiert oder

[1] In [Nitschke 2006] werden die rechtlichen Konsequenzen des Einsatzes (mobiler) Agenten untersucht. Unter anderem wird in diesem Bericht auch darauf eingegangen, wer für mobile Agenten verantwortlich und haftbar ist. Demnach ist der Prinzipal eines Agenten für dessen Handeln bzw. dessen Absichtserklärungen (engl. *declaration of intention*) verantwortlich, sofern ein Fehlverhalten des Agenten nicht auf Fehler in der Programmierung zurückzuführen sind. In diesem Fall wäre der Programmierer (mit-)verantwortlich. Absichtserklärungen eines Agenten verlieren jedoch ihre Rechtswirksamkeit (engl. *legal effectiveness*), sobald dieser nicht mehr dem Einflussbereich seines Prinzipals unterliegt, was bei autonom migrierenden Agenten durchaus geschehen kann. Dies würde bedeuten, dass zum Beispiel ein Vertrag, den ein mobiler Agent, welcher nicht mehr im Einflussbereich seines Prinzipals steht (zum Beispiel durch Netzwerkpartitionierung), abgeschlossen hat, für seinen Prinzipal nicht bindend ist. Jedoch ist auch der Betreiber des Gerätes, auf dem sich ein mobiler Agent befindet, von der gesetzlichen Verantwortung für fremde Agenten entbunden.

einem Agenten die Rückkehr zu seinem Benutzer verweigert wird. Benutzer könnten sich außerdem von autonomen Agenten bedroht fühlen, wenn diese die Eigenschaft besitzen, ihre Umwelt wahrzunehmen (und somit den Benutzer bzw. dessen Geräte als Teil der Umwelt) und so die Möglichkeit haben, einen Benutzer in einem offenen System zu überwachen und zu begleiten respektive zu verfolgen. Hinsichtlich des Schutzes der Privatsphäre eines Benutzers ist weiterhin zu bedenken, dass dessen Verhalten und Interessen, sobald er Informationen seines Systems öffentlich anbietet, von Dritten eingesehen und verfolgt werden können.

Die geforderte Offenheit des Systems birgt darüber hinaus Gefahren, sofern nicht sichergestellt werden kann, dass sich alle Teilnehmer wohlgesonnen sind. So könnten Agenten sich ähnlich wie Viren und Würmer vermehren, Ressourcen des Netzwerkes und der Ausführungsumgebungen blockieren oder diese für unlautere Zwecke missbrauchen [Vigna 2004].

6.1.2 Chancen

Das Konzept birgt natürlich aber auch Chancen. So können durch die Offenheit des Systems und das öffentliche Anbieten und Nachfragen von Ressourcen, diese besser ausgenutzt werden und liegen nicht brach (vgl. beispielsweise Projekte wie *Seti@Home* [Berkeley University 2007] oder *grid.org* [United Devices 2007]), was insbesondere im Zuge der überall verfügbaren Informationsverarbeitung (engl. *pervasive computing*) eine schnellere und effektivere Bearbeitung von Aufgaben ermöglichen könnte. Des Weiteren können Agenten, sobald sie mit einer neuen Umgebung konfrontiert werden, benötigte Ressourcen (z.B. Drucker) selbständig ausfindig machen, ohne dass der Benutzer manuell etwaige Konfigurationseinstellungen vornehmen muss.

Durch die Fähigkeit der Agenten ihren Benutzer autonom überall hin zu begleiten, ohne dass der Benutzer hierfür aktiv Sorge tragen muss, wird dieser letztlich entlastet und von etwaigen Synchronisationsaufgaben befreit, da er auf verschiedenen Geräten immer mit derselben Instanz einer Anwendung arbeiten kann. So hätte ein Benutzer auch überall Zugriff auf seine persönlichen Daten und kann in einer vertrauten virtuellen Umgebung mit seinen gewohnten Anwendungen arbeiten, ohne sich fortwährend an wechselnde Software-Infrastrukturen gewöhnen zu müssen. Die Agenten könnten sich darüber hinaus im Laufe der Zeit auch besser an ihren Benutzer adaptieren und ihm gezielter helfen, da sie ständigen Kontakt mit diesem halten und somit dessen Präferenzen und Gewohnheiten lernen können. Auf diese Weise muss sich der Benutzer nicht allerorten an die Technik anpassen, sondern die Technik passt sich dem Benutzer an.

6.2 Zusammenfassung

In den vorigen Kapiteln wurde das Konzept der kontextabhängigen und eigenverantwortlichen Migration von Software-Agenten in heterogenen Umgebungen erarbeitet. Motiviert ist dieses Konzept einerseits durch aktuelle Forschungsströme in den Bereichen der allgegenwärtigen und mobilen Informationsverarbeitung (engl. *ubiquitous computing* und *mobile computing*) sowie der Umgebungsintelligenz (engl. *ambient intelligence*), andererseits durch den Wunsch nach Unterstützung nomadischer Benutzer im Umgang mit aktueller Technik.

Besondere Herausforderungen erwuchsen aus der Beschaffenheit der adressierten Einsatzumgebung. Denn die Beachtung der zuvor erwähnten Paradigmen erfordert einerseits die Berücksichtigung verschiedener Arten von Ressourcen (insbesondere z.B. mobile Geräte) und andererseits auch die Beachtung einer neuen Art der Kommunikationsbeziehung (z.B. mobile adhoc Netzwerke). Daher wurde bei den entworfenen Modellen und Methoden die Möglichkeit

des gleichzeitigen Einsatzes verschiedener, speziell auf einen Einsatzkontext zugeschnittener Verfahren, berücksichtigt. Auf diese Weise lassen sich einerseits bereits bestehende Software-Infrastrukturen einbinden, und andererseits einzelne Verfahren abhängig vom Kontext dynamisch auswählen.

Das entwickelte Konzept lässt sich in zwei Teilaspekte untergliedern: die Wahrnehmung der Umwelt zur Erstellung eines Umweltmodells und die an die Umweltgegebenheiten angepasste Migration. Das Umweltmodell stellt hierbei den Agenten ihren Kontext in Form einer generischen Zugriffsstruktur zur Verfügung, in der Informationen über interessante Entitäten in der Umwelt und diese Entitäten betreffende Ereignisse repräsentiert sind. Die Schwierigkeit, Informationen in diese generische Zugriffsstruktur zu integrieren, liegt in der Heterogenität der Entitäten und der Beziehungen zwischen diesen. In der Praxis haben sich unterschiedliche Verfahren für den Austausch derartiger Informationen etabliert, die jedoch jeweils nur einen bestimmten Aspekt der zu betrachtenden Umwelt abdecken und in sofern mehreren Einschränkungen unterliegen. Im Rahmen dieser Arbeit wurde eine Auswahl solcher Verfahren näher untersucht und anhand eines Kriterienkatalogs miteinander verglichen. Aus dieser Untersuchung ergab sich die Schlussfolgerung, einen Mechanismus bzw. eine Middleware-Komponente zu entwerfen, die den gleichzeitigen Einsatz einer Vielzahl von Verfahren erlaubt, die Auswahl ein oder mehrerer aktiver Verfahren aber dem aktuellen Kontext anpassen kann, um die Auslastung der für die Ausführung benötigten Hardware- und Energieressourcen zu verringern.

Abhängig von diesen Kontextinformationen kann ein Agent eine Entscheidung für oder gegen eine Migration in eine andere Ausführungsumgebung treffen sowie die Rahmenbedingungen eines Migrationsvorgangs vorgeben. Tritt in der Umwelt zum Beispiel ein Ereignis ein, welches einen Agenten zur Migration bewegt, beispielsweise das Erscheinen einer neuen Ausführungsumgebung, so soll dieser in eigener Verantwortung die Rahmenbedingungen der Übertragung festlegen können. Diese Rahmenbedingungen sind abhängig vom Kontext und beziehen sich im Wesentlichen auf die Effizienz und Verlässlichkeit der Übertragung. So kann der Agent beispielsweise seine aktuelle Ausführungsumgebung anweisen, vor dem Transfer eine Sicherheitskopie seiner selbst anzulegen, die Übertragung als Transaktion durchzuführen und im Falle eines Fehlers die Sicherheitskopie automatisch wieder zu reaktivieren. Außerdem kann der Agent zu Gunsten der Migrationseffizienz selbst entscheiden, ob sein Programmcode und sein Ausführungszustand komplett übertragen oder bestimmte Teile erst bei Bedarf aus der Ferne nachgeladen werden sollen. Für die Umsetzung der Funktionalitäten wurden Anforderungen an eine Migrationskomponente beschrieben, welche im Auftrag eines Agenten, des Benutzers oder der Ausführungsumgebung die Migration planen, ausführen und in allen Phasen überwachen soll.

Diese vorgestellten Optionen tragen insbesondere den unterschiedlichen Verbindungscharakteristiken von Infrastruktur- und mobilen ad-hoc Netzwerken Rechnung. Während in Infrastrukturnetzwerken die Übertragung von Daten schnell und zuverlässig geschieht, weisen ad-hoc Netzwerke meist geringe Datenraten, hohe Latenzzeiten, eine dynamische Topologie und vor allem instabile Verbindungen auf. Abhängig von der Umgebung, in der der Agent migriert, muss dieser daher einen Kompromiss zwischen einer verlässlichen und einer effizienten Übertragung eingehen, da sich beide Qualitäten gegenseitig nahezu ausschließen. Hierbei ist vor allem darauf zu achten, dass der Agent nicht durch endgültigen Abbruch einer Verbindung oder Fehler in einer der Migrationsphasen, den Kontakt zu seinem Benutzer unwiderruflich verliert.

Die für die Umsetzung dieser Forderungen benötigten Mechanismen erlauben darüber hinaus auch eine einfache verteilte Bearbeitung von Aufgaben durch Erzeugen entfernter Duplikate eines Agenten und Delegation von Teilaufgaben an diese. Auf diese Weise können Ressourcen bes-

ser ausgelastet und somit die Bearbeitungszeit einer Aufgabe möglicherweise verringert werden. Eine verkürzte Bearbeitungszeit ist wiederum besonders in ad-hoc Netzwerken interessant, da sich diese jederzeit auflösen können und somit die Chance der Fertigstellung lang andauernder Berechnungen gering ist.

Im Rahmen dieser Arbeit wurden fünf Anwendungsbeispiele gegeben, die einen möglichen Einsatz des Konzeptes in der Praxis veranschaulichen sollten. Anhand dieser Beispiele wurden konkrete Anforderungen an die beiden oben bereits erwähnten Komponenten formuliert, welche sich an den Benutzer, die Ausführungsumgebung und die Agenten richten. Anhand dieser Anforderungen wurde schließlich ein Umweltmodell spezifiziert und die Entitäten und Ereignisse identifiziert, welche im Kontext dieser Arbeit für Agenten von Interesse sein könnten. Dieses Umweltmodell stellt die Grundlage für die Wahrnehmung der Umwelt dar und die verantwortliche Komponente hat die Aufgabe, alle verfügbaren Informationen in einer einheitlichen Form in dieses Modell zu integrieren. Für die Spezifikation von Rahmenbedingungen einer Migration wurde unter Verwendung der Mobilitätssprache *MoL* ein Modell der Mobilität konzipiert, welches die Anforderungen an die Funktionalität der Mobilitätsmechanismen vorschreibt und die Grundlage für die Migrationskomponente darstellt.

Um die Umsetzbarkeit des Konzeptes zu belegen, wurde darüber hinaus die prototypische Implementation zweier Komponenten beschrieben, welche für die *Jadex*-Agentenplattform entwickelt wurden. Die erste Komponente dient der Wahrnehmung der Umgebung und stellt anderen Agenten ein Umweltmodell zur Verfügung, welches Informationen über verfügbare Ressourcen enthält. Die zweite Komponente ist für die Migration von Agenten verantwortlich. Sie bietet vielfältige Arten der Migration, welche von einem Agenten, unter Berücksichtigung der aktuellen Umweltgegebenheiten, adaptiv ausgewählt werden können. Weiterhin ermöglicht sie eine verlässliche Migration auch in dynamischen Umgebungen, indem sie zusätzliche Funktionen zur Sicherung des Migrationsvorgangs und zur Vermittlung von Nachrichten bereitstellt. Aus dem Zusammenspiel dieser Komponenten resultiert schließlich die Möglichkeit, Agenten kontextabhängig und eigenverantwortlich in heterogenen Umgebungen migrieren zu lassen.

6.3 Ausblick

Das in dieser Arbeit vorgestellte Konzept der kontextabhängigen, eigenverantwortlichen Migration wurde in den einzelnen Kapitels zwar eingehend, aber nicht erschöpfend behandelt. Eine Reihe von Gesichtspunkten mussten bei der Betrachtung außen vor gelassen werden, da diese entweder den Rahmen der Arbeit übersteigen oder erst durch praktische Erfahrungen mit dem Konzept offenkundig werden. Aus diesen Gründen werden in diesem Abschnitt eine Reihe von möglichen Weiterentwicklungen präsentiert, welche technischen Verbesserungen, konzeptionelle Erweiterungen und zukünftige Anwendungsvisionen umfassen.

Hinsichtlich der technischen Verbesserungen seien zum Beispiel die Implementation einer Vielzahl weiterer Adapter für Discovery-Verfahren und Repräsentationssprachen zu erwähnen. Denn der Nutzen des Modells einer Ressourcenumwelt hängt im Wesentlichen von der Menge und Qualität der in einem Modell enthaltenen Informationen ab. Hierfür müssten Informationen aus bereits bestehenden Software-Infrastrukturen in das Umweltmodell integriert werden können. Beispielsweise könnten mit einem *Jini*-Adapter bereits alle in einem Jini-Netzwerk angebotenen Dienste in das Umweltmodell integriert werden, auch ohne dass diese explizit durch andere Resource Awareness-Komponenten beworben werden.

Um die Qualität bestimmter Informationen, zum Beispiel die Positionsdaten eines Gerätes oder die Verbindungsgüte zu anderen Netzwerkteilnehmern, zu verbessern und vorhersagen zu können, gilt es bessere Software-Sensoren zu implementieren (vgl. Abschnitt 4.5.5). Diese Software-Sensoren sollen beispielsweise die Datenrate sowie die Antwortzeit bei einem Nachrichtenaustausch messen. Aus diesen Daten ließen sich die Effizienz von Migrationsstrategien weiter erhöhen, Rückschlüsse auf die Distanz zweier Geräte in einem Funknetzwerk schließen und ein möglicher Abbruch einer Verbindung frühzeitig erkennen. Hierfür bedarf es jedoch spezieller, hardwarespezifischer Bibliotheken mit entsprechenden Schnittstellen für die Java-Programmiersprache, die derartige Daten bereits auf Ebene der Hardware erfassen können.

Außerdem fehlen Untersuchungen bezüglich der Performanz dieses Konzeptes bzw. der Umsetzung desselben. Das Konzept verspricht unter anderem eine schnellere und effektivere Bearbeitung von Aufgaben, ohne diese Aussage weitergehend zu evaluieren. Es wurde in Abschnitt 3.2 zwar auf die Performanz des Paradigmas mobiler Agenten im Vergleich zu anderen Paradigmen eingegangen, die konzipierten Middleware-Komponenten beanspruchen selbst, je nach Konfiguration jedoch, ein nicht unerhebliches Maß an Ressourcen (insbesondere Netzwerk-, Speicher- und Prozessorressourcen). Eine Performanzuntersuchung könnte zeigen, für welche Arten von Anwendungen sich trotzdem eine Leistungssteigerung erwarten ließe und welche Mechanismen weitergehend optimiert werden müssen.

Zusätzlich sind sämtliche Sicherheitsbelange zu nennen, die in das Konzept sowie dessen Umsetzung eingearbeitet werden müssen (vgl. Abschnitt 4.3.4). Ohne die Berücksichtigung grundlegender Sicherheitsmechanismen ist ein Einsatz in der Praxis nicht verantwortbar, da ein Missbrauch der Funktionen einfach möglich wäre. Hierzu zählen beispielsweise die Authentifizierung von Benutzern, Ausführungsumgebungen und Agenten, die Integration von *Sandboxes* in Ausführungsumgebungen, um den Zugriff von Agenten auf lokale Ressourcen (z.B. Dateien) beschränken zu können, sowie eine Rechteverwaltung, die unterschiedlichen Akteuren jeweils individuelle Rechte zuweisen kann.

Weitere interessante Möglichkeiten ergeben sich durch konzeptuelle Erweiterungen. Hier wäre zum Beispiel die soziale Umwelt von Agenten zu nennen, deren Integration in das Modell der Ressourcenumwelt bereits in Abschnitt 4.4.1 angedeutet wurde. Die Resource Awareness-Komponente bietet bereits grundlegende und weitergehende Funktionen zur Suche nach Agenten, Diensten etc. und muss zur Aktualisierung ihres Umweltmodells außerdem Verbindungen zu anderen RA-Komponenten aufbauen und erhalten. Über diese Kontakte ließe sich ein soziales Netzwerk aufbauen, auf welchem zum Beispiel Gruppen- und Rollenkonzepte aufsetzen könnten.

Da grundlegende Mechanismen zur Replikation und Verteilung von Agenten durch die Migrationsmechanismen ohnehin bereits abgedeckt sind, wäre es darüber hinaus vorstellbar, weitergehende Funktionen zur Automatisierung des Vorgangs in das Konzept zu integrieren. In diesem Fall müsste ein Agent nicht explizit selbst die Distribution von Teilaufgaben an Kopien seiner selbst im Netzwerk übernehmen und für eine spätere Zusammenführung von Teilergebnissen sowie die Behandlung von Ausnahmesituationen (im Falle von Fehlern) sorgen. Diese Aufgabe könnte von einem speziell hierfür entwickelten Dienst, der auf dem Migration Manager aufsetzt, übernommen werden, was die Entwicklung und das Deployment von verteilten Anwendungen vereinfachen würde.

Auch der Punkt der Lernfähigkeit von Agenten kann das Konzept weitergehend bereichern. So könnte beispielsweise die Resource Awareness-Komponente im Laufe der Zeit lernen, welche Informationen für den Benutzer bzw. dessen Agenten wirklich von Interesse sind und so die

Kommunikation mit anderen RA-Komponenten auf diese Informationen fokussieren. Auf diese Weise würde die Menge auszutauschender Informationen sinken und die Effizienz der Wahrnehmung verbessert werden können.

Hinsichtlich der Anwendung des Konzeptes in der Praxis gibt es eine Reihe von Visionen, in denen der Alltag eines Benutzers durch mobile Agenten, die kontextabhängig und eigenverantwortlich migrieren bzw. handeln können, vereinfacht wird und diesem weitere Möglichkeiten eröffnet werden. Ein nomadischer Benutzer wird sich nicht allerorten an neue Technikumgebungen gewöhnen müssen, sondern arbeitet jederzeit und überall mit seinen ihm vertrauten Anwendungsinstanzen. Diese können sich darüber hinaus selbstständig an die Umgebungen anpassen, indem sie benötigte Ressourcen (z.B. Drucker) eigenständig finden und der Benutzer manuell keine Einstellungen mehr vornehmen muss.

Auch werden Benutzer ihre Anwendungen immer bei sich tragen, sei es ihr Mail-Programm, ihren Kalender, das Fotoalbum oder einen Nachrichtendienst. Persönliche Daten sind somit jederzeit und allerorten zugreifbar. Die Anwendungen arbeiten im Hintergrund und werden selbständig aktiv, sobald sie sich in Reichweite benötigter Ressourcen (z.B. eines Internetzugangs) oder Informationen befinden. So ist auch eine neue Art von Anwendungen vorstellbar, die Menschen mit ähnlichen Interessen und Motiven zusammenbringen. Zum Beispiel könnte eine Anwendung Teilnehmer einer Konferenz mit ähnlichen Interessenprofilen zusammenführen, sobald sich diese in gegenseitiger Nähe aufhalten oder Geschäfte könnten vorbeilaufenden Passanten Angebote unterbreiten und eine Anwendung informiert ihren Benutzer, sobald ein Angebot dessen Interessen entspricht. Letztlich ergänzt dieses Konzept die Paradigmen der Umgebungsintelligenz (engl.*ambient intelligence*) sowie der allgegenwärtigen Informationsverarbeitung (engl. *ubiquitous computing*) durch personalisierte Anwendungen, die eigenständig mit der Umgebung interagieren können.

Literaturverzeichnis

[Abowd u. a. 1999] ABOWD, Gregory D. ; DEY, Anind K. ; BROWN, Peter J. ; DAVIES, Nigel ; SMITH, Mark ; STEGGLES, Pete: Towards a Better Understanding of Context and Context-Awareness. In: *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*. London, UK : Springer-Verlag, 1999. – ISBN 3540665501, S. 304–307

[Acharya u. a. 1997] ACHARYA, Anurag ; RANGANATHAN, M. ; SALTZ, Joel H.: Sumatra: A Language for Resource-Aware Mobile Programs. In: *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*. London, UK : Springer-Verlag, 1997. – ISBN 3–540–62852–5, S. 111–130

[Alterovitz 2001] ALTEROVITZ, Ron: Message Routing Over a (Bluetooth) Scatternet / California Institute of Technology. http://www.cs.caltech.edu/ rona/cs134c/final/, June 2001. – Forschungsbericht

[Antoniou und van Harmelen 2003] ANTONIOU, Grigoris ; HARMELEN, Frank van: Web Ontology Language: OWL / Department of Computer Science, University of Crete, Department of AI, Vrije Universiteit Amsterdam. http://www.cs.vu.nl/ frankh/postscript/OntoHandbook03OWL.pdf, 2003. – Forschungsbericht

[Avancha u. a. 2002] AVANCHA, Sasikanth ; JOSHI, Anupam ; FININ, Timothy: Enhanced Service Discovery in Bluetooth. In: *Computer* 35 (2002), June, Nr. 6, S. 96–99. – ISSN 0018–9162

[Barbeau 2000] BARBEAU, Michel: Service Discovery Protocols for Ad Hoc Networking / School of Computer Science, Carleton University, Canada. 2000. – Forschungsbericht. CASCON 2000 Workshop on ad hoc communications

[Bellifemine u. a. 2003] BELLIFEMINE, F. ; CAIRE, G. ; POGGI, A. ; RIMASSA, G.: *JADE - A White Paper*. Sept. 2003. – http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf

[Bellifemine u. a. 2007] BELLIFEMINE, Fabio L. ; CAIRE, Giovanni ; GREENWOOD, Dominic: *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007. – ISBN 0470057475

[Berger und Watzke 2002] BERGER, Michael ; WATZKE, Michael: Agents in Ad Hoc Environments (FIPA 00068) / FIPA. 2002. – Forschungsbericht. FIPA00068

[Berkeley University 2007] BERKELEY UNIVERSITY, BU. *Seti@Home*. http://setiathome.berkeley.edu/. March 2007

[Berler u. a. 2000] BERLER, Mark ; EASTMAN, Jeff ; JORDAN, David ; RUSSELL, Craig ; SCHADOW, Olaf ; STANIENDA, Torsten ; VELEZ, Fernando ; CATTELL, R. G. G. (Hrsg.) ; BARRY, Douglas K. (Hrsg.): *The object data standard: ODMG 3.0*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000. – ISBN 1–55860–647–5

[Bettstetter und Renner 2000] BETTSTETTER, C. ; RENNER, C.: A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. In: *Proc. EUNICE Open European Summer School*. Twente, Netherlands, September 2000

[Bevin 2007] BEVIN, Geert. *Rife web continuations*. http://rifers.org/wiki/display/RIFE/Web+continuations. January 2007

[Bhagwat und Segall 1999] BHAGWAT, P. ; SEGALL, A. *A routing vector method (RVM) for routing in Bluetooth scatternets*. 1999

[Bisignano u. a. 2003] BISIGNANO, Mario ; CALVAGNA, Andrea ; MODICA, Giuseppe D. ; TOMARCHIO, Orazio: Expeerience: A Jxta Middleware for Mobile Ad-Hoc Networks. In: *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–2023–5, S. 214

[Bloom 1970] BLOOM, Burton H.: Space/time trade-offs in hash coding with allowable errors. In: *Commun. ACM* 13 (1970), Nr. 7, S. 422–426. – ISSN 0001–0782

[BluetoothGroup 2006] BLUETOOTHGROUP, Bluetooth Special Interest G.: *Bluetooth Specification v.1.1 - Part E Service Discovery Protocol, February 2001*. www.bluetooth.com. August 2006. – Seite 1095 im Spec-paper, Stand November 2004

[BMBF 2007] BMBF, Bundesministerium für Bildung und F. *IT-Forschung 2006 - Förderprogramm Informations- und Kommunikationstechnik,Abschnitt 4: Forschungsbereiche*. http://www.it2006.de/de/147.php. 27. March 2007

[Borselius 2002] BORSELIUS, N.: Mobile agent security. In: *Electronics Communications Engineering Journal: Special issue security for mobility* 14 (2002), October, Nr. 5, S. 211–218

[Bouchenak u. a. 2004] BOUCHENAK, S. ; HAGIMONT, D. ; KRAKOWIAK, S. ; PALMA, N. D. ; BOYER, F.: Experiences implementing efficient Java thread serialization, mobility and persistence. In: *Softw. Pract. Exper.* 34 (2004), Nr. 4, S. 355–393. – ISSN 0038–0644

[Bratman 1987] BRATMAN, Michael E.: *Intention, Plans, and Practical Reason*. Harvard University Press, 1987

[Braubach 2007] BRAUBACH, L.: *Architekturen und Methoden zur Entwicklung verteilter agentenorientierter Softwaresysteme*, Universität Hamburg, Diss., 2007

[Braubach u. a. 2005] BRAUBACH, Lars ; POKAHR, Alexander ; BADE, Dirk ; KREMPELS, Karl-Heinz ; LAMERSDORF, Winfried: Deployment of Distributed Multi-Agent Systems. In: MARIE-PIERRE GLEIZES, Franco Z. (Hrsg.): *5th International Workshop on Engineering Societies in the Agents World*, Springer-Verlag, Berlin Heidelberg, August 2005, S. 261–276

[Braun 2003] BRAUN, Peter: *The Migration Process of Mobile Agents-Implementation, Classification, and Optimization*, Friedrich-Schiller-Universität Jena, Computer Science Department, Diss., May 2003

[Braun u. a. 2001] BRAUN, Peter ; ERFURTH, Christian ; ROSSAK, Wilhelm: Performance Evaluation of Various Migration Strategies for Mobile Agents. In: KILLAT, Ullrich (Hrsg.) ; LAMERSDORF, Winfried (Hrsg.): *Informatik Aktuell*, 2001, S. 315–324

[Braun und Kern 2005] BRAUN, Peter ; KERN, Steffen: Towards Adaptive Migration Techniques for Mobile Agents / Friedrich Schiller Universität Jena, Computer Science Department. 2005. – Forschungsbericht. AAMAS '05

[Braun u. a. 2005] BRAUN, Peter ; MUELLER, Ingo ; KOWALCZYK, Ryszard ; KERN, Steffen: Attacking the Migration Bottleneck of Mobile Agents / Swinburne University of Technology. 2005. – Forschungsbericht

[Braun und Rossak 2005] BRAUN, Peter ; ROSSAK, Wilhelm ; COX, Tim (Hrsg.) ; PREISENDANZ, Christa (Hrsg.): *Mobile Agents - Basic Concepts, Mobility Models and the Tracy Toolkit*. Morgan Kaufmann and dpunkt.verlag, 2005

[Broekstra und Kampman 2004] BROEKSTRA, Jeen ; KAMPMAN, Arjohn: *SeRQL: An RDF Query and Transformation Language*. 2004. – Submitted to the International Semantic Web Conference, ISWC 2004

[Burden 2005] BURDEN, Peter. *WSDL - Lecture Notes*. http://www.scit.wlv.ac.uk/ jphb/cp2101/wsdl/WSDL.html. 2005

[Busse 1999] BUSSE, Ingo: *Mobile Agents in Telecommunications*, Technical University of Berlin , Department of Computer Sciences, Diss., February 1999

[Cabri u. a. 1998] CABRI, G. ; LEONARDI, L. ; ZAMBONELLI, F.: Mobile Agent Technology - Current Trends and Perspectives - Cabri.pdf / AICA, Napoli, Italy. 1998. – Forschungsbericht

[Campo 2002] CAMPO, C. *Service Discovery in Pervasive Multi-Agent Systems*. AAMAS Workshop on Ubiquitous Agents on embedded, wearable, and mobile agents. July 2002

[Carzaniga u. a. 1997] CARZANIGA, Antonio ; PICCO, Gian P. ; VIGNA, Giovanni: Designing Distributed Applications with a Mobile Code Paradigm. In: *Proceedings of the 19th International Conference on Software Engineering*. Boston, MA, USA, 1997

[Chakeres und Belding-Royer 2004] CHAKERES, Ian D. ; BELDING-ROYER, Elizabeth M.: AODV Routing Protocol Implementation Design. In: *24th International Conference on Distributed Computing Systems Workshops* 06 (2004), S. 698–703. ISBN 0–7695–2087–1

[Chakraborty u. a. 2002] CHAKRABORTY, Dipanjan ; JOSHI, Anupam ; FININ, Tim ; YESHA, Yelena: GSD: A Novel Group-based Service Discovery Protocol for MANETs. In: *4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN)*. Stockholm. Sweden, September 2002

[Chen und Kotz 2000] CHEN, Guanling ; KOTZ, David: A Survey of Context-Aware Mobile Computing Research / Dartmouth College. Hanover, NH, USA : Dartmouth College, 2000. – Forschungsbericht

[Chen u. a. 2001] CHEN, Harry ; JOSHI, Anupam ; FININ, Timothy: Dynamic Service Discovery for Mobile Computing: Intelligent Agents Meet Jini in the Aether. In: *Cluster Computing* 4 (October 2001), October, Nr. 4, S. 343–354

[Cheng und Marsic 2000] CHENG, L. ; MARSIC, I.: Service discovery and invocation for mobile ad hoc networked appliances. In: *2nd International Workshop on Networked Appliances (IWNA'2000)*. New Brunswick, NJ, December 2000

[Cheng 2002] CHENG, Liang: Service Advertisement and Discovery in Mobile Ad hoc Networks / Department of Computer Science and Engineering. 2002. – Forschungsbericht

[Coulouris und Dollimore 1994] COULOURIS, George F. ; DOLLIMORE, Jean: *Distributed systems: concepts and design, Edition 2*. 2. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1994. – SunRPC Pages 138–144. – ISBN 0–201–18059–6

[Curdt 2007] CURDT, Torsten. *The Javaflow component, Jakarta Commons project*. http://jakarta.apache.org/commons/sandbox/javaflow/index.html. January 2007

[Delisle u. a. 2006] DELISLE, Pierre ; LUEHE, Jan ; ROTH, Mark: JavaServer Pages Specification, Version 2.1 / Sun Microsystems, Inc. 2006. – Forschungsbericht

[Döhler u. a. 2004] DÖHLER, Arndt ; ERFURTH, Christian ; ROSSAK, Wilhelm: An Infrastructure-based Approach to Support Dynamic Networks with Mobile Agents. In: *Proceedings of the 1st IFIP TC6 WG6.6 International Workshop on Autonomic Communication (WAC 2004)*, 2004

[Elenius 2003] ELENIUS, Daniel: *Service Discovery in Peer-to-Peer Networks*, Department of Computer and Information Science, Linköping Institute of Technology, Diplomarbeit, 2003

[Erfurth 2004] ERFURTH, Chrisitan: *Proaktive autonome Navigation für mobile Agenten*, Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik, Diss., Juli 2004

[Erfurth und Rossak 2002] ERFURTH, Christian ; ROSSAK, Wilhelm: Adapting Proactive Mobile Agents to Dynamically Reconfigurable Networks. In: *Informatica (Slovenia)* 26 (2002), Nr. 4, S. 401–406

[Fensel u. a. 2000] FENSEL, D. ; HORROCKS, I. ; HARMELEN, F. V. ; DECKER, S. ; ERDMANN, M. ; KLEIN, M.: OIL in a nutshell. In: DIENG, R. (Hrsg.): *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW'00)*, Springer-Verlag, 2000 (Lecture Notes in Artificial Intelligence 1937), S. 1–16

[Finin u. a. 1994] FININ, T. ; FRITZSON, R. ; MCKAY, D. ; MCENTIRE, R.: KQML as an Agent Communication Language. In: ADAM, N. (Hrsg.) ; BHARGAVA, B. (Hrsg.) ; YESHA, Y. (Hrsg.): *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*. Gaithersburg, MD, USA : ACM Press, 1994, S. 456–463

[FIPA 2002a] FIPA. *FIPA Abstract Architecture Specification*. December 2002a

[FIPA 2002b] FIPA. *FIPA ACL Message Structure Specification*. June 2002b

[FIPA 2002c] FIPA. *FIPA Agent Message Transport Service Specification*. December 2002c

[FIPA 2002d] FIPA. *FIPA Contract Net Interaction Protocol Specification*. December 2002d

[FIPA 2002e] FIPA. *FIPA Subscribe Interaction Protocol Specification*. December 2002e

[FIPA 2004] FIPA. *FIPA Agent Management Specification*. March 2004

[Frank u. a. 2004] FRANK, Christian ; HANDZISKI, Vlado ; KARL, Holger: Service Discovery in Wireless Sensor Networks / Technical University Berlin. 2004. – Forschungsbericht

[Free Software Foundation 2007] FREE SOFTWARE FOUNDATION, FSF. *GNU Lesser General Public License*. http://www.gnu.org/licenses/lgpl.html. May 2007

[Friday u. a. 2004] FRIDAY, Adrian ; DAVIES, Nigel ; WALLBANK, Nat ; CATTERALL, Elaine ; PINK, Stephen: Supporting service discovery, querying and interaction in ubiquitous computing environments. In: *Wireless Networking* 10 (2004), November, Nr. 6, S. 631–641. – ISSN 1022–0038

[Fuenfrocken 1999] FUENFROCKEN, Stefan: Transparent Migration of Java-Based Mobile Agents. In: *MA '98: Proceedings of the Second International Workshop on Mobile Agents*. London, UK : Springer-Verlag, 1999. – ISBN 3–540–64959–X, S. 26–37

[Fuggetta u. a. 1998] FUGGETTA, Alfonso ; PICCO, Gian P. ; VIGNA, Giovanni: Understanding Code Mobility. In: *IEEE Trans. Softw. Eng.* 24 (1998), Nr. 5, S. 342–361. – ISSN 0098–5589

[Gong 2001] GONG, Li: JXTA: A Network Programming Environment. In: *IEEE Internet Computing* 5 (2001), June, Nr. 3, S. 88–95. – ISSN 1089–7801

[GoPC 2007] GOPC. *GOPC: The computer that fits student life*. http://www.gopc.net. May 2007

[Gray 1995] GRAY, Robert S.: Agent Tcl : A transportable agent system. In: *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*. Baltimore, Maryland, December 1995

[Guttman 1999] GUTTMAN, Erik: Service Location Protocol: Automatic Discovery of IP Network Services. In: *IEEE Internet Computing* 3 (1999), July, Nr. 4, S. 71–80

[Guttmann u. a. 1999] GUTTMANN, E. ; PERKINS, C. ; VEIZADES, J. ; DAY, M. *Service Location Protocol Version 2. IETF Internet Draft, RFC 2608*. Internet. June 1999

[Haase u. a. 2004] HAASE, Peter ; BROEKSTRA, Jeen ; EBERHART, Andreas ; VOLZ, Raphael: A comparison of RDF query languages. In: *Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, 2004.*, 2004

[Hagimont und Ismail 1999] HAGIMONT, Daniel ; ISMAIL, L.: A performance evaluation of the mobile agent paradigm. In: *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM Press, 1999. – ISBN 1–58113–238–7, S. 306–313

[Harbeck 2004] HARBECK, Mathias: *BDI-Agentensysteme auf mobilen Geräten*, University of Hamburg, Diplomarbeit, September 2004

[Helal u. a. 2003] HELAL, S. ; DESAI, N. ; VERMA, V. ; LEE, Choonhwa: Konark - a service discovery and delivery protocol for ad-hoc networks. In: *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE* 3 (2003), March, S. 2107–2113

[Helmy u. a. 2005] HELMY, Ahmed ; GARG, Saurabh ; NAHATA, Nitin ; PAMU, Priyatham: CARD: a contact-based architecture for resource discovery in wireless ad hoc networks. In: *Mob. Netw. Appl.* 10 (2005), Nr. 1-2, S. 99–113. – ISSN 1383–469X

[Hermann u. a. 2000] HERMANN, Reto ; HUSEMANN, Dirk ; MOSER, Michael ; NIDD, Michael ; ROHNER, Christian ; SCHADE, Andreas: DEAPspace: transient ad-hoc networking of pervasive devices. In: *MobiHoc '00: Proceedings of the 1st ACM international symposium on*

Mobile ad hoc networking & computing. Piscataway, NJ, USA : IEEE Press, 2000. – ISBN 0–7803–6534–8, S. 133–134

[Horrocks 2002a] HORROCKS, Ian: DAML+OIL: a Description Logic for the Semantic Web. In: *IEEE Data Engineering Bulletin* 25 (2002), Nr. 1, S. 4–9

[Horrocks 2002b] HORROCKS, Ian: DAML+OIL: A Reason-able Web Ontology Language. In: *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*. London, UK : Springer-Verlag, 2002b. – ISBN 3–540–43324–4, S. 2–13

[Horrocks u. a. 2002] HORROCKS, Ian ; PATEL-SCHNEIDER, Peter F. ; HARMELEN, Frank van: Reviewing the design of DAML+OIL: an Ontology Language for the Semantic Web. In: *Eighteenth national conference on Artificial intelligence*. Menlo Park, CA, USA : American Association for Artificial Intelligence, 2002. – ISBN 0–262–51129–0, S. 792–797

[Illmann u. a. 2000] ILLMANN, T. ; KARGL, F. ; WEBER, M. ; KRUGER, T.: Migration of mobile agents in Java: Problems, classification and Solution. In: *Proc. of the International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA 00)*, 2000

[ISO/IEC 2001] ISO/IEC, 14977:1996. *Extended BNF - A generic base standard (EBNF).* http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26153. August 2001

[Jansen und Karygiannis 1999] JANSEN, W. ; KARYGIANNIS, T.: Mobile Agent Security / National Institute of Standards and Technology. 1999. – Forschungsbericht

[Jansen 2000] JANSEN, Wayne A.: Countermeasures for mobile agent security. In: *Computer Communications* 23 (2000), Nr. 17, S. 1667–1676

[Johnson und Maltz 1996] JOHNSON, David B. ; MALTZ, David A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: IMIELINSKI (Hrsg.) ; KORTH (Hrsg.): *Mobile Computing* Bd. 353. Kluwer Academic Publishers, 1996

[Karvounarakis u. a. 2002] KARVOUNARAKIS, Gregory ; ALEXAKI, Sofia ; CHRISTOPHIDES, Vassilis ; PLEXOUSAKIS, Dimitris ; SCHOLL, Michel: RQL: a declarative query language for RDF. In: *WWW '02: Proceedings of the 11th international conference on World Wide Web*, 2002, S. 592–603

[Kendall u. a. 1994] KENDALL, Samuel C. ; WALDO, Jim ; WOLLRATH, Ann ; WYANT, Geoff: A Note on Distributed Computing / Sun Microsystems Inc. Mountain View, CA, USA : Sun Microsystems, Inc., November 1994. – Forschungsbericht

[Kern und Braun 2005] KERN, Steffen ; BRAUN, Peter: Towards Adaptive Migration Strategies for Mobile Agents / Friedrich Schiller Universität Jena, Computer Science Department. 2005. – Forschungsbericht. WRAC '05

[Lange und Oshima 1998] LANGE, Danny ; OSHIMA, Mitsuru: *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998

[Lawrence 2002a] LAWRENCE, Jamie. *LEAP for Ad-Hoc Networks*. http://jamie.ideasasylum.com/research/publications/. 2002a

[Lawrence 2002b] LAWRENCE, Jamie: LEAP into Ad-Hoc Networks. In: *In Proc. of the Ubiquitous Computing Workshop, Bologna, Italy* Media Lab Europe, 2002

[Lee u. a. 2002] LEE, Sung J. ; SU, William ; GERLA, Mario: On-demand multicast routing protocol in multihop wireless mobile networks. In: *Mob. Netw. Appl.* 7 (2002), December, Nr. 6, S. 441–453. – ISSN 1383–469X

[Luck u. a. 2005] LUCK, M. ; MCBURNEY, P. ; SHEHORY, O. ; WILLMOTT, S.: *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005

[Magedanz u. a. 1999] Kap. 5 In: MAGEDANZ, Thomas ; BÄUMER, Christoph ; BREUGST, Markus ; CHOY, Sang: *Grasshopper - A Universal Agent Platform Based on OMG MASIF and FIPA Standards*. ACTS Project InfoWin, September 1999, S. 99–111

[Marin-Perianu u. a. 2005] MARIN-PERIANU, R. S. ; HARTEL, P. H. ; SCHOLTEN, J.: A Classification of Service Discovery Protocols / Univ. of Twente. Enschede : Centre for Telematics and Information Technology, University of Twente, June 2005 (TR-CTIT-05-25). – Technical Report. – ISSN 1381–3625

[Marrow u. a. 2001] MARROW, P. ; KOUBARAKIS, M. ; LENGEN, R.H. van ; VALVERDE-ALBACETE, F. ; BONSMA, E. ; CID-SUERIO, J. ; FIGUEIRAS-VIDAL, A.R. ; GALLARDO-ANTOLIN, A. ; HOILE, C. ; KOUTRIS, T. ; MOLINA-BULLA, H. ; NAVIA-VAZQUEZ, A. ; RAFTOPOULOU, P. ; SKARMEAS, N. ; TRYFONOPOULOS, C. ; WANG, F. ; XIRUHAKI, C.: Agents in Decentralised Information Ecosystems: The DIET Approach. In: *Proccedings of the Symposium on Information Agents for E-Commerce, AISB Convention*. York, UK, March 2001

[McGrath 2000] MCGRATH, Robert E.: Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing / University of Illinois, Department of Computer Science. Champaign, IL, USA : University of Illinois at Urbana-Champaign, 2000. – Forschungsbericht

[Merriam-Webster 2006] MERRIAM-WEBSTER. *Merriam-Webster Online Dictonary*. http://www.m-w.com/. September 2006

[Mertens u. a. 2004] MERTENS, Koenraad ; HOLVOET, Tom ; BERBERS, Yolande: Adaptation in a Distributed Environment. In: *Environments for Multiagent Systems*, 2004, S. 49–59

[Mertens u. a. 2005] MERTENS, Koenraad ; HOLVOET, Tom ; BERBERS, Yolande: A case for adaptation of the distributed environment layout in multiagent applications. In: *SELMAS '05: Proceedings of the fourth international workshop on Software engineering for large-scale multi-agent systems*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–116–3, S. 1–8

[Microsoft 1997] MICROSOFT. *Introduction to Active Channel Technology*. http://msdn.microsoft.com/workshop/delivery/channel/overview/overview.asp. 1997

[Munafo 2007] MUNAFO, Robert P. *The Encyclopedia of the Mandelbrot Set*. http://www.mrob.com/pub/muency.html. March 2007

[Nahata u. a. 2002] NAHATA, Nitin ; PAMU, Priyatham ; GARG, Saurabh ; HELMY, Ahmed: Efficient resource discovery for large scale ad hoc networks using contacts. In: *SIGCOMM Comput. Commun. Rev.* 32 (2002), July, Nr. 3, S. 32–32. – ISSN 0146–4833

[Netscape 1997] NETSCAPE. *Netscape unveils Netscape Netcaster*. http://cgi.netscape.com/newsref/pr/newsrelease385.html. April 1997

[Nidd 2001] NIDD, M.: Service discovery in DEAPspace. In: *EEE Personal Communications* 8 (2001), August, Nr. 4, S. 39–45

[Nitschke 2006] NITSCHKE, Tanja: Legal Consequences of Agent Deployment. In: KIRN, S. (Hrsg.) ; HERZOG, O. (Hrsg.) ; LOCKEMANN, P. (Hrsg.) ; SPANIOL, O. (Hrsg.): *Multiagent Engineering - Theory and Applications in Enterprises*, Springer, 2006, S. 597–618

[Nivio 2007] NIVIO. *Nivio: The desktop that works for you*. http://www.nivio.com. May 2007

[Object Management Group 1999] OBJECT MANAGEMENT GROUP, OMG. *The Common Object Request Broker:Architecture and Specification*. http://www.omg.org/docs/formal/99-10-07.pdf. October 1999

[Odell u. a. 2002] ODELL, James ; PARUNAK, H. Van D. ; FLEISCHER, Mitchell ; BRUECKNER, Sven: Modeling Agents and Their Environment. In: GIUNCHIGLIA, Fausto (Hrsg.) ; ODELL, James (Hrsg.) ; WEISS, Gerhard (Hrsg.): *AOSE* Bd. 2585, Springer, 2002, S. 16–31

[OMG 2003] OMG: Deployment and Configuration of Component-based Distributed Applications Specification / Object Management Group (OMG). 2003. – Forschungsbericht

[Ortega-Ruiz u. a. 2006] ORTEGA-RUIZ, Jose A. ; CURDT, Torsten ; AMETLLER-ESQUERRA, Joan: Continuation-based Mobile Agent Migration / University of Barcelona. http://vafer.org/pub/papers/spasm.pdf, 2006. – Forschungsbericht. unpublished

[Ouellet und Ogbuji 2002] OUELLET, Roxane ; OGBUJI, Uche. *Introduction to DAML: Part I*. http://xml.com/lpt/a/911. January 2002

[Perkins 1998] PERKINS, Charles E.: Service location protocol for mobile users. In: *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC* Bd. 1 IEEE, 1998, S. 141–146

[Pham und Karmouch 1998] PHAM, Vu A. ; KARMOUCH, A.: Mobile Software Agents : An Overview. In: *Communications Magazine* IEEE Volume 36 (1998), July, Nr. 7, S. 26–37

[Pirker u. a. 2004] PIRKER, Michael ; BERGER, Michael ; WATZKE, Michael. *An Approach for FIPA Agent Service Discovery in Mobile Ad Hoc Environments*. http://whitepapers.silicon.com/0,39024759,60149406p-39000341q,00.htm. May 2004

[Pokahr 2007] POKAHR, A.: *Programmiersprachen und Werkzeuge zur Entwicklung verteilter agentenorientierter Softwaresysteme*, Universität Hamburg, Diss., 2007

[Provost 2003] PROVOST, Will. *WSDL First*. http://webservices.xml.com/lpt/a/1250. July 2003

[Prud'hommeaux und Seaborne 2006] PRUD'HOMMEAUX, Eric ; SEABORNE, Andy: *SPARQL Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/. April 2006. – W3C Candidate Recommendation

[Qin und Li 2006] QIN, Donghong ; LI, Zhi: Evaluation and Research of Strong Migration of Mobile Agent for Exploiting Type Inference. In: *qsic* 0 (2006), S. 441–445. – ISSN 1550–6002

[Rivest 1992] RIVEST, R. *RFC1321: The MD5 Message-Digest Algorithm.* http://tools.ietf.org/html/rfc1321. April 1992

[Rotenstreich 2006] ROTENSTREICH, Dr. S. *Introduction to WSDL.* http://www.seas.gwu.edu/ shmuel/cs338/Introduction2006

[Roth 2005] ROTH, Jörg: *Mobile Computing - Grundlagen, Technik, Konzepte.* 2nd. Heidelberg : dpunkt.verlag, 2005. – ISBN 3–89864–165–1

[Rulifson 1969] RULIFSON, J. *RFC0005: Decode Encode Language.* http://portal.acm.org/citation.cfm?id=RFC0005. June 1969

[Russell und Norvig 2003] RUSSELL, Stuart ; NORVIG, Peter: *Artificial Intelligence: A Modern Approach.* 2nd edition. Prentice-Hall, Englewood Cliffs, NJ, 2003

[Sailhan und Issarny 2005] SAILHAN, Francoise ; ISSARNY, Valerie: Scalable Service Discovery for MANET. In: *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications.* Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2299–8, S. 235–244

[Sakamoto u. a. 2000] SAKAMOTO, Takahiro ; SEKIGUCHI, Tatsurou ; YONEZAWA, Akinori: Bytecode Transformation for Portable Thread Migration in Java. In: *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents.* London, UK : Springer-Verlag, 2000. – ISBN 3–540–41052–X, S. 16–28

[Satyanarayanan u. a. 1995] SATYANARAYANAN, M. ; NOBLE, Brian ; KUMAR, Puneet ; PRICE, Morgan: Application-aware adaptation for mobile computing. In: *SIGOPS Oper. Syst. Rev.* 29 (1995), Nr. 1, S. 52–55. – ISSN 0163–5980

[Schilit u. a. 1994] SCHILIT, Bill ; ADAMS, Norman ; WANT, Roy: Context-Aware Computing Applications. In: *IEEE Workshop on Mobile Computing Systems and Applications.* Santa Cruz, CA, US, 1994

[Schön 1998] SCHÖN, Eckhardt: Das Resource Description Framework (RDF) - ein neuer Weg zur Verwaltung von Metadaten im Netz / Technische Universität Ilmenau. 1998. – Forschungsbericht

[Seaborne 2004] SEABORNE, Andy. *RDQL - A Query Language for RDF.* http://www.w3.org/Submission/RDQL/. January 2004

[Sekiguchi u. a. 1999] SEKIGUCHI, Tatsurou ; MASUHARA, Hidehiko ; YONEZAWA, Akinori: A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation. In: *COORDINATION '99: Proceedings of the Third International Conference on Coordination Languages and Models.* London, UK : Springer-Verlag, 1999. – ISBN 3–540–65836–X, S. 211–226

[Shaw und Garlan 1996] SHAW, Mary ; GARLAN, David: *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996. – ISBN 0131829572

[Shiao 2004] SHIAO, Dan. *Mobile Agent: New Model of Intelligent Distributed Computing.* http://www.recursionsw.com. October 2004

[Skonnard 2003] SKONNARD, Aaron: Understanding WSDL / Northface University. 2003. – Forschungsbericht

[SpiegelOnline 2007] SPIEGELONLINE. *Kompletter Computer im Internet-Browser*. http://www.spiegel.de/netzwelt/web/0,1518,482720,00.html. May 2007

[Stamos und Gifford 1990] STAMOS, James W. ; GIFFORD, David K.: Remote evaluation. In: *ACM Trans. Program. Lang. Syst.* 12 (1990), Nr. 4, S. 537–564. – ISSN 0164–0925

[Strasser und Schwehm 1997] STRASSER, Markus ; SCHWEHM, Markus: A Performance Model for Mobile Agent Systems. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* University of Stuttgart, Faculty of Computer Science, 1997, S. 1132–1140

[Suezawa 2000] SUEZAWA, Takashi: Persistent execution state of a Java virtual machine. In: *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. New York, NY, USA : ACM Press, 2000. – ISBN 1–58113–288–3, S. 160–167

[Sun 2001a] SUN, Microsystems: Jini Architecture Specification v. 2.1 / Sun Microsystems, Inc. 2001. – Forschungsbericht

[Sun 2001b] SUN, Microsystems: Jini Technology Core Platform Specification / Sun Microsystems, Inc. 2001. – Forschungsbericht

[Sun 2007a] SUN, Microsystems. *Discover the secrets of the Java Serialization API*. http://java.sun.com/developer/technicalArticles/Programming/serialization/. January 2007a

[Sun 2007b] SUN, Microsystems. *J2ME Personal Profile 1.0*. http://java.sun.com/products/personalprofile/. May 2007b

[Sun 2007c] SUN, Microsystems. *The Java Programming Language*. http://java.sun.com. January 2007c

[Sun 2007d] SUN, Microsystems. *Understanding Network Class Loaders*. http://java.sun.com/developer/technicalArticles/Networking/classloaders/. January 2007d

[Tauberer 2006] TAUBERER, Joshua. *What Is RDF*. http://xml.com/lpt/a/1665. July 2006

[Traversat u. a. 2003] TRAVERSAT, Bernard ; ARORA, Ahkil ; ABDELAZIZ, Mohamed ; DUIGOU, Mike ; HAYWOOD, Carl ; HUGLY, Jean-Christophe ; POUYOUL, Eric ; YEAGER, Bill: Project JXTA 2.0 Super-Peer Virtual Network / Sun Microsystems, Inc. 2003. – Forschungsbericht

[Truyen u. a. 2000] TRUYEN, Eddy ; ROBBEN, Bert ; VANHAUTE, Bart ; CONINX, Tim ; JOOSEN, Wouter ; VERBAETEN, Pierre: Portable Support for Transparent Thread Migration in Java. In: *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*. London, UK : Springer-Verlag, 2000. – ISBN 3–540–41052–X, S. 29–43

[Tsalgatidou und Pilioura 2002] TSALGATIDOU, Aphrodite ; PILIOURA, Thomi: An Overview of Standards and Related Technology in Web Services. In: *Distributed and Parallel Databases* 12 (2002), September, Nr. 2-3, S. 135–162

[Tyan und Mahmoud 2005] TYAN, Jerry ; MAHMOUD, Qusay: A Comprehensive Service Discovery Solution for Mobile Ad Hoc Networks. In: *Mobile Networks and Applications* 10 (2005), January, Nr. 4, S. 423–434. – ISSN 1383–469X

[United Devices 2007] UNITED DEVICES, UD. *grid.org*. http://grid.org/home.htm. March 2007

[UPnP 1999] UPNP, Forum. *Simple Service Discovery Protocol*. http://www.upnp.org/resources/specifications.asp. October 1999

[UPnP 2000] UPNP, Forum. *Understanding Universal Plug and Play: A White Paper*. http://studies.ac.upc.edu/EPSC/FSD/UnderstandingUPNP.pdf. June 2000

[UPnP 2003] UPNP, Forum. *Universal Plug and Play Device Architecture Version 1.0.1*. http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf. December 2003

[Vigna 1998] VIGNA, Giovanni (Hrsg.): *Mobile Agents and Security*. Bd. *1419*. Springer, 1998 (Lecture Notes in Computer Science). – ISBN 3–540–64792–9

[Vigna 2004] VIGNA, Giovanni: Mobile Agents: Ten Reasons For Failure. In: *mdm* 00 (2004), S. 298. ISBN 0–7695–2070–7

[W3C 1998] W3C. *Notation 3*. http://www.w3.org/DesignIssues/Notation3. March 1998

[W3C 2001] W3C. *Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/2001/NOTE-wsdl-20010315. March 2001

[W3C 2003] W3C. *Simple Object Access Protocol (SOAP) Version 1.2*. http://www.w3.org/TR/soap12-part1/. June 2003

[W3C 2004a] W3C. *OWL Web Ontology Language Overview*. http://www.w3.org/TR/owl-features/. February 2004a

[W3C 2004b] W3C. *OWL Web Ontology Language Semantics and Abstract Syntax*. http://www.w3.org/TR/owl-semantics/. February 2004b

[W3C 2004c] W3C. *OWL Web Ontology Language Use Cases and Requirements*. http://www.w3.org/TR/webont-req/. February 2004c

[W3C 2004d] W3C. *RDF Concepts and Abstract Syntax*. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. February 2004d

[W3C 2004e] W3C. *RDF Primer*. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/. February 2004e

[W3C 2004f] W3C. *RDF Vocabulary Description Language 1.0: RDF Schema*. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/. February 2004f

[Weiss 1999] WEISS, Gerhard (Hrsg.): *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Cambridge, MA, USA : MIT Press, 1999

[Weyns u. a. 2004] WEYNS, Danny ; PARUNAK, H. Van D. ; MICHEL, Fabien ; HOLVOET, Tom ; FERBER, Jacques: Environments for Multiagent Systems State-of-the-Art and Research Challenges. In: *E4MAS*, 2004, S. 1–47

[Weyns u. a. 2005] WEYNS, Danny ; SCHUMACHER, Michael ; RICCI, Alessandro ; VIROLI, Mirko ; HOLVOET, Tom: Environments in multiagent systems. In: *The Knowledge Engineering Review* 20 (2005), June, Nr. 2, S. 127–141. – ISSN 0269–8889

[Weyns u. a. 2004] WEYNS, Danny ; STEEGMANS, Elke ; HOLVOET, Tom: Towards Active Perception In Situated Multi-Agent Systems. In: *Applied Artificial Intelligence* 18 (2004), Nr. 9-10, S. 867–883

[Weyns u. a. 2005] WEYNS, Danny ; VIZZARI, Giuseppe ; HOLVOET, Tom: Environments for Situated Multi-agent Systems: Beyond Infrastructure. In: *E4MAS* Bd. 3830, Springer, 2005, S. 101–117

[White 1997] WHITE, James E.: Mobile Agents. In: BRADSHAW, Jeffrey M. (Hrsg.): *Software Agents*. AAAI Press / The MIT Press, 1997, Kapitel 19, S. 437–472

[White 1999] WHITE, James E.: Telescript technology: mobile agents. In: *Mobility: processes, computers, and agents* 1 (1999), S. 460–493. ISBN 0–201–37928–7

[Wikipedia 2006a] WIKIPEDIA. *Ad hoc routing protocol list*. http://en.wikipedia.org. September 2006a

[Wikipedia 2006b] WIKIPEDIA. *Agent Environment*. http://en.wikipedia.org. July 2006b

[Wikipedia 2006c] WIKIPEDIA. *Awareness (DE)*. http://de.wikipedia.org/wiki/Awareness. September 2006c

[Wikipedia 2006d] WIKIPEDIA. *Awareness (EN)*. http://en.wikipedia.org/wiki/Awareness. September 2006d

[Wikipedia 2006e] WIKIPEDIA. *Push*. http://de.wikipedia.org/wiki/Push. November 2006e

[Wikipedia 2006f] WIKIPEDIA. *Umwelt*. http://de.wikipedia.org/wiki/Umwelt. July 2006f

[Wikipedia 2007a] WIKIPEDIA. *Mandelbrot Set*. http://en.wikipedia.org/wiki/Mandelbrot. March 2007a

[Wikipedia 2007b] WIKIPEDIA. *Quality of Service*. http://de.wikipedia.org/wiki/Qos. March 2007b

[Wiklander 2001] WIKLANDER, Erik: *Mobile Resource Awareness*, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Diplomarbeit, February 2001

[Wong u. a. 1999] WONG, David ; PACIOREK, Noemi ; MOORE, Dana: Java-based mobile agents. In: *Commun. ACM* 42 (1999), Nr. 3, S. 92–ff.. – ISSN 0001–0782

[Wooldridge und Jennings 1995] WOOLDRIDGE, Michael ; JENNINGS, Nicholas R.: Intelligent Agents: Theory and Practice. In: *Knowledge Engineering Review* 10 (1995), Nr. 2, S. 115–152

[Wooldridge 2001] WOOLDRIDGE, Michael J.: *Introduction to Multiagent Systems*. New York, NY, USA : John Wiley & Sons, Inc., 2001. – ISBN 047149691X

[Wu und Zitterbart 2001] WU, Jidong ; ZITTERBART, Martina: Service Awareness in Mobile Ad Hoc Networks. In: KILLAT, Ulrich (Hrsg.) ; LAMERSDORF, Winfried (Hrsg.): *Kommunikation in Verteilten Systemen*, Springer, February 2001 (Informatik Aktuell), S. 69–80

[Xie u. a. 2002] XIE, Jason ; TALPADE, Rajesh R. ; MCAULEY, Anthony ; LIU, Mingyan: AMRoute: ad hoc multicast routing protocol. In: *Mob. Netw. Appl.* 7 (2002), December, Nr. 6, S. 429–439. – ISSN 1383–469X

[Zambonelli u. a. 2003] ZAMBONELLI, Franco ; JENNINGS, Nicholas R. ; WOOLDRIDGE, Michael: Developing multiagent systems: The Gaia methodology. In: *ACM Trans. Softw. Eng. Methodol.* 12 (2003), Nr. 3, S. 317–370. – ISSN 1049–331X

[Zhu u. a. 2005] ZHU, Changzheng ; YIN, Zhaolin ; ZHANG, Aijuan: Mobile Code Security on Destination Platform. In: *ICCNMC*, 2005, S. 1263–1270

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.
Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments einverstanden.

Hamburg, den __________ Unterschrift: _________________________