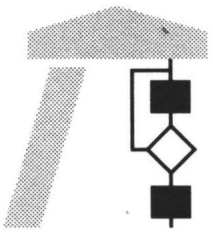
# Interner Bericht

## Mobile Agenten im Internet

## Infrastruktur und Interaktion

Friedrich van Megen

314/2001



FACHBEREICH
INFORMATIK

UNIVERSITÄT
KAISERSLAUTERN

Postfach 3049 · D-67653 Kaiserslautern

# Mobile Agenten im Internet
# Infrastruktur und Interaktion

Friedrich van Megen
Universität Kaiserslautern
AG ICSY
Bau 36 Raum 407
67663 Kaiserslautern

mailto:vanmegen@informatik.uni-kl.de

Mobile Agenten haben sich in den letzten Jahren zunehmend in der Architektur und Programmierung verteilter Systeme bewährt. Es sind Programme, die einen Internen Zustand mit sich führen, während sie verschiedene, möglicherweise auf unterschiedlichen Plattformen basierende, Systeme besuchen. Auf dem jeweiligen System nehmen sie Dienste in Anspruch, indem sie entweder lokale Bibliotheken ansprechen, oder auf durch das System bereitgestellte Dienste zugreifen. Dabei müssen mobile Agenten sowohl alle vom Programm benötigten Daten, wie auch den gesamten Code mit sich führen. Zwar sind die Daten ein wichtiger (wenn nicht sogar der entscheidende) Teil eines Agenten, trotzdem wird in der Regel nicht als wertvoller, eigenständiger Part angesehen. Dies ist jedoch nicht immer ratsam, könnten doch Agenten am aktuellen Aufenthaltsort einen „Container" zurückzulassen um ihm anderen Agenten zur Verfügung zu stellen (natürlich erst nach erfolgter Zugriffskontrolle), bzw. die Daten erst dann auf ein Migrationsziel übertragen, wenn sich durch lokale Aufrufe des Systems herausgestellt hat, dass sie dort benötigt werden.

Diese Arbeit ist zweigeteilt, insofern, als dass sie sich mit den zwei verschiedenen „Ebenen" der mobilen Agenten beschäftigt. Im ersten Teil werden die für die Migration und Nutzung der Resourcen notwendigen Aspekte besprochen. Dabei wird der Schwerpunkt auf die notwendige Unterstützung durch die Umgebung gelegt, wobei nicht eine neue integrierte Umgebung entworfen, sondern vielmehr die notwendigen Blöcke aufgezeigt werden sollen. Diese können dann als Teil eines Environments oder aber als eigentständige Komponente bereitsgestellt werden. Der zweite Teil beschäftigt sich mit den durch die Interaktion verschiedener Agenten entstehenden Probleme. Stichworte hierbei sind die Kostenkontrolle (wer bezahlt auf welche Art für in Anspruch genommene Dienste), Workflow Unterstützung, sowie Sicherheit in einem offenen, verteilten System, in dem es keine zentrale Überprüfung von Rechten und Identitäten geben kann.

Abgeschlossen wird diese Ausarbeitung mit einer Bewertung der auf den beiden Ebenen gefundenen Problemen und Eigenheiten, wobei dann die Frage aufgeworfen wird, ob Agenten in der heutigen Form überhaupt sinnvoll sind.

# Inhaltsverzeichnis:

# 1. Einleitung

Diese Arbeit bewegt sich auf dem mit einem Alter mehr als 15 Jahren immer noch jungen Forschungsgebiet der mobilen Agenten. Ein mobiler Agent ist ein Programm, das in einem, oftmals heterogenem, Netzwerk verschiedener Rechner und Betriebssystemen selbständig zwischen ansonsten unverwandten Systemen migrieren kann, um eine Aufgabe des Besitzers auszuführen, oder selbständig Daten zu sammeln, auszuwerten und sich auf Grund der gefundenen Daten neue Ziele zu setzen. Dazu führt ein mobiler Agent in der Regel seinen gesamten Code, sowie alle notwendigen Daten mit sich, um auf dem Rechner, auf dem er sich gerade befindet, lokale Dienste in Anspruch zu nehmen, über diesen Rechner entfernte Dienste anderer Agenten in Anspruch zu nehmen, lokale Dienste anzubieten, oder auch entfernt ansprechbare Dienste zu veröffentlichen. Dies alles natürlich nur unter der Voraussetzung, dass der Agent über die entsprechenden Rechte verfügt und sich gegenüber dem System ausweisen kann.

Mobile Agenten als solches, sind kein neues Konzept in der Informatik. Vielmehr führen sie Ansätze und Methoden, die schon in anderen Forschungsgebieten bekannt sind, zusammen und bieten erst durch die Implementierung dieser Konzepte in einer homogenen Umgebung eine Grundlage für neue Lösungsansätze. Der Begriff des Agenten kommt aus der künstlichen Intelligenz. Dort beschäftigt man sich mehr mit der Planung und der Kooperation verteilt arbeitender Einheiten, die nicht notwendigerweise zum Zeitpunkt der Instanzzierung schon alle anderen Instanzen kennen, ja nicht einmal von der genauen Bedeutung der „Sprache" anderer Agenten wissen. Hier geht es um die Kooperation und das Verständnis potentiell unbekannter Individuen. Agenten handeln nach dem Grundsatz von Believe, Desire, Intention (BDI). Ausgehend von ihrem Bild der Umwelt (Believe) versuchen sie ihnen gestellte Aufgabe zu erfüllen (Desire), indem sie die Umwelt gezielt ändern (Intention). Dabei ist es nebensächlich, ob ein Agent seinen physischen Ort verlässt, oder nicht [1]. In der Softwareentwicklung kennt man Agenten als „aktive Objekte", wobei damit in der Regel ein anderer Name für ein bestehendes Konzept gewählt wird. So ist nach dieser Definition potentiell jeder Server, bzw. jeder Dienst ein Agent, wenn er über einen internen Zustand und (mindestens) eine externe Schnittstelle verfügt. Im Forschungsgebiet der verteilten Systeme, definiert man über Agenten die Komponenten in einem verteilten System, die miteinander kommunizieren. Damit sind diese Objekte eine andere Abstraktion des Begriffs „Service" bzw. „Server". Mobile Agenten schließlich sind eine Untergruppe der Agenten, die sich durch ihre Fähigkeit auszeichnen, den Instanziierungsort physisch verlassen zu können. Dazu müssen sowohl der Code, wie auch die referenzierten Daten zwischen den bei der Migration beteiligten Rechnern übertragen werden. Zusätzlich muss, wenn Starke Migration bereitgestellt wird, der Zustand des (virtuellen) Prozessors verpackt und am Zielort wieder restauriert werden.

Im folgenden Kapitel wird eine Einführung in das Forschungsgebiet der Mobilen Agenten gegeben und dabei eine Abgrenzung zu den verwandten Gebieten der verteilten System, mobilen Systeme und Cluster gegeben werden.

Das dritte Kapitel beschäftigt sich mit den Anforderungen mobiler Agenten an die bereitzustellende Infrastruktur. Dabei werden Ressourcenfragen, Konfiguration und Lokationsmeldung besonders herausgestellt werden.

Das vierte Kapitel behandelt Probleme, die sich bei der Nutzung der Infrastruktur ergeben. Hierbei werden exemplarisch Probleme aufgezeigt, die sich aus Workflow-Unterstützung, Bezahlung von in Anspruch genommenen Diensten, sowie die Sicherheitsprobleme die sich aus der Übertragung von Code und Daten auf nicht vertrauenswürdigen Systemen ergeben.
Es schließt mit einer subjektiven Wertung die Ergebnisse der beiden vorangegangenen Kapitel.

# 2. Mobile Agenten – Definitionen und Beispiele

## 2.1. Einführung

Agenten werden in der Informatik in unterschiedlichen Zusammenhängen mit verschiedener Bedeutung verwendet.

In der Informatik trat der Agent zuerst im Forschungsgebiet der künstlichen Intelligenz in Erscheinung, und das schon Ende der 70er Jahre in den Arbeiten von Carl Hewitt geht aber schon auf Arbeiten aus den 50er und 60er Jahren zurück [2].
Carl Hewitts Concurrent Actor Model ([3] nach [4]) gilt als Ausgangspunkt der Forschungen zu Software Agenten. Er beschreibt ein unabhängiges, interaktives und parallel ablaufendes Objekt, welches als "Actor" bezeichnet wird. Es hat einen von der Umwelt abgeschirmten Zustand und kann mit anderen Objekten kommunizieren.

Ein Actor: "is a computational agent which has a mail address and a behaviour. Actors communicate by message-passing and carry out their actions concurrently." [3]

In der Künstlichen Intelligenz bezeichnet ein Agent eine Software, die in der Lage ist, mit ihrer Umwelt zu kommunizieren und daher selbstständig neues Wissen zu erarbeiten. Dies geschieht über eine Lernschleife, genauer, der Aufnahme von externen Eindrücken, der Berechnung der nächsten Schritte auf Grund der Eindrücke und des auszuführenden Auftrags und schließlich dem Anstoßen von externen Methoden zur Änderung der momentanen Umwelt (BDI Modell) um das gewünschte Ziel zu erreichen. In der künstlichen Intelligenz ist der Aspekt des „mobil sein" nicht die entscheidende Eigenschaft eines Agenten. Agenten müssen sich nicht physisch zwischen Knoten bewegen. Vielmehr ist hier das Hauptaugenmerk auf die „soziale Komponente" gelegt, die sich in der Planung und Zusammenarbeit vieler Agenten miteinander ausdrückt. Dazu wurden eigens Sprachen entwickelt, die auch einander vorher völlig unbekannten Agenten die Kommunikation ihrer Wünsche, Ergebnisse und Eindrücke ihrer Welt ermöglichen [5] [6].

Woher kommt ursprünglich der Begriff des Agenten? Das aus dem lateinischen stammende Wort steht im deutschen für den Vertreter bzw. Handelnde; für den Makler und Bevollmächtigten. Im englischen hat es darüber hinaus noch die Bedeutung des Maklers, der Güter anderer transportiert bzw. Im Auftrag anderer handelt (Häuser, Grundstücke, Einkauf). Einen Agenten zeichnen im allgemeinen folgende Eigenschaften [7]:

- The autonomy and persistence of the agent and its ability to initiate actions or work by itself
- That agents are essentially bounded entities with limited capabilities and restricted resources
- That agents need ways of observing and manipulating the world, as well as communication with other agents
- That agents may be viewed as „black boxes": knowledge of their intimate workings is neither necessary or desirable for their classification
- That agents have goals (with may be modified by the actions of other agents or the environment) and methods or achieve them
- How the definition of a software agent relates to that of a human agent
- How viewing the agent and the various roles or may play affects its classification

Dabei lassen sich folgende Konzepte genauer Klassifizieren:

Autonomie: Agenten sind nicht im Ganzen durch den Besitzer vorbestimmt. Sie können in gewissem Rahmen selbständig handeln ohne auf eine ständige physische Verbindung zum Erzeuger angewiesen zu sein.

Persistenz: Agenten müssen ihren eigenen Zustand mit sich führen um sicherzustellen, dass gesammelte Daten nicht gelöscht werden. Dennoch sind sie auf die Hilfe des Agentensystems für eine dauerhafte Speicherung ihrer Daten angewiesen, haben sie doch selbst keinen direkten Zugriff auf das unter dem Agentensystem liegende Betriebssystem (falls es überhaupt ein Host Betriebssystem gibt [8]).

Unabhängigkeit: Agenten sind in der Regel auf sich selbst gestellt. Daher müssen sie alle benötigten Ressourcen ständig mit sich führen, da ja zu keinem Zeitpunkt gesichert ist, dass sie auf eventuell zurückgelassene Daten wieder zugreifen können.

Interaktion: Agenten arbeiten nicht in einer statischen Welt, in der die Umgebung nur von dem Handeln der Agenten selbst abhängt. Vielmehr müssen sie selbst auf Änderungen des sie umgebenden Systems reagieren, wobei Änderungen jederzeit, asynchron für den Agenten auftreten können. Agenten sollen auf die Umwelt reagieren. Dazu muss das Agentensystem dem Agenten eine Sicht auf die Umwelt ermöglichen, bzw. Änderungen der aktuellen Umgebung dem Agenten mitteilen. Dazu werden in der Regel asynchrone Mitteilungen (Events) genutzt [9] [10] [11].

Sicherheit: Dieser Aspekt beschreibt den Schutz zwischen Agenten Agentensystem. Es muss einerseits sichergestellt werden, dass ein Agent und seine Daten wirklich „echt" sind, bevor man ihm als Agentensystem die Ausführung auf dem lokalen Rechner gestattet. Dies kann relativ einfach über signierten Code und explizit geschützten Daten erreicht werden. Dabei geht das Agentensystem davon aus, dass der Agent selbst für die Sicherung seiner Daten während der Übertragung sorgt, bzw. das sendende Agentensystem diese Aufgabe für ihn während der Übertagung übernimmt. Im letzteren Fall müssen sich aber die beteiligten Agentensysteme kennen und vertrauen.
Andererseits muss ein Agent vor den neugierigen Blicken eines Agentensystems geschützt werden, das ja zu jeder Zeit „in den Agenten hineinschauen" kann und dadurch an sensitive Daten gelangen kann. Dieser Schutz ist nur sehr viel schwieriger zu realisieren, muss doch ein Agentensystem Code und Daten lesen dürfen, um ihn ausführen zu können.

Mobil: Agenten können sich „frei" bewegen und sowohl Daten, wie auch Code dynamisch zwischen physischen Systemen transportieren und damit auf eine geeignete Art ihre Arbeit auf dem Zielsystem fortsetzen.

Intelligent: Bedeutet für Agenten, dass sie sich nicht eine feste Strategie „blind" ausführen. Vielmehr muss ein Agent adaptiv auf die Umgebung reagieren, erwartete Veränderungen in seine Planung aufnehmen und neue Optionen erkennen und nutzen.

Andere Autoren halten dies für die entscheidenden Eigenschaften eines Agenten [12]:

Autonomie: Die Autoren definieren Autonomie als die Möglichkeit des Agenten selbständig zu handeln, ohne jeden Aspekt ihrer Handlung vom Besitzer vorgegeben zu bekommen.

Soziales Verhalten: Wird über eine gemeinsame Sprache der Agenten untereinander ermöglicht. Es wird besonderer Wert darauf gelegt, dass sie zum einen generisch und einfach erweiterbar ist, zum anderen so mächtig, dass auch komplexe Dinge ausgedrückt werden können. Für diesen Aspekt wurden Agent Communication Languages (ACL) entwickelt [13], speziell KQML [5] [14]. Die heutige Forschung geht in Richtung von „ontology markup languages", vergleiche [15].

Reaktivität: Ist mit „Asynchron" vergleichbar und meint, dass der Agent auf externe Ereignisse reagieren muss. Dies sind in der Regel andere Agenten bzw. Meldungen des (lokalen) Agentensystems. Ein Agent nimmt diese Reize seiner Umwelt auf und nutzt sie für seine weitere Planung.

Proaktivität: Beschreibt die Möglichkeit eines Agenten, selbständig zu planen, um die ihm gestellte Aufgabe zu erfüllen. Da sein Besitzer in der Regel nicht mit ihm in ständigem Kontakt steht, muss der Agent selbst die Initiative ergreifen. [16]. Dieses Eigenschaft wird in der künstlichen Intelligenz erforscht, indem Komponenten der Planung, Zielrichtung und Adaption zusammengeführt werden [1].

Innerhalb der künstlichen Intelligenz stellen Agenten in der elektronischen Umgebung quasi den Ersatz für den Menschen dar, ein Subjekt, das selbständig „denkt" und handelt. Man kann ihnen Aufträge geben, ohne sich um die konkrete Realisierung kümmern zu müssen. Dabei nehmen sie das Wissen ihres jeweiligen Besitzers als Grundlage ihrer Planung um dessen Wünschen möglichst zu entsprechen (in der KI [17], allgemein [16] [4]). Im Unterschied hierzu definiert ein verteiltes System die Grundlagen, die Agenten erst die Kommunikation mit anderen ermöglicht und ist daher unterhalb der von der KI definierten Softwareschicht einzuordnen. Es implementiert Kommunikationsprimitive, Sicherungsfunktionen, sowie Schnittstellen, die es den Komponenten erlauben, von Anderen angebotene Dienste zu nutzen. Oft wird Wert auf die physische Mobilität einer Komponente gelegt, auch wenn dies kein zwingendes Attribut eines verteilten Systems ist. Dies sieht man zum Beispiel an Amoeba, wo die Möglichkeit der Migration auf Prozessebene erst vor kurzem implementiert wurde [18]. Bis dahin wurde der Ausführungsort eines Prozesses nur bei dessen Erzeugung, transparent für den Anwender, durch das System selbst festgelegt. Oft meint verteilt nur, dass wir uns in einem, meist heterogenen, Umfeld befinden, in dem Dienste auf standardisierte Weise gefunden und angesprochen werden können. In diesem Zusammenhang ist Mobilität nur insofern von Bedeutung, als dass es zur Erhöhung der Redundanz und damit zur Reduzierung der Ausfallwahrscheinlichkeit eingesetzt wird. Mobilität als ausgezeichnete Eigenschaft wird selten gefunden.

Die Mobilität einer Komponente kann auf unterschiedliche Art erreicht werden. Dies kann sowohl mit, wie auch ohne die Migration des eigenen Codes geschehen:

Parametrisierte entfernte Ausführung: Komponenten besitzen Referenzen auf entfernte Dienste, rufen diese mit jeweils angepassten Daten auf. [19]. Diese Dienste können durch das System selbst bereitgestellt werden, Komponenten selbst implementieren jedoch keine Dienste die andere Komponenten nutzen könnten.

Statische Agenten: Im Gegensatz zur parametrisierten entfernten Ausführung, bei der Komponenten nur existierende Dienste nutzen können, können statische Komponenten auch selbst Dienste anbieten. [20] [21].

Entfernte Ausführung beliebigen Codes: Komponenten, die sich dieses Mittels bedienen, enthalten Code, nicht notwendigerweise in der gleichen Sprache wie die Komponente selbst verfasst, den sie auf einen entfernten Rechner transferieren, um ihn dort lokal evaluieren zu lassen. Die Umgebung muss hierfür nur die notwendigen Schnittstellen für die Übertagung des Codes bereitstellen. Was sie nicht muss, ist eine vollständige Umgebung für die Ursprungskomponente zu implementieren. Klassische Beispiele hierfür sind die Druckersprachen Postscript und PCL, welche die Mächtigkeit einer vollständigen Programmiersprache haben [22] [23]. Die auf dem Lokalen Rechner laufenden Programme senden den Druckauftrag als ausführbaren Code zum Drucker und lassen ihn dann dort, lokal ausführen, da ja erst auf dem Drucker selbst die gewünschte Funktionalität (Druckwerk) zur Verfügung steht.

Migrierende Komponenten: Diese Komponenten implementieren eine Mischung aus 2. und 3. Das Besondere ist, dass sie einen beliebigen Code, nämlich sich selbst, auf das Zielsystem bringen können, um ihn dort auszuführen.

Bei dieser Aufstellung wird suggeriert, dass Komponenten sich jeweils über eine funktionale Schnittstelle verständigen. Das ist jedoch nicht zwangsweise der Fall. So ist es durchaus vorstellbar, dass die Kommunikation allein über den Austausch von Nachrichten implemen-

tiert wird [9]. Der erste Ansatz mobiler Agenten im Web-Umfeld waren Applets, die ihren Code auf ein in der Regel unbekanntes System transferieren um dort eine Aufgabe zu erfüllen. Dies wird durch Java ermöglicht, das eine standardisierte Umgebung bietet, wodurch auf jedem Zielsystem wieder ein bekanntes Environment vorausgesetzt werden kann [24]. Im Vergleich zu einem verteilten System, in dem die Knoten fest über mehrere Orte verteilt sind und sich die darauf laufenden Komponenten frei bewegen können, sind bei „mobile computing" die Knoten selbst nicht mehr physisch an einen Ort gebunden. Das Konzept eines mobilen Agenten bekommt hierdurch eine neue Qualität. Zum einen müssen die Knoten selbst auf Migration vorbereitet sein. Zum anderen müssen Agenten in einer derartigen Umgebung mit dem (zumindest zeitweisen) Abbruch der Verbindung zum Restsystem rechnen. Spätestens hier wird deutlich, warum es Ziel sein muss, den kompletten Zustand eines Agenten von einem System auf ein anderes System zu transportieren: Ein Agent kann seinen Auftrag nicht mehr ausführen, wenn er plötzlich keinen Zugriff mehr auf seine Daten hat. Erste Beispiele für mobile Systeme werden im Zusammenhang mit der Kommunikation in Mobilfunksystemen implementiert. Exemplarisch sei das Suchen von Diensten in der näheren Umgebung genannt. Hierbei muss der Agent nicht nur die Wünsche des Auftraggebers berücksichtigen, sondern auch noch dessen aktuellen Standort (der sich während der Lebenszeit des Agenten durchaus mehrfach ändern kann).

Wie man aus diesen wenigen Beispielen sehen kann, ist der Begriff des Agenten häufig schon vorbelegt, und das, je nach Gebiet auf dem man sich befindet, unterschiedlich. Dadurch wird es schwierig, die einzelnen Ergebnisse und Methoden der verschiedenen Forschungsrichtungen in Verbindung zu bringen und miteinander zu vergleichen.

Im weiteren Kapitel werden mobile Komponenten/Agenten im Kontext eines verteilten Systems, sowie eines Mobilen Systems vorgestellt. Abschließend werden die verschiedenen Konzepte der Mobilität von Daten und Code aufgezeigt, bewertet und schließlich Gründe für die Nützlichkeit mobiler Agenten gegeben werden.

## 2.2. Verteilte Systeme

Bis Anfang der 90er Jahre [25] zielte der Entwurf verteilter Systeme darauf ab, die Verteilung der physischen Ressourcen und deren Organisation vor Entwicklern und Anwendern so gut wie möglich zu verstecken. Ziel war es, das System als ganzes homogen erscheinen zu lassen. Dabei geschieht folgendes. Die Betriebsmittel eines lokalen Knoten werden repräsentiert durch abstrakte Ressourcen. Durch diese Schicht halten Programme keine Referenzen mehr auf physische Objekte und Objekte können über Knotengrenzen hinweg referenziert werden. Dies gilt selbst für Ressourcen, wie CPU und physischen Speicher, die sich nicht real auf andere Knoten übertragen lassen.
Sicherheit wird durch eine übergeordnete Instanz bereitgestellt. Dies kann eine eigene Entwicklung sein (Capabilities bei Amoeba), auf bereits bestehende Systeme aufbauen (wie beispielsweise Kerberos), auf externe Hardware zurückgreifen, oder durch die Wahl der Repräsentierung implizit implementiert werden. Letzteres gilt beispielsweise für die asymmetrische Verschlüsselung, bei der ein Client auch ohne die Hilfe eines externen Sicherungsdienstes entscheiden kann, ob die gegebenen Informationen verändert worden sind und sich externe Sicherheit auf das sichere Verteilen der öffentlichen Schlüssel beschränkt [26]. Kommunikation der dann auf den verschiedenen Knoten ablaufenden Prozesse wird über eine implementierte Distributed Shared Memory Architektur der Systeme möglich, oder aber mittels klassischer Kommunikationsmittel, wie sie RPC, Messages und Signale bieten. Gemeinsamer Speicher hat den Vorteil, dass Programmierer Algorithmen einfacher umsetzen können, da sie sich nicht um die Kommunikation der Systeme untereinander kümmern müssen, während die expliziten Kommunikationsprimitive oft eine bessere Performanz durch die Ausnutzung weiterer Informationen bezüglich der Lokalität der Daten, erreichen. Schließlich muss man noch die Synchronisation erwähnen. Hier liegt die Schwierigkeit auf der verteilten, performanten Deadlockerkennung.

Einen Überblick über verteilte Systeme findet man in [25]. Neuere Beispiele sind Amoeba [27], Panda [28], bzw. [29]. Objekte können in einem derartigen System frei verteilt werden und referenzieren Ressourcen ortstransparent. Stellvertreter werden an den Stellen, an denen externe Komponenten Zugriffspunkte erwarten, in der Regel durch das Betriebssystem selbst realisiert. Dadurch, dass die Objekte nicht selbst neue Kommunikationspunkte definieren können, sind sie auf die durch das Betriebssystem bereitgestellten Kommunikationsprimitive beschränkt. Dies ändert sich in einem Agentensystem mit mobilen Agenten da diese einen Knoten selbständig wieder verlassen können, sie jedoch Kommunikationsendpunkte zurücklassen müssen, damit andere Agenten sie weiterhin erreichen können. Diese Kommunikationsendpunkte können nicht mehr durch das (verteilte) Betriebssystem implementiert werden, da die Schnittstellen nicht zum Zeitpunkt des Designs des Betriebsystems feststehen, sondern später, dynamisch, definiert werden ([21], oder auch bei jini [30]). Derartige Agenten müssen in einen Ortsdienst eingetragen werden, in dem ihr aktueller Aufenthaltsort jeweils vermerkt ist. Andere Objekte, die während ihrer Ausführung auf eine Tote Referenz stoßen (weil sie eine veraltete Referenz auf das Objekt haben) sind gezwungen, über den Ortsdienst den aktuellen Aufenthaltsort des Agenten erneut aufzulösen und die eigene Referenz entsprechend anzupassen. Hier sei stellvertretend als Beispiel für ein durch das Betriebssystem bereitgestelltes Kommunikationsprimitiv das Port Konzept in Amoeba vorgestellt, das es erlaubt die Kommunikationsendpunkte von Diensten zwischen Knoten migrieren zu lassen. Dazu besteht der eigentliche Port aus zwei Zugriffspunkten, einem öffentlichen, an den man Nachrichten als Klient schicken kann, sowie einem privaten, zu dem man sich als Server binden muss, um eingehende Nachrichten abzuholen. Der private Leseport lässt sich nicht aus dem öffentlichen Schreibport ableiten. Um nun zwischen den Systemen migrieren zu können ohne dass sich der Port verändert, erfordert das Betriebssystem Unterstützung von der Hardware in Form eines Filters für die aktiven Ports. Diesem übergibt das Betriebssystem eine Liste der Adressen der gerade auf diesem Knoten erreichbaren Ports. Es ist die Aufgabe der Netzwerkkarte, die zu diesen Ports gehörenden Pakete zu empfangen. Dadurch „merkt" ein Klient nicht, wenn seine Anfragen von einer anderen Netzwerkkarte (die Netzwerkkarte des Rechners, auf dem sich der Leseport gerade befindet) abgenommen werden und Antworten über diese Netzwerkkarte kommen [27]. Obwohl die Ports selbst Orttransparent, und damit mobil, sind, gilt dies bei frühen Versionen von Amoeba nicht für die Prozesse. Erst in neuerer Zeit wurde es ermöglicht, nicht nur zur Erzeugungszeit eines Prozesses dessen Host zu wählen, sondern ihn auch zur Laufzeit zu ändern. [18].

Zusammenfassend kann man festhalten, dass bei verteilten Systemen die Anstrengung darin besteht, alle von einem Programm (vergleichbar einem Agenten) benötigten Zugriffe auf externe Dienste und Ressourcen, wie auch andere Komponenten so zu abstrahieren, dass man den Dienst selbst zu einem beliebigen Zeitpunkt migrieren lassen kann. Dies ist möglich, da ein Dienst von nichts mehr direkt abhängig ist und keine andere Komponente direkt vom Dienst abhängig ist. Obwohl vielversprechend, haben sich derartige Systeme bisher nicht auf breiter Front durchgesetzt. Dies mag zum einen am Zwang nach einer homogenen Betriebssystem Umgebung liegen, zum anderen in der (falschen) Annahme, man könnte die Probleme die sich durch die Einführung eines Netzwerkes ergeben würden, vor den eigentlichen Prozessen vollständig verbergen. Es hat sich gezeigt, dass es nicht möglich ist, das Netz vollständig transparent für den Anwender bzw. Programmierer zu halten. Zu vielfältig sind die möglichen Fehlerquellen, zu unterschiedlich die je nach Situation notwendigen Korrekturen. Daher erscheint es sinnvoller, diese Aufgabe nicht im Betriebssystem selbst, sondern in einer Schicht außerhalb des Betriebssystems, innerhalb des Benutzerkontextes zu behandeln. Möglicherweise liegt die fehlende Relevanz derartiger Systeme auch daran, dass ein verteiltes Betriebssystem aus Sicht möglicher Klienten zu beschränkt ist. In der Regel umfasst es nur verhältnismäßig wenige Knoten, die meist über ein lokales Netz verbunden sind. Gewünscht ist hingegen die Verteilung der Komponenten bzw. die Mobilität der Ressourcen innerhalb eines sehr viel größeren Netzes. Aktuelle Forschungen auf diesem Gebiet führen zu Globe, welches explizit für den Einsatz in Weitverkehrsnetzen entworfen wurde [31].

Eine zweite Strategie zur Verteilung der Komponenten ist es, die Verteilung nicht mehr im Betriebssystem selbst, sondern durch eine geeignete Library auf Userebene zu implementieren. Der Unterschied ist hier, dass nicht das Programm migriert; vielmehr ist es durch geeignete Methoden möglich, die im entfernten Systemen laufenden Komponenten transparent anzusprechen. Frühe Ansätze sind der RPC Mechanismus, bei dem die Komponenten eines Knoten bei einem lokalen „mapper" angemeldet werden. Dieser Mapper stellt, vergleichbar einem Namensdienst und Lokationsdienst, diese Informationen externen Klienten zur Verfügung [19]. Dazu muss ein potentieller Klient zwei Dinge besitzen, nämlich eine Referenz (die „Adresse") des Servers, sowie eine Klientenseitige Repräsentierung des Serverobjekts. Letztere besitzt eine Funktionalität ähnlich den CORBA Client Stubs und lässt sich aus der Schnittstellenbeschreibung generieren. Die Funktionalität des Portmappers ist vergleichbar mit den Aufgaben der Object Adapter (BOA, POA) in CORBA. Unter dem Paradigma der objektorientierten Programmierung, seien hier speziell CORBA und JavaRMI erwähnt, die beide auf beliebigen Knoten gestartete Komponenten erlauben. Klienten und Komponenten kommunizieren ungeachtet von Prozess- und Rechnergrenzen miteinander [20] [21]. Hier besteht die Kommunikationsaufnahme aus zwei Teilen. Dies sind die Adresse des Partners, sowie einem lokalen Stellvertreter für das Serverobjekt. Wie beim RPC ist es Aufgabe der Anwendung, die Adresse des Partners zu finden, während der lokale Stellvertreter aus der Schnittstellenbeschreibung generierbar ist. Was auch diesem Ansatz fehlt, ist die Migration der Komponenten während sie im System aktiv sind. Zwar ist es möglich, auf jedem Knoten auf dem die entsprechende Komponente installiert ist, eine Instanz zu erzeugen, doch gibt es keine standardisierte Möglichkeit diesen Ort zur Laufzeit für angemeldete Klienten transparent zu verlassen. CORBA entspricht diesem Wunsch am ehesten, erlaubt doch das interne IIOP Protokoll eine LocationForward Antwort, die dem Klienten den neuen Aufenthaltsort mitteilt. Da dies standardisiert ist [32], ist es zumindest theoretisch möglich, den Code auf einen anderen Knoten zu verlagern. Zusätzlich definiert der Corba LifeCycle Service die auf Anwendungsebene zu implementierenden Schnittstellen, mit deren Hilfe ein standardisierter Übergang auf einen anderen Knoten ermöglich wird. Da aber eine Möglichkeit fehlt, einen „Forwarder" zurückzulassen, der etwaigen Interessenten die neue Adresse der Server-Objekts mitteilen könnte, bzw. ein Interface, mit dem man den lokalen ORB von der bevorstehenden Migration unterrichten könnte, verlieren Klienten die Referenz auf das Serverobjekt, sobald dieses den aktuellen Prozessraum verlässt. Sowohl RPC, als auch CORBA und JavaRMI gehen von der Annahme aus, dass das Netzwerk jederzeit verfügbar ist, und das der Ausfall von Verbindungen bzw. Verzögerungen beim Datentransport gegenüber dem Anwender und Programmierer verborgen werden kann. Für diesen dauert ein Aufruf entsprechend länger.

Einem Client den transparenten Zugriff auf externe Ressourcen zu schaffen, erlaubt zwar die Illusion eines gemeinsamen Systems, einfacher und in letzter Konsequenz effektiver wäre es jedoch die Komponenten die miteinander kommunizieren müssen, möglichst „nah" beieinander zu halten und dies bei einer eventuellen Änderung der Konfiguration entsprechend zu berücksichtigen [28], da durch den Wegfall des Netzes und die Verringerung der beteiligten Knoten die Wahrscheinlichkeit von Ausfällen sinkt. Insgesamt scheinen verteilte Systeme durch die enge Bindung an ein funktionierendes Netz als Basis für ein Agentensystem weniger geeignet.

## 2.3. Mobile Systeme

Im Unterschied zu verteilten Systemen, bei denen die Hardware an einem Ort fest installiert ist, sich aber die Software in Form von Programmen bzw. Agenten bewegt, ist es bei mobilen Systemen umgekehrt. Hier migrieren nicht Programme, sondern die sie beherbergenden Systeme wechseln den physischen Ort und terminieren zeitweise die Verbindung zum restlichen System, um zu einem anderen Zielpunkt wieder Kontakt zur Umgebung aufzunehmen [33]. Exemplarisch sei hier ein Applet in einem Mobiltelefon angeführt. Obwohl fix bezüglich der Hardware „Telefon", ist es doch nicht stationär, da sich der Träger bewegen kann. Man

sollte sich vergegenwärtigen, dass im Gegensatz zur Migration innerhalb von verteilten Systemen, die Migrationgeschwindigkeit des Agenten hier durch den Träger bestimmt wird – und somit relativ gering ist. Ein anderes Beispiel für mobile Systeme sind wearable computers bei denen der Träger selbst Teil eines mobilen Computersystems wird und Daten über ein ad hoc Netzwerk mit in der Nähe gefundenen, anderen Systemen austauscht. Die Daten erreichen schließlich ein Gateway, über das sie in das stationäre Netz gerouted werden. Dieser Anwendungsfall wird heute innerhalb des Militär als Verbindungsnetzwerk für Kampfverbände erprobt [34]. Es bietet einen interessanten Ansatzpunkt, da hiermit ortsbasierte Dienste realisiert werden können, sobald der im mobilen System laufende Agent zusätzliche Informationen über seinen aktuellen Standort, beispielsweise über GPS, erhielte. Die Verbindung zum nächsten Pizza Service, oder der kürzeste/sicherste Weg zum nächsten Nachschubdepot sind denkbare Anwendungsfälle. Speziell der Übergang auf die nächste Generation im Mobilfunknetz wird zusammen mit den immer leistungsfähiger werdenden Endgeräten verstärkt zu Lösungen führen, bei denen der Anwender seine Aufgaben nicht mehr zwangsweise an einem stationären System lösen muss, sondern Objekte, Dienste und Ergebnisse ständig mit sich führen bzw. erreichen und bearbeiten kann [35] [36]. Durch die Mobilität der Hardware werden in solchen Systemen die Umgebungen selbst zu Agenten, die in einer sich ständig ändernden Umwelt zusammenfinden und –arbeiten müssen [1].
Die heute verfügbaren Klassen von mobilen Systemen lassen sich unterteilen in:
*Notebooks/Webpads* – Dies ist der am weitesten erforschte Bereich der mobilen Anbindung. Hier ist es ein Nutzer, der (in der Regel) als **Client** gegenüber dem System auftritt und Ressourcen nutzen möchte. Diese umfassen beispielsweise Fileserver- und Druckdienste. Das Notebook selbst tritt gegenüber dem restlichen System normalerweise nicht als Dienstanbieter auf, auch wenn dies in Ausnahmefällen durchaus möglich ist (Webserver-Dienste, Kommunikationsdienste). Es besteht daher keine Notwendigkeit, dass das Endsystem und das stationäre System ständig den Aufenthaltsort abgleichen. Vielmehr reicht es aus, sich beim ersten Zugriff auf eine stationäre Ressource nach einer erfolgten Migration mit dem fixen Netzwerk zu synchronisieren. Dies kann beispielsweise die Erneuerung von DHCP Adressen bedeuten und die Anmeldung an den lokalen Sicherheitsservice.

*Smartcards* – Oder allgemeiner, Authentifizierung, weist einen (mobilen) User gegenüber dem System als legalen Nutzer aus. Hierbei tritt das mobile System (die Smartcard) gegenüber dem stationären System als Dienst auf, dessen alleinige Zweck die eigene Authentifizierung ist. Bei der Passworteingabe bei einem Notebook weist sich der Anwender durch „Wissen" aus, dass beliebig kopierbar ist, während die Smartcard einem Benutzer eine, durch nicht-reproduzierbare Speicherung einmalige und fälschungssichere, **Identität** gibt [37]. Für einen Überblick siehe auch [38]. Ein Abgleich des aktuellen Standorts ist nicht notwendig, vielmehr ist das der Nebeneffekt der Authentifizierung da der Standort des Authentifizierungsterminals, über den die Anmeldung erfolgte, bekannt ist.

*Mobile Devices* – Sind Geräte, die eine echte **Erweiterung** des stationären Systems darstellen. Beispiele sind Mobilfunk-Endgeräte, die als physischer Endpunkt im Funknetz auftreten. Hierbei ist es wesentlich, dass das stationäre System immer den physischen Aufenthaltsort jedes angemeldeten Endgeräts besitzt. Im Gegensatz zu den vorherigen Klassen muss das System hier ständig den Überblick über die aktuelle Konfiguration behalten. Es wird sich später noch zeigen, dass es sehr schwierig ist, dies performant zu gewährleisten.

*Autonome mobile Roboter* - Arbeiten auf sich selbst gestellt in einer ihnen unbekannten Umgebung. Das auf einem solchen System laufende Planungsmodul benötigt in der Regel keine Verbindung zu anderen Systemen um seine Aufgabe zu erfüllen. Meist sind die am Roboter angebrachten Sensoren ausreichend um ein für die Aufgabe genügendes Abbild der Umwelt zu erzeugen. Eine Verbindung zu einem stationären System als **Teilnehmer** ist hier optional, da vom Systemdesign her verzichtbar, jedoch manchmal nützlich um an detailiertere Informationen zu gelangen, als sie die eingebauten Sensoren liefern könnten. Auch kann es so dem restlichen System über die Verbindung in das Netzwerk die eigenen Absichten mitteilen. Der aktuelle Aufenthaltsort muss durch die Fähigkeit anderer Teilnehmer die Umgebung selbständig zu erkunden, nicht unbedingt bekannt gemacht werden. Dennoch ist es sinnvoll, da der Standort möglicherweise in die Planung anderer Teilnehmer eingeht.

Wenn wir die Klasse mobiler Systeme betrachten, die auf ein ständiges und konsistentes Bild der physischen Konfiguration benötigen, so stößt man auf Probleme, die durch das häufige Migrieren der beteiligten Komponenten entstehen. Der Aufenthaltsort muss dem restlichen System durch den Namensdienst oder einen spezialisierten Locations-Dienst bereitgestellt werden. Da dieser Dienst keine veralteten Aufenthaltsorte referenzierter Objekte ausgeben darf, muss ein Objekt dem Dienst vor dem eigenen Wechsel bzw. nach der Migration, wenn diese – wie im Fall mobiler Geräte – nicht vorhersagbar ist, die beabsichtigte Veränderung des Standortes mitteilen. Im Vergleich von verteilter, und mobiler Lösung zeigt sich, dass beide von einem funktionierenden Locations-Dienst abhängen. Dieser hat zwei Aufgaben. Zum einen dient er dem Finden von Objekten. Zum anderen wird er genutzt, wenn entschieden werden muss, ob ein Objekt noch lebt, was in der Regel gleichgestellt wird mit der Frage nach Erreichbarkeit. Ein Agentensystem als Teil eines mobilen Systems tritt gegenüber dem stationären System selbst als Agent, womit sich die Frage stellt, ob sich die Methoden des Suchens und Findens eines Agenten nicht auch auf das Suchen und Finden von Agentensystemen abbilden lassen [39].

Kurz erwähnt werden sollen noch die sich aus unterschiedlich starken **Bindungen** zwischen den Knoten eines Systems ergebenden Konsequenzen. Die Art der Bindung bestimmt maßgeblich die Möglichkeiten zur Kommunikation bzw. zur Migration zwischen den Knoten. Die engste Bindung bilden die Knoten eines **Clusters**. Hierbei wird die Hardware und die darauf laufende Software repliziert, um Ausfälle einzelner Komponenten vor den Nutzern des Systems zu verbergen. Im Gegensatz dazu ist die Bindung in einem **Verteilten System** bereits deutlich schwächer. Dies beginnt schon mit der eigentlichen Hardware, die in einem Clustersystem meist innerhalb eines Racks, oft sogar innerhalb eines einzigen Gehäuses untergebracht ist, während die Knoten eines verteilten Systems typischerweise innerhalb eines Raumes bzw. innerhalb eines Gebäudes oder administrativen Einheit stehen. Durch ein spezielles Betriebssystem wird eine für den Benutzer *-transparente Umgebung gebildet, wobei insbesondere Orts-, Migrations-, und Fehler- Transparenz angestrebt wird. Damit nimmt der Benutzer das System nicht mehr als ein aus Teilen zusammengesetztes System wahr, sondern bekommt den Eindruck, als sei das System eine homogene Einheit. Alle Aspekte der durch die physische Verteilung resultierenden Probleme hinsichtlich gemeinsam nutzbaren Speichers, Synchronisation, und Dateisystemem werden durch eine Softwareschicht – realisiert durch das Betriebssystem – verborgen. Da die am verteilten System teilnehmenden Knoten über das gleiche Betriebssystem verfügen (müssen), wird man abhängig vom Hersteller dieser Software. Die schwächste Kopplung zwischen den Knoten wird bei **Netzwerk Systemen** erreicht. Hier besteht die einzige „Bindung" aus einem gemeinsamen Netzwerkprotokoll derart, dass Prozesse auch über Knotengrenzen hinweg kommunizieren können. Es wird erst gar nicht versucht, den Eindruck eines virtuellen Einmaschinensystems zu schaffen. Der entscheidende Vorteil ist, dass man so nicht mehr auf jedem Knoten das gleiche Betriebssystem benötigt. Synchronisation, Kommunikation und Resourcensharing über Knotengrenzen hinweg wird dadurch weg vom Betriebssystem, auf die auf den einzelnen Knoten arbeitenden Benutzerprozesse verlagert. Obwohl auf den ersten Blick unnötig kompliziert, erweist sich dieser Ansatz als vorteilhaft, da unterschiedliche Konzepte parallel ge-

testet werden können und bei Verbesserungen an einzelnen Komponenten nicht alle Knoten gleichzeitig angepasst werden müssen [27] [40] [41].

## 2.4. Wie mobil ist ein Agent?

In diesem Abschnitt befassen wir uns mit den verschiedenen Aspekten der Mobilität eines Objekts. Es kann hierbei unterschieden werden, ob Daten, Code oder das Objekt selbst auf dem entfernten System Funktionen ausführt.

**Daten** – Mobile Daten sind die älteste Art der entfernten Ausführung. Hierbei werden die auszuführenden Funktionen bereits durch auf dem entfernten System laufende Prozesse bereitgestellt und benötigen nur noch die zu prozessierenden Daten. Ein Client löst die Adresse des entfernten Systems auf und übergibt der Funktion, sowie die aktuellen Daten und wartet auf das Ergebnis seines Aufrufs. Je nach Implementierung werden die Daten binär, oder in einem auch von Menschen lesbaren Format übertragen. Beispiel für ein binäres Übertragungsprotokoll ist der RPC. Die Funktionen werden durch den Server implementiert und über eine Unterprogrammnummer sowie eine Versionsnummer, veröffentlicht durch den Portmapper, externen Anwendungen bekannt gemacht. Der Client muss die zu bearbeitenden Daten in einem durch XDR festgelegten, plattformunabhängigen Format bereitstellen und erhält die Antwort des Servers im gleichen Format, das dann ggf. wieder in die lokale Repräsentation umgewandelt werden muss [19] [42]. Andere Übertragungsprotokolle, besonders im Umfeld der IETF, benutzen eine ASCII Darstellung um sowohl die gewünschte Funktion auszuwählen, als auch die Daten zu übertragen. Populäre Beispiele sind HTTP, SMTP, sowie NNTP. Bei diesen Diensten wird die auszuführende Funktion über einen Namen angesprochen und Parameter erst durch den Service selbst in ein internes, binäres Format übersetzt [43] [44] [45] [46].

Bei der Mobilität des Codes kann unterschieden werden, welcher Teil des Codes auf dem entfernten System ausgeführt wird. Ist es vom eigentlichen Klienten losgelöster Code, der nur auf dem entfernten System eine sinnvolle Bedeutung hat, oder sind es Teile des Programms, bzw. der gesamte Prozess, der auf dem entfernten System ausgeführt werden soll?

**Entfernter Code** – Die entfernte Ausführung von Code findet man bereits in vielen Anwendungen. Externe Geräte, deren Funktionalität nicht vollständig durch den Aufruf von Funktionen werden können, definieren in der Regel eine eigene Sprache, in der benutzerdefinierte Aktionen beschrieben werden können. Druckersprachen, wie zum Beispiel Postscript, fallen in diese Kategorie. Der Client migriert einen Teil seines Codes auf das Zielsystem und lässt ihn dort evaluieren. Obwohl entfernt, hat diese Form der Migration nichts mit der hier behandelten Form zu tun, sind doch Quellsystem und Zielsystem nicht homogen (Druckersprache ungleich Assemblersprache des Hosts) und die Kommunikation meist nur in einer Richtung möglich (vom Host auf das externe System) mit eventueller Signalisierung des Fehlerzustands. Die Rückkehr des Codes auf den Host ist in der Regel weder vorgesehen, noch erwünscht [22] [23].

**Mobiler Code** – Im Gegensatz dazu verallgemeinert die Migration von Code den Gedanken des entfernten Codes insofern, als dass auch die kontrollierte Rückkehr von Ergebnissen ermöglicht wird. Damit wird nicht mehr nur ein Teil des Codes, auf das entfernte System übertragen, sondern der komplette Agent, d.h., Daten inklusive Code. Je nach verfolgter Strategie wird darüber hinaus noch der Zustand aller zum Zeitpunkt der Übertragung laufenden Threads eingefroren und auf dem entfernten System wiederhergestellt. Nach erfolgter Übertragung werden auf dem Ursprungssystem die gespeicherten Daten und Code gelöscht. Auf dem Zielsystem hingegen wird der Code fortgeführt oder in einem definierten Zustand gestartet.

Im folgenden beschränken wir uns auf den Mobilen Code, der den kontrollierten Wechsel zwischen den Knoten zulässt. Die Art, wie Mobiler Code den Besuch der verschiedenen Knoten vorbereitet, lässt sich zu dessen genaueren Klassifikation nutzen. Eine mögliche Migrationstrategie besteht aus dem **Erzeugen eines „Reiseplans"** (Itinerary, [47] [48]), der eine sortierte Liste der zu besuchenden Knoten enthält. Für das Laufzeitsystem ist dieser Plan Grundlage der Entscheidungen welcher Teil des Agenten auf welchen Knoten repliziert werden muss. Diese Vorausplanung seitens des Agenten ermöglicht dem System eine bessere Ressourcenverwaltung. So kann zum Beispiel der Code als Teil einer anderen Aktion bereits vor dem eigentlichem Besuch des Agenten auf die noch nicht besuchten Knoten kopiert werden. Eine weitere Migrationstrategie ist es, dass **der Agent selbst die Migration anstößt**, woraufhin Code und Daten auf das Zielsystem übertragen werden. Unterschiede hinsichtlich der Transparenz des Migrationsaufrufs ergeben sich durch die mögliche Übertragung des Prozesszustands zusätzlich zu den Daten. Zum einen unterscheidet man die „Schwache Migration". Hierbei ist es die Aufgabe des Agenten vor dem Migrationsschritt alle Ressourcen geeignet zu verpacken, die am Zielort wieder benötigt werden. Dies ist deswegen notwendig, da der Prozesszustand selbst nicht als Teil der Migration auf das Zielsystem übertragen wird, der Agent also wieder mit dem Aufruf einer festen oder durch die eingepackten Parameter ausgewählten Funktion aktiviert wird. Dieser Ansatz wird häufig im Zusammenhang mit Java Implementierungen benutzt, da die Standard Java VM die Serialisierung von Threads nicht unterstützt, dies aber notwendig für die portable Übertragung des Prozesszustands ist [49] [50] [9] [47]. Weil besonders mächtig, wird in der Regel „Starke Migration" angestrebt. Hierbei werden die Daten, der Code, sowie der aktuelle Ausführungszustand aller Threads des Agenten auf das Ziel übertragen, womit sich das Verpacken des internen Zustand durch den Agenten selbst erübrigt, da auf dem Zielsystem nichts rekonstruiert werden muss. In der Regel wird dies heute bei Agentensystemen in interpretierten Sprachen erreicht; weniger bei bereits kompilierten Sprachen, da man sich hier schnell auf der Ebene des Betriebssystems wiederfindet und man mit VMs und unterbrochenen Betriebsystemaufrufen hantieren muss [51] [28] [27]. In jüngerer Zeit ist die Unzulänglichkeit der Java VM, den Thread-Stack portabel zu verpacken und zu restaurieren, bereits mehrfach Ziel von Verbesserungen gewesen. Zum einen wurde vorgeschlagen, die VM so zu erweitern, dass man von innerhalb der VM die Thread Stack Informationen zugreifen kann. Da dies eine Änderung der VM bedingt, ist diese Lösung nicht befriedigend. Andere Gruppen beschäftigen sich mit dem dynamischen re-write von bereits compilierten Klassen derart, dass sie Code beim Laden der Klasse einfügen, der im Fall einer Migration für das Einpacken der lokalen Zustände sorgt. Zusätzlich erlaubt es der eingefügte Bytecode – für den ursprünglichen Code transparent – den gesicherten Zustand wieder herzustellen, und das inklusive der Aufrufhierarchie. Aktuelle Implementierungen dieser Technik installieren einen eigenen Classloader der beim Laden die Klassen entsprechend instrumentiert. Als negativen Effekt erhöht diese Technik den Codeumfang auf circa das Doppelte. Zusätzlich kostet der eingebaute Code Performance [52] [53]. Eine letzte Strategie **überlässt die Migration der Umgebung**. Indem sich der Agent an eine bestimmte Ressource bindet (er beschafft sich ihre Referenz auf geeignete Weise) signalisiert er dem System, dass er nach Möglichkeit zusammen mit der Ressource auf einen geeigneten Knoten transferiert werden möchte. Ob die Möglichkeit dazu besteht, hängt von verschiedenen Faktoren ab, unter anderem, ob der Prozess, bzw. seine bereits referenzierten Ressourcen auch auf einen gemeinsamen Knoten transferierbar sind. Diese Entscheidung wird jedoch dem System überlassen, dass adaptiv bzw. konfiguriert entfernte Kommunikation ermöglichen kann, oder die Partner auf einen gemeinsamen Knoten mit anschließend lokaler Kommunikation zusammenführen könnte. Man verliert bei diesem Design auch nicht die Möglichkeit der expliziten Migration, reicht es doch, sich an einen auf dem Zielknoten verankertem Objekte ohne Remotekommunikationsmöglichkeit zu binden [28] [27].

## 2.5. Warum mobile Agenten?

In diesem Abschnitt befassen wir uns mit den Vorteilen mobiler Agenten aus der Sicht von Betriebssystembauern bzw. Systementwicklern. Dies Sichtweise der KI sei hier ausdrücklich

ausgeklammert. Auch sollen hier die allgemeinen Vorteile von Agenten gegenüber anderen Ansätzen der verteilten Programmierung unberücksichtigt bleiben.

**Mobile, autonome Agenten benötigen keine homogene Betriebssystemumgebung**, vielmehr genügt eine lose Kopplung der beteiligten Systeme. Speziell können sie auch in Abwesenheit von Teilen des Systems bzw. dem Fehlen einer Verbindung zum Restsystem, immer noch einen Teil ihrer Aufgaben erfüllen und bei Wieder-Erreichbarkeit des Restsystems mit ihrer Aufgabe fortfahren. Hierbei muss aber zwischen mobilen Agenten und autonomen Agenten unterschieden werden. Ein mobiler Agent erlaubt es dem Objekt sich frei in der Menge der erreichbaren Knoten zu bewegen. Es wird keine Aussage darüber gemacht, ob der Agent auch eine sinnvolle Aufgabe erledigen kann, ohne mit seinem Besitzer in Kontakt zu treten. Erst ein autonomer Agent, der nicht zwangsweise mobil sein muss, ist per Definition zu selbständigen Handeln in der Lage. Dies wird durch eine Planungskomponente erreicht, die aus dem Bild der Umgebung und einem vorgegebenen Ziel die nächsten Schritte errechnet und über Aktautoren die Umwelt versucht zu verändern. Damit sind die Konzepte „Mobilität" und „Autonomie" orthogonal zueinander und können somit auch unabhängig voneinander implementiert werden [9].

**Potentiell verringern** mobile Agenten im Vergleich zu ihren stationären Kollegen die **Netzwerkbelastung**, da sie für Aufgaben bei denen eine große Anzahl Daten zu bewerten sind, durch die vorherige Migration auf den betroffenen Knoten die Menge der über das Netz übertragenen Daten deutlich verringern können. Diese Intention gibt es bei vielen Ansätzen [54], jedoch ist im heutigen Umfeld die Bandbreite des Netzwerks auf einem vergleichbaren Niveau wie die Anbindung eines Prozessors an den lokalen Hauptspeicher, womit es nicht mehr erstrebenswert scheint, die genutzte Bandbreite zu beschränken. Ein Problem besteht allerdings selbst dann, wenn die Netzwerke eine hohe Bandbreite aufweisen: Dies ist die möglicherweise nicht unerhebliche Latenzzeit, die die Effizienz vieler kleiner Abfragen verringert, die aber einfach und effektiv durch die Migration auf den Zielknoten umgangen werden kann.

Autonome Agenten **arbeiten auch in einer Umgebung mit hoher Fehlerrate**, da sie nicht auf einen ständigen Kontakt zu ihrem Besitzer angewiesen sind. Durch die Migration ist eine zusätzliche Möglichkeit gegeben, um auf Fehler zu reagieren [34].

**Durch „Intelligenz" weniger Kommunikation** mit Besitzer notwendig. Mobile Agenten arbeiten auf entfernten Knoten. Bei klassischen Systemen, die meist stationär betrieben werden und eine permanente Netzverbindung haben, wird dieser Vorteil nicht gleich offensichtlich. Doch im Bereich des „Mobile Computing", mit Geräten die nicht mehr auf eine dauerhafte Netzverbindung angewiesen sind, lösen sich vom Besitzer trennende Agenten elegant das Problem, wie man eine Aufgabe startet, ohne auf das Ergebnis warten zu müssen. Vorstellbar ist ein Agent, der ein bestimmtes Buch in den Online-Läden sucht und seinem Besitzer dann das Ergebnis der Recherche per email mitteilt. In diesem Zusammenhang kommt der Sicherheit in einem verteiltem System natürlich besondere Bedeutung zu, sei es der Schutz der Agenten vor anderen Agenten/dem System, wie auch der Schutz des Systems und seiner Ressourcen vor potentiell gefährlichen Agenten. Auf diesem Gebiet wird intensiv geforscht [55] [11] [56].

Mobile Agenten unterstützen **verteiltes Rechnen und verteiltes Wissen** auf „natürliche Weise". Sind sie zusätzlich Autonom, können sie sonst kaum zugängliche Probleme lösen helfen. Forschungen haben beispielsweise erfolgreich Modelle aus der Tierwelt auf das Problem des Routings in Netzwerken übertragen. Dabei bedienen sich Agenten der mittels Botenstoffen aufgebrachten Markierung vorheriger Instanzen. Wenn mehrere Instanzen den gleichen Weg wählen, verstärkt sich der „Duft" und damit die Präferenz folgender Agenten den gleichen Weg zu wählen. Durch das simulierte „Verduften" nach endlicher Zeit, wird eine Adaption auf sich verändernde äußere Zustände erreicht, da einmal gefundene Routen, um aktiv zu bleiben, ständig erneuert – und damit getestet – werden müssen. [57] [58] [59].

# 3. Anforderungen mobiler Agenten an die Infrastruktur

Im vorangegangenen Kapitel wurden Autonome und Mobile Agenten und die sie beherbergenden Systeme vorgestellt. Dieses Kapitel beschäftigt sich mit den Problemen, die sich aus der Nutzung konkreter Umgebungen ergeben. Speziell die Verwaltung von (lokalen und entfernten) Ressourcen, das Suchen und Finden von Ressourcen im System, sowie die Konfiguration einzelner Komponenten, werden hier näher untersucht. Dabei liegt der Schwerpunkt auf der Bereitstellung der notwendigen Funktionalität und der Bewertung bestehender Lösungen unter Berücksichtigung der Anforderungen an die Infrastruktur.

## 3.1. Das Ressourcen Dilemma

Viele der im Zusammenhang mit Ressourcen auftretenden Probleme sind im Umfeld nicht verteilter Programme unbekannt. Dies betrifft beispielsweise den eigentlich simplen Lebenszyklus eines Prozesses. Im nicht verteilten Fall kann ein Client, unterstützt durch das lokale Betriebssystem, zu jedem Zeitpunkt entscheiden, ob der gesuchte Prozess noch aktiv ist. Auch der Zugriff auf Funktionen ist deterministisch, und Fehler, die nicht auf fehlerhafte Programmierung zurückzuführen sind, korrelieren direkt zu Hardwareproblemen. Speziell sind Ergebnisse nach endlicher Zeit verfügbar, und wenn nicht, kann man problemlos testen, ob der beauftragte Prozess noch aktiv ist. Der Zugriff auf entfernt vorliegende Daten erfolgt durch Kommunikation des lokalen Prozesses mit auf dem entfernten System laufenden Prozessen, welche die gewünschten Daten zurückliefern. Jedoch ist die Bereitstellung jeder möglichen Funktion auf einem entfernten System praktisch nicht möglich, wodurch in der Regel eine große Datenmenge über das Netzwerk transportiert werden muss, selbst wenn nur ein verdichteter Wert gewünscht wird. Ein Agentensystem, das nun den Transfer und die Ausführung von beliebigen Code auf dem entfernten System ermöglicht, kann den Auswertungsprozess auf den entfernten Rechner transferieren, dort die Daten lokal verdichten und zusammen mit dem Ergebnis wieder zurück auf den Ursprungsrechner holen. Da die Migration für die beteiligten Prozesse nicht transparent ist und durch den Programmierer explizit angestoßen wird, hat das lokale System ein Problem wenn es den Zustand eines Prozesses zurückgeben soll, da es für Prozesse, die den lokalen Rechner verlassen haben, nicht mehr entscheiden kann, ob der Prozess noch aktiv ist. Wichtiger noch, sind mit dem Verlassen alle lokalen Referenzen auf den bisherigen Prozess nicht mehr gültig. Eines der Probleme der mobilen Agenten ist das nicht vorhersagbare Kosten/Nutzen Verhältnis zwischen lokal einzusparendem Datentransfer gegenüber der Datenmenge, die sich aus dem Code des Agenten und den bereits gesammelten Daten zusammensetzt. Tests haben gezeigt, dass häufig der Vorteil der lokalen Abfrage von Attributen und dem sich daraus ergebenen, reduzierten Netzwerkverkehr, erkauft wird mit dem Transfer des Agenten über viele Zielsysteme. Da ein nicht-trivialer Agent aus einer nicht unerheblichen Menge Code besteht, resultiert dies in einer großen Menge Code, die während der „Reise" mitgeführt werden muss. Besonders sei hier auf den Einsatz von Agenten im Zusammenhang mit Netzwerk Montitoring hingewiesen [54] [50] wo die normalerweise per SNMP Protokoll zentral durch eine Applikation erfragten Eigenschaften von Netzwerkknoten durch Migration von Teilen der Applikation auf die beteiligten Rechner und anschließender Rückkehr auf das zentrale System ersetzt werden. Man verspricht sich eine höhere Performance des Systems zusammen mit einer erhöhten Ausfallsicherheit, doch zeigt sich, dass zumindest eine höhere Performance nicht automatisch durch den Einsatz von mobilem Code erreicht werden kann. Zwei Effekte verhindern dies. Zum einen ist in der Regel der Code des Agenten groß gegenüber der abgefragten Datenmenge auf den Netzwerkknoten. Zwar können die Latenzzeiten der vielen Abfragen ersetzt werden durch den Transfer des Programms auf den Zielrechner und die darauf folgende lokale Abfrage der Parameter, doch verliert sich die Zeitersparnis in der bei der Migration für die Übertragung des Codes verursachten Wartezeit. Versuche, die Übertragung zu optimieren und auf Seiten der beteiligten Systeme Caches zu verwenden, tragen erheblich zu einer Verringerung dieses Zeitverlusts bei, doch bekämpft der Einsatz von Caches nur das Sym-

ptom, nicht das Problem selbst. Praktisch entspricht ein Cache einer (automatischen) Installation der Komponente auf dem entfernten System. Zum anderen wird die Menge der Daten die während der Abarbeitung des Auftrags akkumuliert wird, stetig größer, muss aber während der gesamten Laufzeit des Agenten transportiert werden, auch wenn sie nicht lokal benötigt werden. Dieses Problem zeigt sich gerade bei der Abfrage von Statusinformationen in einem Netzwerk. In der Regel sind hier die Daten der beteiligten Rechner unabhängig voneinander und nur für den Besitzer des Agenten interessant. Dennoch müssen sie bis zur Rückkehr des Agenten auf den Knoten des Besitzers mitgeführt werden und erhöhen so bei jedem weiteren Migrationschritt den Datenumfang. Entschärfen lässt sich dies nur dann relativ einfach, wenn entweder der Reiseplan bekannt ist, oder man davon ausgehen kann, dass sich der Agent mehrfach auf ein System transferieren möchte. In diesen Fällen können die Ergebnisse entweder bereits asynchron vom Agenten auf die Zielknoten übertragen werden, oder die Daten auch nach Verlassen des Agenten weiterhin lokal gespeichert, und erst beim letzten Transfer des Agenten mit auf das Zielsystem genommen werden.

Löst man die Ressourcen vom Agenten selbst und fasst man einen Agenten auch wieder als eine Ressource auf, kann dieser eine beliebige Ressource auf einem besuchten Agentensystem zurücklassen und die dort gespeicherten Daten zur gegebenen Zeit wieder einsammeln bzw. zerstören [60]. Dies ist nicht die einzige Möglichkeit, Ressourcen gewinnbringend zu nutzen. Man muss allerdings die verschiedenen Freiheitsgrade unterscheiden, mit denen eine Ressource ausgestattet werden kann. Zum einen unterscheiden sich Ressourcen hinsichtlich ihrer Möglichkeit, den lokalen Knoten zu verlassen:

Eine Freie Ressource hängt von nichts anderem ab. Diese Ressource kann bei Bedarf mit dem Agenten migrieren, bzw. nachträglich auf den momentanen Aufenthaltsort des Agenten verschoben werden. Diese Ressourcenart ist die richtige Wah. für das beschriebene Beispiel der temporären Zwischenspeicherung von Daten. Bei einem späteren Besuch kann der Agent die bereits gespeicherten Werte wieder aufnehmen und sie mit den aktuellen Werten vergleichen, wieder abspeichern, bzw. als Anlass zur Benachrichtigung einer anderen Instanz nehmen. Schließlich wird der Agent die Ressource wieder freigeben.

Die Lokal-gebundene Ressource unterscheidet sich von einer „freien" Ressource dadurch, dass sie nicht den lokalen Knoten verlassen kann. Dies ist in der Regel dann der Fall, wenn sie auf lokalen Libraries zugreift für die eine entfernte Nutzung nicht vorgesehen ist. Die Ansteuerung eines zeitkritischen Geräts über eine lokale Schnittstelle verhindert zum Beispiel die Migration der steuernden Ressource.

Abhängige Ressourcen, entstehen bei der Aggregation von freien und gebundenen Ressourcen. Eine Abhängige Ressource selbst ist entweder frei oder lokal gebunden. Sie wird aber auf jeden Fall lokal gebunden, wird ihr eine Referenz auf eine gebundene Ressource zugewiesen wird. Nur stellt sich die Frage ob es sinnvoll ist, einer normalerweise freien Ressource durch diese Art der Zuweisung die Migration zu verbieten, verursacht es doch unerwartete Zustände seitens eines Agenten, der durch diesen Umstand (möglicherweise durch eine gar nicht selbst ausgeführte Aktion) unerwartet an den aktuellen Knoten gebunden werden kann.

Solange der Referenzierende auf dem lokalen Knoten bleibt, verursacht keine Ressourcenart ein Problem. Erst beim Verlassen des lokalen Systems muss entschieden werden, ob die Migration des Agenten möglich ist, oder abgebrochen werden muss. Dazu gibt es mehrere Ansätze, wie man durch Abhängige Ressourcen verursachte Zustände erkennen kann. Zum einen ist es natürlich immer möglich, dieses Problem zu ignorieren und es dem Agenten zu überlassen, im Konfliktfall die betroffenen, gebundenen Ressourcen zu erkennen und sie entweder freizugeben, oder die Migration auf einen späteren Zeitpunkt zu verschieben. Eine zweite Möglichkeit ist es, diejenigen Abhängigen Ressourcen besonders zu markieren, die zusammen mit dem Agenten migrieren sollen. Durch die (rekursive) Markierung soll verhindert werden, dass sie gebundene Ressourcen aufnehmen können. Sobald nicht mehr nur primitive Datentypen bzw. Strukturen ohne Referenz auf andere Typen in eine Abhängige

Ressource eingefügt werden sollen, muss man entweder konservativ sein und eigentlich Freie Ressourcen als nicht migrierbar klassifizieren (da sie Referenzen auf andere, möglicherweise gebundene Ressourcen enthalten könnten) oder aber die Fehlererkennung bis zum Zeitpunkt der Migration aufschieben (mit den Preis, dass erst dort erkannt wird, dass eine oder mehrere Ressourcen nur lokal verfügbar sind). Dieser letzte Ansatz wird zum Beispiel innerhalb der Java VM benutzt, wo man der Umgebung durch die Implementierung eines Interfaces (Serializable) anzeigt, dass die enthaltenen Daten keine lokalen Ressourcen referenzieren und transferiert werden können. Zum Zeitpunkt der Migration wird dann (im Rahmen des Marshalling) getestet, ob das durch das Interface gegebene Versprechen wirklich gehalten wird. Durch diese „Lazy" Überprüfung werden kurzzeitige Verletzungen des Status toleriert und führen nicht gleich zum Abbruch. Schließlich ist es möglich, den Ressourcen zu erlauben, ihren aktuellen Status dynamisch zu setzen. Es ist dann Aufgabe eines Ressourcen-Containers, den aktuellen Zustand zu überwachen und gemäß einer Container-Policy Änderungswünsche der im Container vorhandenen Ressourcen zu erlauben, bzw. den Statusübergang zu unterbinden. Damit bleiben vier Überprüfungsstrategien, nämlich, gar keines, strikt durch Typsystem, dynamisch zum Zeitpunkt des Transfers, oder dynamisch bei Statusänderung einer Abhängigen Ressource. Dieser letzte Ansatz erscheint am erfolgversprechendesten, kann er doch alle Situationen berücksichtigen, da durch den Agenten selbst eine geeignete Strategie implementierbar ist. In mobilen Systemen sind Ressourcen analog zu verteilten Systemen zu verwalten, da einer Feste Ressource in einem mobilen System nicht anders zu handhaben ist, als in einem verteilten System. Selbst wenn der Host bewegt wird, so wird auch die lokale Hardware – und damit die lokalen Libraries und Ressourcen – mitbewegt.

Ressourcen können nicht nur anhand ihrer Affinität zum lokalen System beschrieben werden. Auch die mögliche Art des Zugriffs eignet sich zur Unterscheidung. Je nachdem, ob er lokal, oder entfernt ausgeführt wird, ergibt sich:

Zum einen der direkte, lokale Zugriff ohne weitere Mithilfe des Systems. Interne Ressourcen oder Ressourcen, die vom Agenten selbst bereitgestellt werden, sind normalerweise auf diese Art erreibar. Entfernte Agenten haben keinen Zugriff auf die Ressource, solange sie nicht vorher auf das lokale System migrieren. In der Regel ist diese Ressourcen-Klasse ebenfalls lokal gebunden, doch muss das nicht immer der Fall sein (Proprieräres Interface auf die internen Daten ohne Abhängigkeiten auf lokale Dienste).

Zum anderen kann eine Ressource Zugriffe über einen Proxy erlauben. Dies ist dann der Fall, wenn die Ressource selbst entweder nicht auf dem lokalen Knoten verfügbar ist, oder lokale Richtlinien (Sicherheitsrichtlinie) den direkten Zugriff verbieten. Der Proxy stellt gewissermaßen einen Vermittler zwischen dem Klienten und der angesprochenen Ressource dar und kann damit Aufgaben erfüllen, die über das einfache, transparente Weiterleiten von Aufrufen hinausgehen. Dazu muss die Ressource entweder einen standardisierten Zugang zu den angebotenen Funktionen bereitstellen (damit man den Proxy ggf. automatisch generieren kann) oder den Proxycode publizieren. In beiden Fällen wird der Proxy Teil des Klienten und kann somit auch interne Zustände speichern, was bei der gemeinsamen Nutzung einer Proxy-Instanz über mehrere Klienten nicht möglich wäre. Bei der Migration wird der Proxy allerdings unter Umständen invalidiert, und zwar dann, wenn er lokal gebundene Ressourcen für die Implementierung seines Dienstes benötigt.

Schließlich kann eine Ressource noch über einen Forwarder angesprochen werden. Der Unterschied zum Proxy liegt im Kontext, in dem der Forwarder instanziiert wird. Während ein Proxy Teil des Klienten ist, in seinem Kontext arbeitet und mit dem Agenten migriert, wird ein Forwarder von der Ressource zurückgelassen, wenn sie migriert, um als Teil der Umgebung nachfolgenden Klienten eine Lokalisierung der Ressource zu ermöglichen. Ein Proxy wird unter Umständen invalidiert wenn der Klient migriert, da die in ihm gespeicherten Informationen möglicherweise nur lokale Bedeutung haben, was auf grund der statischen Natur eines Forwarders nicht passieren kann. Damit ein Forwarder den Kontakt zum referenzierten Objekt halten kann, implementiert er Teile der Funktionalität eines LocationServers, auch wenn dies mit anderer Zielsetzung geschieht: Während der LocationService den Erstkontakt zwischen zwei Parteien gestattet, wird durch den Forwarder eine erneute Kontaktaufnahme zu einem bereits bekannten Partner ermöglicht.

Zusammenfassend kann man sagen, dass es Ziel für ein Ressourcenmanagement sein muss, die Abstraktion von Freien und Gebundenen Ressourcen derart zu gestalten, dass deren Nutzung durch Klienten einfach und ohne künstliche Barrieren möglich ist. Für Ressourcen muss es einfach möglich sein zu migrieren, ohne Rücksicht auf eventuelle noch in Klienten vorhandene Referenzen nehmen zu müssen. Die erste Forderung ist durch von der Ressource zu implementierende dynamische Überprüfung realisierbar, während die zweite Forderung durch dynamische Implementierungen von Proxyklassen erreicht werden kann, die zur Laufzeit zwischen Original/Proxy/Forwarder umschalten können.

## 3.2. Mobilität hat ihren Preis: Wie finden sich Ressourcen?

Im vorangegangenen Kapitel wurden die unterschiedlichen Klassifizierungen von Ressourcen vorgestellt und der lokale, sowie der entfernte Zugriff beschrieben. Das nun folgende Kapitel beschäftigt sich mit den Fragen, die sich aus der Lokalisierung von nur über einen symbolischen Namen bekannte Ressourcen ergeben. Hierzu werden die unterschiedlichen Arten von Nameservern, bzw. der hier notwendige Abwandlung in Form eines Locationservice, vorgestellt und ihre Eignung für die Verwaltung von mobilen Objekten untersucht.

Ein LocationService ist zuständig für das Mapping eines Universal Resource Name (URN) auf eine Universal Resource Location (URL). Hierzu stellt er zwei Interfaces zur Verfügung. Über das Update Interface macht sich eine Ressource dem LocationService bekannt und trägt ihren aktuellen Aufenthaltsort ein, während über das Search Interface ein Klient nach einem ihm bekannten symbolischen Namen sucht und dessen zuletzt bekannten Aufenthaltsort als Ergebnis erhält. Um dies effektiv zu gewährleisten, muss ein LocationService zwei Bedingungen erfüllen. 1. Suchen muss schnell sein und den „richtigen" Wert liefern. 2. Änderungen müssen schnell sein, auch wenn das Objekt nicht auf dem gleichen System ist, wie der Service. Wie wir im weiteren Verlauf noch sehen werden, ist es schwierig, diese sich wiedersprechenden Ziele gleichzeitig zu gewährleisten. Das Suchen nach einem Objekt kann durch den LocationService auf zwei unterschiedliche Arten realisiert werden. Mit der Hilfe der lokalen Knoten, oder ohne die Hilfe der lokalen Knoten. Wenn die Hilfe der lokalen Knoten gewünscht wird, muss durch geeignete Maßnahmen sichergestellt werden, dass diese auch vertrauenswürdig sind, das heißt, sie dürfen keine mutwillig veränderten oder ungültigen Referenzen herausgeben. Es lassen sich vier Strategien beschreiben, mit deren Hilfe ein LocationService eine Ressource lokalisieren kann. Die ersten beiden Lösungen benötigen die Hilfe der lokalen Systeme, die anderen kommen ohne eine derartige Unterstützung aus.

**Forward Chain** – Hierbei speichert jeder Knoten, der von einer Ressource (ein Agent, bzw. Programm kann ebenfalls als Ressource aufgefasst werden) besucht wird, einen Verweis auf den neuen nächsten Aufenthaltsort der Ressource. Ein Klient muss dieser Liste folgen, um

an deren Ende den Knoten zu erreichen, auf dem die Ressource schließlich lokal zu finden ist. Dies ist aus der Sicht der Ressource die schnellste Update-Variante, da die gesamte Funktionalität lokal in den beteiligten Systemen implementiert wird und nur die initiale Registrierung dem LocationService bekannt sein muss. Problematisch ist, dass hierbei die ständige Verfügbarkeit aller Knoten zwingend vorausgesetzt wird und dass alle Knoten die Informationen in dauerhaftem Speicher puffern müssen, um im Fall eines Absturzes beim anschließenden Neustart die Daten restaurieren zu können. Ist auch nur ein Knoten nicht (mehr) verfügbar, so ist die Information aller folgenden Knoten durch den Bruch der Kette wertlos. Weil simpel, wird dieser Ansatz dennoch häufig angewendet, wenn auch in der Regel erweitert um Mechanismen, die einerseits die Kette möglichst kurz halten, andererseits Informationen ausnutzen um „verstorbene" Ressourcen zu detektieren [61] [39] [11]. Um die Länge der Kette zu beschränken, genügt es, beim Wechsel der Ressource auf den nächsten Rechner den vorherigen Knoten über diesen Wechsel zu unterrichten und dann die eigene Referenz zu löschen – außer, man ist der erste Knoten in der Kette. In diesem Fall muss man die Referenz auf jeden Fall behalten. Als Resultat wird der erste Knoten nach kurzer Zeit immer auf den aktuellsten Knoten zeigen. Im statischen Fall (keine Ressource bewegt sich mehr) wird das System damit auf eine einstufige Kette zurückfallen. Um die Ausfallsicherheit zu erhöhen, kann ein Knoten nicht nur Referenzen auf seinen direkten Vorgänger halten, sondern zusätzlich bis zu n-Vorgänger über den eigenen Zustand benachrichtigen. Dadurch, dass er selbst weitere Referenzen hält, kann er auch für den Fall, dass seine n-1 direkten Vorgänger ausgefallen sind, immer noch die Kette für einen Klienten in der Form updaten, dass der Klient die ausgefallenen Knoten überspringt. Diese Verbesserung der Ausfallsicherheit geht allerdings auf Kosten der Performance, müssen doch nun ‚n' Rechner benachrichtigt werden und nicht mehr nur der direkte Vorgänger. Damit eine Vorwärtsreferenz nicht beliebig lange im Knoten gespeichert werden muss, muss ein Mechanismus vorgesehen werden, der ein automatisches Aufräumen ermöglicht. Obwohl für einen produktiven Einsatz unerlässlich, wird diese Forderung lange nicht von allen im Einsatz befindlichen Lösungen berücksichtigt [11] [62]. Aus Sicht der Sicherheit ist eine Forward Chain dennoch nicht optimal, da ein Klient jedem befragten Knoten glauben muss, dass die zurückgelieferten Informationen korrekt sind. Ein Knoten könnte auch manipulierte Einträge zurückliefern. Da die Kette beliebig lang werden kann, erhöht das auch entsprechend das Risiko, an einen unkooperativen bzw. angreifenden Knoten zu geraten.

**Search** – Hierbei zieht der Klient Vorteile aus dem Wissen um die Reiseroute der zu suchenden Ressource, bzw. aus der Tatsache, dass er die möglichen Zielsysteme bereits im voraus kennt (Topologiewissen). Es wird daher kein ausgezeichneter LocationService benötigt. Unterscheiden kann man zwei Ansätze: Wenn die Menge der möglichen Aufenthaltsorte bekannt ist, beschränkt sich die eigentliche Suche auf das sequentielle/parallele Abfragen der möglichen Ziele und die Auswertung, ob die gesuchte Ressource im Moment der Abfrage lokal auf einem dieser Knoten vorhanden ist. Wenn das Topologiewissen fehlt und auch der Reiseplan der Ressource nicht bekannt, bzw. zu umfangreich ist, wird die Suche komplexer. Der Klient kann über einen Multicast, bzw. Broadcast mit einer „Entfernungsbeschränkung" immer weitere Kreise um seinen physischen Knoten ziehen, in Erwartung, schließlich den Knoten mit der gesuchten Ressource zu erreichen. Je weiter sich die Ressource physisch vom suchenden System wegbewegt hat, desto kostspieliger wird die Suche; und das nicht nur für den Suchenden, da die Anzahl erreichter Knoten und damit die Anzahl Knoten, die die Nachricht empfangen und beantworten müssen, mit steigendem Suchradius entsprechend steigt. Wenn sich Broadcastsuche verbietet, bleibt dem Suchenden nur der Schuss ins Blaue. Er kann zum Beispiel die (unvollständige) Reiseroute, den letzten bekannten Aufenthaltsort oder das Wissen um andere Ressourcen, die sich „normalerweise" in der „Nähe" der gesuchten Ressource befinden, verwenden, um den wahrscheinlichen Standort zu erraten. Wenn auch das keine Einschränkung des Suchraums bringt, hat der Suchende immer noch die Möglichkeit, andere, eigentlich nicht direkt mit der Ressource verbundene Informationen ausnutzen. Darunter würde beispielsweise die Anfrage an die TopX der bisher befragten Knoten fallen, die Beschränkung der Broadcastsuche auf die Domaine der zu suchenden Ressource (so sie denn bekannt ist, möglicherweise als Teil des Namens), oder die

Domaine der Applikation. Schließlich kann der Suchende noch versuchen, Seiteneffekte der Aktionen der Ressource auszunutzen. Wenn diese beispielsweise Emails während ihrer Reise verschickt und der Suchende Zugriff auf deren Headerdaten hat, so kann er aus dem Mailpfad zumindest auf das grobe Zielgebiet schließen. Hinsichtlich bereits angesprochener Sicherheitsbedenken verhält sich das Suchen vergleichbar mit einer Forward Chain. Auch hier ist der Suchende auf die Kooperation der befragten Systeme angewiesen und muss darauf vertrauen, dass die zurückgelieferten Daten nicht manipuliert wurden.

Andere Lösungen benötigen keine Hilfe der lokalen Systeme bei der Suche nach einer Ressource. Dies wird erreicht durch die Speicherung aller notwendigen Daten durch einen dedizierten Dienstanbieter. Ein lokales System ist somit nicht mehr Teil des LocationServices, sondern nur noch dessen Klient. Die folgenden Ansätze benutzen einen oder mehrere ausgezeichnete LocationServer, die den aktuellen Aufenthaltsort in Form von URN/URL Tupeln zu speichern. Die Abfragen sind öffentlich, d.h., es werden keine weiteren Einschränkungen bzgl. Sicherheit gemacht, allerdings ist die Erzeugung und das Ändern von Einträgen nur durch berechtigte Objekte möglich. Eine sichere Variante ist die Implementierung eines LocationServices direkt auf dem System des Besitzers der Ressource. Updates des aktuellen Standorts können über ein gesichertes und vom Abfrage-Interface unabhängiges Protokoll durchgeführt werden, was einerseits Optimierungen des Protokolls ermöglicht, andererseits mögliche Angriffspunkte minimiert, da ein Angreifer durch den Bruch eines Systems nicht automatisch Zugriff auf alle anderen Systeme bekommt. Ein entscheidender Nachteil ist der Zwang der ständigen Erreichbarkeit, was gerade im Hinblick auf mobile Systeme vermieden werden muss. Eine Menge von ständig erreichbaren LocationServern ermöglicht die Delegation der Ortsspeicherung an einen dedizierten Server. Dazu wählt eine Ressource bei ihrer Erzeugung einen der verfügbaren LocationServer aus, trägt dort ihr Mapping von URN auf URL ein und veröffentlicht ihre Wahl.

Die Frage ist: Wer wählt den LocationServer aus? Dies ist problematisch, da:

1. Eine „andere" Ressource einen bereits bekannten Namen durch die eigene Implementierung belegt haben könnte.
2. Wer hindert ein System daran, eine Ressource derart zu manipulieren, dass Updatenachrichten an einen anderen LocationService geschickt werden (der damit die Ressource überwachen und Nutzer identifizieren kann)?
3. Wer hindert ein System daran, die Referenzen derart zu ändern, dass sie auf einen anderen LocationService zeigen (wodurch ebenfalls die Nutzer identifiziert und überwacht werden können)?
4. Wer hindert ein System daran, aus den Abfragen und Updatenachrichten die Nutzer und Ressourcen in Verbindung zu bringen (um mit diesem Wissen „lohnende" Angriffspunkte auszuwählen)?
5. Wie verhindert man illegale Updatenachrichten von Systemen, die eine falsche URL für eine ihnen über die URN bekannte Ressource eintragen möchten?
6. Wie bleibe ich sowohl als Nutzer, wie auch als Ressource gegenüber dem LocationService anonym (damit er keinen Nutzen aus diesem Wissen ziehen kann).
7. Wie verhindert man, dass ein System eine große Menge von legalen Namen sammelt um durch deren Registrierung zu einem späteren Zeitpunkt einen LocationService lahmzulegen (der LocationService muss diese illegalen Updates erkennen können)?

Einige Autoren schlagen vor, eine Menge von Servern über das Internet zu verteilen und über eine geeignete (Hash-)funktion einen dieser Server auszuwählen. Dabei wird die Anzahl der Server abhängig gemacht von der (vermuteten) Anzahl der im System registrierten Ressourcen. Obwohl auf den ersten Blick eine verlockende Idee [63], gibt es zwei Gründe gegen diesen Ansatz. 1. Alle Knoten müssen die gleiche Liste der verfügbaren (aber nicht notwendigerweise aktiven) Server haben und, 2. Einmal publiziert, darf sich diese Liste nicht mehr ändern (durch Aufnahme neuer Server bzw. Ersatz nicht mehr operierender Server). Genauer, die Länge der Liste muss gleich bleiben, da ansonsten die Abbildung bereits bestehender

Referenzen über die Hashfunktion nicht mehr gewährleistet ist. Vergleicht man das mit dem System des Internet Namensdienstes, so wären diese Server vergleichbar mit den Root-Servern im DNS System, ohne weitere Hierarchiestufen unterhalb dieser root Zone. Aufgrund dieser Einschränkungen erscheint es wenig ratsam die Auswahl eines LocationServers über eine feste Liste, kombiniert mit einer Hashfunktion zu realisieren. Neuere Forschungen auf dem Gebiet der mobilen Systeme nutzen den physischen Ort des Rechners zur Bestimmung eines zuständigen LocationServers und speichern diese Daten als Teil des Namens, wodurch sie von einer Änderung der Topologie unabhängig werden. Die dem Namen hinzugefügten Daten sind dabei nur für den Namensdienst selbst interpretierbar und verletzen damit nicht das Gebot der Orttransparenz [Steen, 2000 #41 [29]]. Damit bleibt, den Namen des gewählten LocationServices mit in die Referenz aufzunehmen. Dieser Ansatz soll im folgenden untersucht werden.

Um im Vorfeld definierten Bedingungen zu erfüllen, müssen einige Anforderungen an eine URN beachtet werden:

1. Die URN muss den gewählten LocationService enthalten (explizite Angabe des gewählten HLR) und muss gegen Veränderung geschützt sein, darf aber nicht den Namen bzw. Signatur des Besitzers der Ressource selbst beinhalten (sonst wären alle Ressourcen eines Besitzers erkennbar).
2. Weitere Aliase auf die URN müssen ebenfalls den gewählten LocationService enthalten und gegen Verfälschung gesichert sein. Sie sollten ebenfalls nicht mit dem Namen des Herausgebers signiert werden, müssen aber von einer vertrauenswürdigen Stelle signiert worden sein um sie vor Verfälschungen zu schützen.
3. Die URN einer Ressource darf sich nicht aus bekannten URNs desselben Erzeugers herleiten lassen (sonst wäre der Besitzer identifizierbar und man könnte eine URN bereits „auf Verdacht" belegen).
4. Update Nachrichten müssen geschützt sein gegen ein Replay der Registrierung (wodurch sonst ein System Ressourcen übernehmen könnte, die das System einmal besucht haben).

Man kann die ersten beiden Punkte lösen, wenn man jeweils eine unterschiedliche URN für Update und Suche vorsieht. In diesem Fall könnte ein LocationService bzw. ein lokales System zwar alle Ressourcen eines Besitzers bzw. alle dessen Referenzen mitschneiden, aber das Wissen um den Zusammenhang der gefundenen Daten fehlt, da die Zuordnung von Klienten zu Ressourcen fehlt. Wenn zusätzlich die Ressource bzw. die Referenz nicht vom Besitzer selbst signiert wird, sondern durch eine unabhängige und für verschiedene Klienten tätige Instanz, hat ein potentieller Angreifer keine Möglichkeit, Nutzer und Anbieter zu korrelieren. Als Frage bleibt, wer die Update-URN und die Such-URN erzeugt. Die Ressource selbst und auch der Besitzer sollten dies nicht implementieren. Statt dessen bietet es sich an, diese Aufgabe an den gewählten LocationService zu delegieren. Es gibt in der Literatur unterschiedliche Meinungen, ob ein LocationService für Agentensysteme hierarchisch oder flach implementiert werden sollte [63] [64]. Hierarchische Lösungen werden meistens im Zusammenhang mit dem Finden des (Agenten) Systems diskutiert. Typische Lösungen nutzen spezielle Namen im bereits bestehenden Namensdienst DNS, bzw. verzichten ganz auf eine Infrastruktur und zwingen da mit den Suchenden und Gesuchten zum Austausch dieser Information über unspezifizierte Wege. Andere Implementierungen nutzen die im DNS System bereitgestellten Service Resource Records, um das Mapping von Dienst-Namen auf einen physischen Knotennamen durchzuführen. Dies entkoppelt den Dienst vom Knotennamen und erlaubt es gleichzeitig, mehrere virtuellen Knoten auf dem gleichen physischen System zu betreiben. Flache Implementierungen haben natürlich den Vorteil, dass sie es einem Nutzer sehr schnell ermöglichen, jede Ressourcen zu finden. Dies wird erkauft mit einer nur sehr eingeschränkten Skalierbarkeit des Systems. Dennoch erscheint ein Hierarchischer Ansatz für eine große Anzahl zu verwaltende Objekte die geeignetere Wahl zu sein. Hierarchische Entwürfe haben das Problem der Skalierbarkeit nicht, dafür sind Nutzer nicht mehr in der Lage, den gesamten Suchraum direkt abzufragen, sondern müssen (selbst, oder mit Hilfe des

Systems) durch die Hierarchiestufen navigieren. Eine Kombination aus flachem und hierarchischen Ansatz ist ebenfalls denkbar. Dabei ist jedes System eindeutig über einen Record im hierarchischen Service registriert. Zusätzliche Einträge fassen die logisch zusammenhängenden Einheiten als View zusammen, womit das Gesamtsystem als Gruppe auftreten kann. Gruppenmitglieder müssten sich nicht direkt kennen, da die logische Strukturierung allein Aufgabe der Administration ist [65], und können sich automatisch durch den Gruppeneintrag finden.

Viele der angesprochenen Sicherheits-, und Privacy- Bedenken verschwinden, man das Suchen und das Finden einer Ressource so trennt, dass, selbst wenn sich Suchender und Ressource auf dem gleichen Knoten befinden, keine Komponente dies detektieren könnte. Damit sind Angriffe auf die Anonymität einer Komponente durch die Auswertung der Suchanfragen ausgeschlossen, da vom Suchnamen nicht mehr auf den Besitzer der Komponenten geschlossen werden kann. Möglich wird dies durch die Unterscheidung zwischen einem Suchnamen und einem Updatenamen. Dazu wählt eine Ressource lokal einen LocationServer und eine URN unter dem sie ihre Daten im LocationService ablegen möchte. Diesen URN übergibt sie dem gewählten LocationServer, der daraufhin lokal einen Suchnamen generiert. Er kann dazu entweder eine Tabelle mit Such- und Update- URN führen und damit diese Daten vollständig trennen, oder eine Einwegfunktion auf die gegebene URN anwenden (zusammen mit einem nur ihm bekannten Secret) und nur noch den berechneten Wert speichern. Letzteres hat den Vorteil, dass selbst wenn ein Angreifer Zugriff auf die Daten des LocationServers erlangt, er dennoch nicht auf die Updatenamen schließen kann, solange der Suchraum groß genug ist. Wie sichert aber eine Ressource den von ihr gewählte Update URN vor Veränderungen? Da sie den Namen nur selbst verwendet, reicht es, den Update-URN (UURN) derart im eigenen Code einzubetten, dass er nicht durch ein besuchtes System verändert werden kann. Eine Ressource besteht im allgemeinen aus einem unveränderlichen, sowie einem dynamischen Teil. Der Feste Bereich umfasst den Code sowie Teile der Daten. Im veränderlichen Anteil liegen all die Daten, die sich (potentiell) während der Lebenszeit der Ressource ändern (dürfen). Die Ressource legt nun den URN, ergänzt um den Namen des verwendeten LocationServers in ihren unveränderlichen Daten ab. Anschließend lässt sie den unveränderlichen Teil durch eine geeignete Instanz signieren. Dadurch weiß ein System nichts über den Eigentümer der Ressource, kann aber dennoch entscheiden, ob es der Ressource vertraut und ob die Ressource während des Transports verändert worden ist. Die Ressource selbst kann diese Signatur benutzen um Veränderungen an der eingebetteten URN festzustellen und ggf. auf Manipulationen reagieren. Der Such- URN (SURN) wird ebenfalls um den gewählten LocationService ergänzt, durch eine unabhängige Instanz signiert, und dann Klienten zur Verfügung gestellt. Sinnvollerweise wir der SURN nicht innerhalb der Ressource gespeichert, doch ist auch eine Speicherung des SURN in der Ressource keine Problem, da ein Angreifer durch die fehlende Beziehung zwischen UURN und SURN keinen Nutzen aus diesen Daten ziehen kann.

Durch die Trennung von UURN und SURN ergeben sich einige interessante Aspekte hinsichtlich der Privacy von Ressourcen und deren Nutzern. Es ist für eine andere Ressource nicht mehr möglich den Namen einer anderen Ressource durch eine eigene Implementierung zu belegen, da die andere Ressource ihren UURN frei wählen kann. Ebenso kann ein System eine Ressource nicht derart manipulieren, dass ein anderer LocationServer referenziert wird. Dazu müsste sie einen Teil des festen Datenbereichs manipulieren, was unweigerlich zur Invalidierung der Signatur führt. Selbst, wenn der Angreifer eine eigene Signatur erfolgreich an die Ressource anhängen könnte, hätte er durch diese Manipulation nichts gewonnen, da er den SURN nicht kennt (dazu müsste er mit dem im original gewählten LocationServer zusammenarbeiten). Er kann Klienten, die den Namen bereits haben, nicht manipulieren, weiß er doch nicht, welcher SURN zu dem manipulierten UURN passt. Die Integrität des in der Ressource selbst gespeicherten UURNs ist selbst dann gesichert, wenn der Name des gewählten LocationServers nicht durch eine Signatur geschützt ist. Sollte ein angreifendes System den Namen des LocationServer ändern und auf einen nicht mit dem Angreifer zusammenarbeitenden LocationServer umlenken, so müsste es vorher den original UURN dort registrieren, womit die Quelle des Update im LocationServer bekannt wäre.

dort registrieren, womit die Quelle des Update im LocationServer bekannt wäre. Wenn nun, wie bereits gefordert, die Referenz durch eine neutrale Stelle signiert wurde, kann dieser Angriff durch die nun ungültige Signatur erkannt werden. Aber selbst wenn eine Ressource ihren UURN nicht durch eine Signatur schützt und ein Angreifer diesen durch Registrierung in einem „befreundetem" LocationService und anschließender Manipulation der Referenz verändert, so hat der Angreifer dadurch nichts gewonnen, da er den SURN nicht ableiten kann, ihn aber zwingend benötigt, um etwaigen Klienten der manipulierten Ressource die veränderte Referenz unterzuschieben. Allerdings muss zugestanden werden, dass es einem Angreifer durch diese Manipulation ermöglicht wird, die Route der Ressource zu verfolgen, woraus er gegebenenfalls weitere Informationen ziehen kann. Untersuchungen müssen zeigen, ob, und ggf., wie man diesen Man-in-the-Middle Angriff verhindern kann. Das Hijacken von Ressourcen in Form gefälschter Update Nachrichten durch einen Angreifer wird durch den großen Suchraum, verbunden mit der zufälligen Wahl von UURN und SURN, verhindert. Außerdem ließe sich ein derartiger Angriff durch den LocationService selbst effektiv behindern, indem dieser einem Klienten nur eine feste (und möglicherweise geringe) Anzahl von neuen Registrierungen pro Zeiteinheit erlauben würde. Damit sind normale Updates nicht langsamer, jedoch wird jeder Versuch einen Namen zu erraten, verlangsamt und die Betriebsbereitschaft des Servers bliebe auch während eines Angriffs sichergestellt. Eine wesentliche Eigenschaft der Trennung von SURN und UURN ist die damit einhergehende Anonymität aller Beteiligten gegenüber den besuchten Systemen. Ein System kann selbst dann nicht Klienten und Anbieter von Diensten zuordnen, wenn diese auf dem gleichen System laufen. Erst wenn ein Angreifer die bei Suchen zurückgelieferten Daten auswertet und damit Referenzen auf die auf dem lokalen System vorhandenen Ressourcen findet, kann er die Daten zusammenführen. Allein aus SURNs ist dies nicht möglich. Andere Angriffspunke sind dann gegeben, wenn der Angreifer bei jeder ankommenden Ressource alle gespeicherten SURNs durch eine Read-Attack (nicht erkennbar für das Opfer) sammelt, für jede dieser SURNs eine Abfrage beim jeweils referenzierten LocationServer durchführt und aus den Ergebnissen auf die Ressourcen schließt, die gerade auf dem System laufen. Die UURNs dieser Ressourcen kann er sich dann ebenfalls durch eine Read-Attack beschaffen und somit über die Zeit die Verbindungen zwischen SURN und UURN dieser Ressourcen lernen. Dieses Problem von Read-Attacks ist allerdings nicht auf diese Art von Angriffen beschränkt. Die Frage bleibt: Wie verhindert man das bloße Lesen von Daten und deren Nutzung durch einen Angreifer? Es gibt hierzu Forschungen, doch ist bisher keine befriedigende Lösung bekannt [55]. Siehe hierzu auch Kapitel 4.2. Um einem einmal besuchten System die Möglichkeit zu nehmen, Updates zu verfälschen, muss eine Updatenachricht durch ein zusätzliches Cookie gesichert werden. Dieses Cookie wird sowohl im LocationServer selbst, wie auch in den veränderlichen Daten der Ressource gespeichert und nach jeder Update-Nachricht entweder vom LocationServer neu vorgegeben oder von der Ressource selbst berechnet und dann dem LocationServer zusammen mit dem aktuellen Update übergeben. Damit hat das verlassene System nach der nächsten Migration der Ressource keine Möglichkeit mehr den LocationServer zu manipulieren, außer, das Agentensystem und der LocationService arbeiten bei diesem Angriff zusammen. Damit bleibt schließlich die Forderung nach weitest gehender Anonymität aller Beteiligten. Dies ist durch die Einschaltung von einer durch alle Beteiligten vertrauten Instanz erreicht, die an Stelle des wirklichen Eigentümers die Echtheit der Referenzen validiert.

In diesem Kapitel wurden die verschiedenen Strategien vorgestellt, nach denen ein Namensdienst und LocationService entworfen werden können. Möglich sind flache Strukturen, sowie Hierarchische Lösungen. Flache Strukturen bieten keine Skalierbarkeit während Hierarchische Lösungen problematisch hinsichtlich der Geschwindigkeit bei Suchanfragen sind. Bei Hierarchischen Ansätzen unterscheidet man die Art, wie das Home Location Register (HLR) ausgewählt wird. Trends im Zusammenhang mit mobilen Systemen lassen es ratsam erscheinen, ortsbezogene Daten mit bei der Wahl eines LocationServers zu berücksichtigen um damit die Nachteile der Hierarchischen Speicherung teilweise zu kompensieren. Weitere Forschungen müssen zeigen, ob sich diese Methoden auch bei verteilten Systemen, in de-

nen die Migrationgeschwindigkeit um ein vielfaches höher als bei mobilen Systemen ist, anwenden lassen.

## 3.3. Konfiguration: Konsistent und Verteilt?

Agenten müssen als Nutzer der Infrastruktur eines verteilten Systems ein konsistentes Abbild der Umwelt aufbauen. Dazu werden sie unter anderem Konfigurationsinformationen nutzen, die im System für die Konfiguration der jeweiligen Komponenten vorgehalten werden. Ein diese Daten verwaltender KonfigurationService muss einige Bedingungen hinsichtlich der Verfügbarkeit und Skalierbarkeit erfüllen. Dabei kann er sowohl für die Konfiguration der lokalen Systeme selbst, wie auch, zu einem späteren Zeitpunkt, für die Konfiguration der im System migrierenden Agenten genutzt werden. Im weiteren Verlauf dieses Kapitels beschränken wir uns auf die Konfiguration der Komponenten da sich die Ergebnisse auf die Konfiguration der Agenten abbilden lassen. Als Motivation für den Entwurf eines KonfigurationServices steht die Frage, warum eine konsistente Konfiguration überhaupt erstrebenswert ist.

Im folgenden einige Gründe für die Sammlung der Daten durch einen spezialisierten Dienst:

- Fehlertoleranz durch Replikaktion bedingt Wissen von potentiellen Partnern. Dazu müssen sich die Partner finden können, aber auch den aktuellen Zustand (also Konfiguration) abgleichen können.
- Migration nur dann generisch, wenn kein Wissen über die Topologie notwendig ist. Ansonsten müssen entweder die benutzten Namen oder die möglichen Server-Standorte im Vorfeld bekannt sein.
- Verteilte Datenhaltung benötigt Wissen über die möglichen Fundorte. Ein System muss „wissen", wo es seine lokalen Daten in einer globalen Sicht bereitstellen kann oder wissen, wie es sich im globalen Verbund registrieren kann.
- Semantisches Routing ist immer dann von Vorteil, wenn die gewünschten Daten entweder – wie im web – nicht zentral vorliegen, der Nutzer an einem zentralen Punkt die Abfrage stellt, oder dann, wenn mehrere Kopien auf unterschiedlichen Systemen bereitgestellt werden und ein Lastausgleich über diese Systeme stattfinden soll.

Eine Komponente kann auf viele Arten konfiguriert werden. Im einfachsten Fall liegen Konfigurationsdaten und die zugehörige Komponente auf dem gleichen System. Der Zugriff geschieht über die normalen Dateifunktionen des lokalen Betriebssystems. Wenn mehrere Komponenten auf dem gleichen Knoten liegen, kann die Speicherung aller Daten an einer gemeinsamen Stelle im Dateisystem sinnvoll sein. Dies kann durch die Verwendung eines gemeinsamen Verzeichnisses realisiert werden, in dem jede Komponente die sie betreffenden Daten ablegt. Alternativ kann ein lokaler Dienst ein Interface für die lokalen Komponenten bereitstellen und die Daten performant und sicher verwalten. Sinnvollerweise werden hierbei die Daten in einer lokalen Datenbank abgelegt. Bei mehreren physischen Knoten müssen die Konfigurationsdaten nicht zwingend auf dem lokalen System gespeichert werden. Ebenso ist ein Service denkbar, der die zentrale Speicherung der Konfigurationsdaten erlaubt. Damit muss auf dem lokalen System nur noch der Name und die Adresse des zentralen Services bekannt gemacht werden, während die restliche Konfiguration über die zentrale Stelle erfolgt. Ein zentraler Service ist bei mehreren Knoten nicht die einzige Lösung. So ist auch die verteilte Speicherung der Konfigurationsdaten über mehrere Knoten hinweg vorstellbar. Hierzu wird der Service auf mehreren Knoten gestartet, die eigentlichen Daten auf die beteiligten Knoten aufgeteilt, und gegebenenfalls repliziert. Konfigurationsdaten sind zwar in der Regel nur für die Komponente selbst interessant, doch wird häufig auch ein Zugriff anderer Komponenten auf die gespeicherten Daten benötigt. Dies liegt an der Notwendigkeit der Änderung einer bestehenden Konfiguration durch ein von der jeweiligen Komponente mitgeliefertes Programm, oder durch eine generische Applikation. Aus diesem Grund ist es wichtig, die Form der Speicherung, sowie Semantiken

Grund ist es wichtig, die Form der Speicherung, sowie Semantiken für den parallelen Zugriff auf die Daten verbindlich festzulegen. Nicht alle vorgestellten Strategien sind für diesen Zweck geeignet. Solange man der Komponente selbst die Speicherung ihrer Daten, möglicherweise in einem gemeinsamen Verzeichnis, erlaubt, ist weder eine festgelegte Form der Daten, noch der parallele Zugriff darauf garantiert. Ein lokaler, bzw. zentraler Service hingegen erfüllt beide Forderungen: Die Daten werden in einem einheitlichen Format abgelegt und da der Zugriff über genau ein Interface geschieht, ist auch die Serialisierung der Zugriffe durch diese zentrale Stelle möglich. In einem verteilten Service hängt das gemeinsame Format der gespeicherten Daten davon ab, dass alle am verteilten Dienst beteiligten Prozesse die gleiche Form bei der Speicherung benutzen. Der parallele Zugriff auf die Daten muss entweder über aufwändige verteilte Synchronisationsmechanismen oder durch eine geschickte Wahl bei der Verteilung der Daten sichergestellt werden. Welche Datenformate sollte das Interface bereitstellen? Eine Antwort ist schwierig, ist doch gerade die Beziehung zwischen reicher Beschreibung und effizienter Implementierung ein Problem. Austauschsprachen wie XML, welche die Syntax der Attribute selbst als Beschreibung definieren, können dieses Problem teilweise lösen, da sie eine Installation neuer Beschreibungen (DTD) erlauben. Damit kann zur Laufzeit ohne Änderung der bestehenden Interfaces die Menge der unterstützten Konfigurationsbeschreibungen erweitert werden. Man muss sich aber vor Augen halten, dass genau dies der Forderung nach einem einheitlichen Format der Konfiguration widerspricht. Außerdem legt eine DTD nicht die Semantik der einzelnen Konfigurationsparameter fest. Dies erlauben erst weitere Sprachen [6].

Gibt es Probleme hinsichtlich Performance und Skalierbarkeit? Der Lösung, alle notwendigen Daten auf den lokalen Systemen zu speichern, benötigt keinen (zentralen) KonfigurationService. Dafür ist eine komponentenübergreifende Konfiguration nicht möglich, außer, die gemeinsamen Attribute werden in jeder lokalen Datenbasis repliziert, was einerseits aufwändig, andererseits fehleranfällig ist. Auch verlangt dieser Ansatz eine ständige Erreichbarkeit aller Komponenten da nur eine aktive Komponente konfigurierbar ist, was ressourcenfressend ist. Als vorteilhaft erweist sich der Zugriff über lokale Dateien: Damit ist ein Zugriff immer lokal, schnell und deterministisch. Auch hinsichtlich der notwendigen Ressourcen für den Zugriff ist diese Lösung verlockend, da keine besonderen Ansprüche gestellt werden. Die Verwendung eines gemeinsamen Verzeichnisses zur Speicherung der Daten bringt keine Vorteile hinsichtlich Skalierbarkeit und Ressourcenverbrauch. Erst durch die Verwendung eines lokalen Servers (evtl. mit Datenbank) ergeben sich für das System einige Verbesserungen. Zum einen muss eine Komponente nicht mehr ständig aktiv sein um anderen ihrer aktuelle Konfiguration mitteilen zu können. Dies schont Ressourcen, da nun inaktive Komponenten aus dem Speicher entfernt werden dürfen. Zusätzlich erlaubt ein zentraler Dienst den kontrollierten, parallelen Zugriff auf die dort gespeicherten Daten, was aber mit einem Single-Point-Of-Failure erkauft wird: Fällt der lokale Dienst aus, fallen ebenfalls alle abhängigen Komponenten aus (Dies sind alle Komponenten, die ihre Konfigurationsdaten über diesen Dienst verwalten und nach dem Ausfall des Dienstes Werte Abfragen, bzw. Setzen müssen). Ist jedoch sowohl die Anzahl installierter Komponenten überschaubar, wie auch die Häufigkeit von Änderungen gering, so stellt dies eine performante und aus Sicht eines Administrators bequeme Art der Verwaltung dar. Wird hingegen die Anzahl der Komponenten größer, stößt dieses System an Grenzen. Dies wird verursacht durch die große Anzahl lesender Zugriffe, die steigende Belastung durch Änderungen, sowie die fehlende Möglichkeit, veraltete und überflüssige Einträge zu erkennen und zu entfernen. Da der Server keine Möglichkeit hat, tote Einträge zu erkennen, muss er alle Einträge auf unbestimmte Zeit vorhalten. Ein zentraler Dienst in einem System mit mehreren Knoten ist vom Verhalten vergleichbar dem lokalen Dienst bei nur einem Knoten. Da für diesen Fall Gesagte gilt auch hier, wobei sich das Problem hinsichtlich der Ausfallsicherheit und Performance durch den Anstieg aktiver Komponenten noch verstärkt. Es kommt noch das Netzwerk hinzu, das ebenfalls Quelle zusätzlicher Probleme ist. Die Nutzung eines Clusters [41] entschärft dies zwar, trägt jedoch nicht zur Beseitigung des Bottlenecks bei.

Ein verteilter Service kombiniert Stärken (und teilweise Schwächen) der bekannten Lösungen. Durch die Partitionierung der Daten auf mehrere Server erhöht sich die Ausfallwahrscheinlichkeit des gesamten Dienstes um die Anzahl der für den Dienst notwendigen Systeme. Dafür erhöht die Zugriffsgeschwindigkeit (bei „guter" Wahl der Knoten) beim Suchen/Update. Die Replizierung der Daten verringert die Ausfallwahrscheinlichkeit. Solange ein System noch aktiv ist, auf dem die gesuchten Daten vorliegen, merkt ein Klient nichts vom Ausfall einiger Komponenten. Das gleiche gilt für die Zugriffsgeschwindigkeit beim Suchen, da sich mehrere Server diese aufwändige Operation teilen können. Allerdings verringert sich durch die Replizierung die Zugriffsgeschwindigkeit beim Update durch die notwendige Synchronisierung aller Server, auf denen sich (nun veraltete) Kopien dieser Daten befinden. Nachdem wir nun typische Stärken und Schwächen gesehen haben, halten wir die gewünschten Attribute eines verteilten KonfigurationServices fest:

- Das Antwortverhalten darf nicht von der Anzahl der gleichzeitig im System aktiven Komponenten abhängig sein.
- Abfragen müssen „billig" sein.
- Updates dürfen „teuer" sein.
- Die Unterstützung von parallelen Operationen darf den einfachen Zugriff nicht verteuern. (Der Regelfall muss der schnellste Fall sein.)
- Die Kosten beim parallelen Zugriff auf die gleichen Daten dürfen für die beteiligten Komponenten hoch sein.
- Migration der Daten möglichst nah an den physischen Aufenthaltsort des Nutzers.
- Daten müssen aus Performancegründen gecached werden.

In dieser Aufstellung ist die Verschiebung von Daten an den physischen Aufenthaltsort des Nutzers ein interessanter Aspekt. Bisherige Ansätze ziehen keinen Nutzen aus dem Wissen des physischen Aufenthaltsortes ihres Nutzers, wobei es nicht einmal auf eine wirklich genaue Positionsangabe ankommt. Vielmehr reicht eine Unterteilung in Regionen, um die Konfigurationsdaten nah an den Knoten des hauptsächlichen Nutzer zu bringen. Wie schon bei der Migration und denen sich daraus ergebenden Bedingungen an die Server, liegt auch hier der Schwerpunkt auf der möglichst (örtlich) lokalen Interaktion des Klienten und des den Dienst implementierenden Services. Dazu kann, wie auch schon beim LocationService, sowohl ein flacher Ansatz, wie auch eine hierarchische Organisation der den Dienst bereit stellenden Server gewählt werden. Weitere Forschungen müssen zeigen, ob die beim LocationService gebrachten Gründe für eine hierarchische Speicherung auch hier anwendbar sind.

Ein direkter Vergleich vom KonfigurationService zum LocationService drängt sich auf:

| **Frage** | **KonfigurationService** | **LocationService** |
|---|---|---|
| Zugriffsverhalten (Read/Write) | 80R/20W | 20R/80W |
| Konkurrierende Zugriffe | Ja | Nein |
| Serverwahl | nah am physischen Objekt | nah am physischen Objekt |
| Tote Einträge detektieren | Ja | Ja |
| Speicherort wandert mit Objekt | Ja(Sinnvoll) | Ja(Notwendig) |
| Sicherung (Updates/Queries) | Ja(U)/Ja(Q) | Ja(U)/Nein(Q) |
| Kosten (Updates/Queries) | Teuer(U)/Billig(Q) | Billig(U)/Billig(Q) |
| Synchronisierung | Teuer | Nicht anwendbar |

Somit bleibt abschließend der Eindruck, dass man KonfigurationService und LocationService auf einer gemeinsamen Basis entwerfen kann. Es muss sich zeigen, ob sich die gefundenen Gemeinsamkeiten gewinnbringend nutzen lassen und inwieweit die Unterschiede (speziell die Unterstützung konkurrierender Zugriffe) einer einheitlichen Implementierung im Wege stehen.

# 4. Spielregeln beim Zusammenspiel verschiedener Agenten

Im vorangegangen Kapitel lag der Schwerpunkt auf der Evaluierung einer für mobile Agenten geeigneten Infrastruktur. In diesem Kapitel bewegen wir uns auf einer abstrakteren Ebene. Hier wird mit der vorgestellten Infrastruktur Lösungen für Problemen auf Agentenebene gesucht. Damit steht jetzt die Interaktion Umgebung/Agent bzw. Agent/Agent im Mittelpunkt.

## 4.1. Ressourcen und Kosten

In einem physisch verteilten System liegen die Rechte an den einzelnen Komponenten, bzw. Diensten in der Regel bei mehreren Kostenträgern. Dies hat zur Folge, dass Agenten, die sich nicht auf ihrem Heimatknoten befinden (in der Regel aber selbst dort) für in Anspruch genommene Dienste eine Gegenleistung erbringen müssen. Dies kann auf Basis eines Geschenks geschehen („Ich habe die Rechenleistung übrig. Nimm sie Dir, wenn Du sie brauchst."), eines Tauschgeschäfts („Ich bekomme Rechenzeit vor Dir, dafür bekommst Du das Ergebnis meiner Berechnungen (auch)") , auf Versprechen („Ich nutze Deinen Prozessor und wenn ich das Ergebnis auf Deinem Rechner als erster finde, bekommst Du dafür einen Preis"), auf Vorauskasse („Wenn Du mir n-Einheiten bezahlst, kannst Du bei mir für m-Minuten rechnen"), oder auf Rechnung („Wenn Du m-Minuten bei mir rechnest, dann kostet das Deinen Besitzer n-Einheiten"). Alle diese Ansätze haben ihre spezifischen Vor- und Nachteile die im folgenden vorgestellt werden sollen.

Das kostenlose Anbieten ist für den Nutzer eines Dienstes die günstigste Lösung wenn man allein die absoluten Kosten der Dienstleistung berücksichtigt. Wenn jedoch auch andere Parameter, wie beispielsweise die Dauer bis zu einem Ergebnis, die Genauigkeit oder die Sicherheit der Daten vor Ausspähung, in die Kostenfunktion eingehen, sind Dienste, die Kosten verursachen, dafür aber entsprechende Zusagen bieten, in der Summe günstiger. Oft kommt es aber nicht auf garantierte Diensteparameter an, und selbst große Projekte, wie das Seti@home Projekt und das RC5 Contest Projekt leben davon, dass die Nutzer dem jeweiligen Agenten kostenlos Rechenzeit auf ihrem lokalen System, sowie freien Netzzugang zum Austausch der Rohdaten und Ergebnisse zur Verfügung stellen [66] [67]. In einem kommerziellen Umfeld werden sich dennoch selten kostenlose Dienstleistungen finden, weshalb sie im weiteren nicht näher betrachtet werden sollen. Eine speziell im Offline Bereich bekannte Methode ist das Bezahlen mittels Geld. Den Fall der Bezahlung in Waren (Naturalien) kann man als Spezialfall des Bezahlens mit Geld sehen und soll aus diesem Grund nicht einzeln betrachtet werden.

Wie kann nun ein Agent für in Anspruch genommene Dienste bezahlen, bzw. wie kann er für die Erbringung von Diensten in Rechenschaft gezogen werden:

1. Der Agent führt das Geld zum Bezahlen der Ware/Dienstleistung mit sich und überträgt es der entsprechenden Ressource nach dem Erbringen des Dienstes. Hierbei ist durch geeignete Maßnahmen zu verhindern, dass der Agent beraubt wird. Ein Agent selbst kann weder das wiederholte Ausgeben seines Geldes verhindern, noch das gezielte Ersetzen der gefüllten Geldbörse durch eine leere Börse mit Hilfe einer Manipulationsattacke erkennen.
2. Der Agent führt speziell codiertes Geld für jede in Anspruch genommene Dienstleistung mit sich. Damit kann ein Dieb zwar das Geld stehlen, aber keine Dienste in Anspruch nehmen, die nicht auch der wirkliche Eigentümer nutzen wollte. Dieser Ansatz ist dann sinnvoll, wenn die ungefähren Kosten und alle zu besuchenden Dienste im voraus bekannt sind. Verbinden ließe sich dies beispielsweise mit dem Iternity Ansatz zur Planung einer Reise eines mobilen Agenten: Da die Reiseroute schon vor Beginn der Reise bekannt ist, kann ein Agent auch die voraussichtlich notwendigen Geldmittel mitführen oder zumindest das Geld derart codieren, dass es nur auf den zu besuchenden Knoten gültig ist.

3. Der Agent führt präpariertes Geld mit sich, dessen Gültigkeitsdauer beschränkt ist und bei dem der Verkäufer der Ware lokal entscheiden kann, ob es für die aktuelle Transaktion verwendbar ist. Der Vorteil ist, dass auch bei Verlust des Agenten nach einer endlichen Zeitspanne davon ausgegangen werden kann, dass kein weiteres Geld mehr ausgegeben wurde (es wäre sonst beim Herausgeber oder dessen Stellvertreter in dauerhafte Währung eingetauscht worden). Problematisch ist hierbei nur, dass die Uhren im System relativ eng gekoppelt sein müssen, damit alle beteiligten Parteien der gleichen Meinung über die Restgültigkeit der Geldeinheiten sind.
4. Der Agent führt nur eine „Kreditkarte" mit sich. Die Ressource stellt gegenüber dem Herausgeber der Kreditkarte ihre Dienste in Rechnung (entweder vor oder nach Inanspruchnahme durch die Klienten). Vorteil: Betrügereien sind zwar weiterhin möglich, aber der Angreifer wird identifizierbar, da er sich gegenüber dem Herausgeber der Kreditkarte identifizieren muss wenn er die Schuld einzulösen versucht. Durch die Verwendung von Quittungen, die Anbieter ihren Kunden über die Nutzung der Ressource ausstellen, können weitere Missbrauchsversuche erkannt werden. Zusätzlich verhindert eine Gültigkeitsdauer den Zeitraum, in dem Missbrauch mit der Karte betrieben werden kann.
5. Der Agent selbst ist die Währung. Ein Dienst generiert mittels Algorithmus einen Hashcode über den unveränderlichen Code des Klienten und kann daraus die Kreditwürdigkeit ableiten. Dies würde beispielsweise durch die Meldung des Hashcode an das Kreditkartenunternehmen möglich sein oder die Migration auf einen „Kassen"-Knoten, von dem aus ein Bezahlen möglich ist. Ein Angreifer hätte keine Möglichkeit einen erschlichenen Code zu nutzen solange das System nicht selbst der Angreifer ist.

Keiner dieser Ansätze löst das Problem von durch Lese-Attacken gestohlener Referenzen auf das Geld bzw. die Kreditkartennummer. Es ist einem Angreifer immer möglich, dem Opfer ohne dessen Wissen eine Kopie der Daten zu entwenden. Dies unterscheidet die Problematik grundsätzlich von allen gerne gezogenen „Real World" Vergleichen, in denen die Daten den Besitzer nicht ohne dessen Wissen verlassen (er muss die Transaktion ausführen). Implementierbar erscheint das sichere Bezahlen von Dienstleistungen mit Kreditkarte und Sammeln der Quittungen im Agenten mit anschließender Rückkehr zum Besitzer, selbst in einer ungesicherten Umgebung. Wenn das lokale System vertrauenswürdig ist, kann der Agent ein initiales Secret mit seinem öffentlichen Schlüssel verschlüsseln und zusammen mit dem öffentlichen Schlüssel in seinem Datenbereich ablegen (den Schlüssel im unveränderlichen Teil der Daten; das initiale Secret in den veränderlichen Daten.). Bei jeder Nutzung einer kostenpflichtigen Ressource bekommt der Agent eine elektronische Quittung über die Transaktion. Diese Quittung hängt der Agent nun an das bereits gespeicherte Secret an und verschlüsselt das **gesamte** Paket erneut mit dem öffentlichen Schlüssel. Ein Angreifer kann nun zwar die Daten löschen oder gegen eine bei einem vorherigen Besuch gespeicherte Version austauschen. Was er jedoch nicht kann, ist die Manipulation einzelner Quittungen, da nicht einmal mehr der Agent selbst die Möglichkeit hat, diesen Teil seiner Daten zu dechiffrieren. Allerdings kann ein angreifendes System diesen öffentlichen Schlüssel durch einen eigenen austauschen, um dann im weiteren Verlauf beim erneuten Besuch des Agenten alle bis dahin gesammelten Daten zu manipulieren. Dieses Angriffsmuster kann nur dann erfolgreich sein, wenn es dem Angreifer gelingt, den Agenten vor dessen Rückkehr zum Besitzer noch einmal auf das eigenen System zu lenken. Gelingt dies nicht, kann der Besitzer die manipulierte Liste erkennen. Wenn der Agent zu seinem Besitzer zurückkehrt, wird er die gespeicherte Liste mitbringen. Der Besitzer kann die Liste durch seinen geheimen Schlüssel wieder Stück für Stück entschlüsseln und am initialen Secret erkennen, ob jemand die Liste durch eine andere, manipulierte Version ausgetauscht hat.

Es gibt bereits Erfahrungen mit dem Aufbau einer Zertifizierungsstelle und dem sicheren Verteilen der durch dieses System verifizierten Schlüssel. Bisherige Systeme gehen davon aus, dass ein Individuum ein „Secret" besitzt, dass durch keine weitere Instanz gelesen werden kann. Aktuelle Forschungen müssen zeigen, in wieweit bekannte Systeme auf den Fall er-

weiterbar sind, dass verhindert werden kann, dass Code (und damit auch jedweder Schlüssel) durch ein angreifendes System nicht detektierbar ausgelesen werden kann.

## 4.2. Sicherheitsaspekte

In einem System in dem die einzelnen Knoten nicht unter einer administrativen Einheit stehen, kann für die Sicherheit der Komponenten nur schwer garantiert werden. Zu vielfältig sind die möglichen Angriffsmuster potentieller Angreifer. Das Problem liegt darin, dass sich ein mobiler Agent nicht einmal mehr dann auf die Integrität der ihn beherbergenden Umgebung verlassen kann, wenn sich diese vorher ausgewiesen hat. Gleiches gilt für einen angreifenden Agenten. Daher ist es sinnvoll, die Komponenten und das System voreinander abzuschotten, um so möglichst wenige Angriffspunkte zu bieten.

Eine Schutz der Komponenten ist an diesen Stellen sinnvoll:

1. Schutz des Systems/ lokale Ressourcen vor angreifenden Agenten
2. Schutz zwischen unterschiedlichen Agenten im gleichen System
3. Schutz von Agenten vor angreifendem System

Der Schutz von Ressourcen vor einem angreifenden Agenten ist vergleichbar mit dem Schutz lokaler Ressourcen vor dem Missbrauch durch Prozesse. Hierbei ist darauf zu achten, dass ein Prozess keine Monopolisierung einer Ressource erreicht, indem er alle verfügbaren Einheiten auf sich vereinigt. Effiziente Lösungen zu diesen Problemen sind bekannt. Ähnliches gilt für den Schutz unterschiedlicher Agenten im gleichen System. Man abstrahiere einen Agenten zu einem Prozess und reduziert das Problem so auf die Abschottung verschiedener Prozesse voneinander. Dieser Schutz wird von fast allen bekannten Server- Betriebssystemen geboten. Bleibt noch der Schutz vor denial of service (DOS) Attacken, bei denen eine große Menge voneinander unabhängiger Rechner das Opfer gezielt durch Absetzen zahlreicher und umfangreicher Aufträge in die Knie zu zwingen versuchen. Ein direkter Schutz ist nicht möglich, dennoch können die Folgen dieses Angriffstyps meist gemildert werden. Dazu werden die maximale Anzahl, der einem Auftrag zugeordneten Ressourcen, sowie die Anzahl der parallel möglichen Aufträge limitiert und der Zugriff priorisiert, womit ein Angreifer den Zielknoten nicht mehr komplett belegen kann.

Wenn man aber, wie im dritten Fall, annimmt, dass die Laufzeitumgebung selbst angreift, sind Manipulationen möglich, die sich grob in drei Klassen einteilen lassen:

1. Modifizierende Angriffe (ändern den Agenten Code/Daten)
2. Lesende Angriffe (liefern sensitive/geheime Daten des Agenten)
3. Täuschende Angriffe (liefern bei Aufrufen falsche/modifizierte Daten zurück um ein bestimmtes Verhalten auszulösen. Benutzen dazu das Wissen, dass sie aus den beiden vorangegangenen Angriffsmustern gewonnen haben)

Jede dieser Klassen hat ihre besonderen Eigenheiten, auf die ein Agent achten muss, will er sie erkennen bzw. vermeiden. Modifizierende Angriffe auf den Code kann ein Agent detektieren wenn er an dritter Stelle eine Signatur über den unversehrten Code hinterlegt hat. Diese Signatur kann er zu gegebener Zeit mit der aus seinem aktuellen Code generierten, aktuellen Signatur, durch die dritte Stelle vergleichen lassen indem er zuvor auf ein vertrautes System migriert. Es reicht nicht, den Test auf einem potentiell unsicherem System auszuführen, kann dieses doch durch einen täuschenden Angriff die erfragten Daten vor der Rückgabe an den Klienten modifizieren. Problematisch bleibt der Schutz der sich verändernden Daten des Agenten. Um diese wirkungsvoll vor Veränderung zu schützen – unter Voraussetzung, dass das System keinen Täuschenden Angriff durchführt – kann ein Agent periodisch Hashcodes über diese Daten anfertigen, diese mit einem im unveränderlichen Teil des Codes mitgeführ-

ten, öffentlichen Schlüssel verschlüsseln und auf einem vertrauten System ablegen. Ein Erkennen von Veränderungen zum Zeitpunkt der Manipulation ist so nicht möglich und es ist auch nicht garantiert, dass alle Manipulationen schließlich entdeckt werden können. Lesende Angriffe zu verhindern ist nicht möglich, da das Ausführen von Code immer auch das Leserecht beinhaltet. Daher muss ein Agent, der fürchtet Opfer einer derartigen Attacke zu werden, auf dem letzten noch vertrauten System all die Daten zurücklassen, deren Offenlegung er verhindern möchte. Ein solcher Knoten existiert immer, schließlich ist das System, auf dem der Besitzer den Agenten instrumentiert hat, implizit vertrauenswürdig. Um indirekt auf einen Lesenden Angriff zu schließen, kann man auf die Folgen der durch die Nutzung dieser Daten verursachten Kosten triggern. Wenn der Zeitraum der Gültigkeit der Daten beschränkt ist, kann ein Agent seinen Code verschleiern um, die Unversehrtheit der Daten für einen begrenzten Zeitraum zu garantieren. Dazu wird der Code so verändert, dass er zwar weiterhin ausführbar ist, es einem Leser aber nicht mehr ohne weiteres möglich ist, den Programmfluss nachzuvollziehen. Weiterhin können Methoden eingesetzt werden, die das Mitverfolgen des Programmablaufs zur Laufzeit verhindern, bzw. detektierbar machen [68]. Die Frage bleibt, in wieweit eine realistische Abschätzung über die Dauer des Schutzes gemacht werden kann, aus der sich die Gültigkeitsdauer des Geldes ergibt. Sind Lesende Angriffe ein Sicherheitsrisiko und ist, bedingt durch die Langlebigkeit des Agenten, die Verschleierung der Daten ebenfalls keine Option, muss sich das Zielsystem vor der Migration als „sicher" ausweisen. Dies kann durch die Vergabe einer Signatur durch eine Zertifizierungsstelle sein, oder die Nutzung sogenannter Black-Box Systeme, auf denen ausschließlich ein Agentensystem läuft und die gegen (physische) Manipulationen geschützt sind. Auch ein Black-Box System muss sich zusätzlich durch einen Schlüssel ausweisen, der aber, bedingt durch die geschlossene Konstruktion, wesentlich besser geschützt werden kann [69].

Die Art des dritten Angriffsmusters beruht auf einer Verknüpfung der beiden vorgenannten Ansätze. Dabei werden manipulierender Angriff und lesender Angriff zu einem täuschenden Angriff kombiniert. Einen täuschenden Angriff kann ein Agent selbst dann nicht erkennen, wenn er über Methoden verfügt um die vorgenannten Attacken abzuwehren. Das Problem liegt in der Möglichkeit des Systems, dem Agenten eine Umgebung vorzutäuschen, in der er darauf verzichtet, zusätzliche Sicherungsmittel einzusetzen. Auch kann das angreifende System dem Opfer suggerieren, dass es die zu Erhöhung der Sicherheit notwendigen Funktionen aufgerufen hat. Beispielsweise kann der Aufbau eines sicheren Kanals dadurch verhindert werden, dass der originale Schlüssel durch eine Leseattacke aus dem Code extrahiert und durch einen manipulierten, bekannten Schlüssel des Angreifers ersetzt wird. Routinen, welche die Authentizität des Schlüssels überprüfen könnten, werden durch eigene, immer das erwartete Ergebnis zurückliefernde Versionen ersetzt. Dadurch wird eine Man-in-the-Middle Attacke möglich, bei der ein Angreifer die Daten seines Opfers schon vor der eigentlichen Übertragung lesen und ggf. modifizieren kann. Ein Angreifer kann aber in der Übertragungskette noch weiter vorne ansetzen. Dabei versucht er nicht den Aufbau des sicheren Kanals oder die Verschlüsselung der Daten zu verhindern, sondern zieht Nutzen aus der Tatsache, dass das Opfer Routinen des Systems zur Verschlüsselung nutzt, bzw. derartige Routinen als Teil des eigenen Codes enthält. Der Angreifer kann diese Funktionen durch eigene Routinen ersetzen, um dann Klartext und verschlüsselten Text zu speichern. Das Dekodieren der übertragenen Nachricht besteht dann nur noch aus einem simplen Pattern matching bei dem die bei der Übertragung abgefangenen Daten mit den bei der Verschlüsselung mitgeschnittenen Daten verglichen werden. Das Opfer konnte ja nicht nur Teile der Verschlüsselten Daten übertragen, sondern auch das Opfer muss die verschlüsselten Daten an den durch die Funktionsaufrufe gegeben Grenzen aufbrechen.

Letztendlich stellt sich die Frage, wie man auf den Bruch des Vertrauensverhältnisses reagieren sollte, kann man die mögliche Verletzung nicht verhindern. Es gibt verschiedene Strategien, wie man auf das Erkennen eines Betrugs reagieren kann. In seinem Buch „The Evolution of Cooperation" hat Robert Axelrod [70] vier Grundregeln beschrieben, die jeder Teilnehmer eines Spiels beherzigen sollte. Grundlage ist das so genannte „Gefangenen Problem". Hierbei sitzen zwei Gefangene in einer Zelle und wollen ausbrechen. Dies können sie

nur gemeinsam. Jeder Gefangener hat die Wahl, mit dem anderen zusammen zu arbeiten, oder den Ausbruchsversuch zu melden. Das entscheidende Problem ist „traue ich dem anderen?". Die Entscheidung basiert allein auf eigenen Beobachtungen. Als beste Strategie hat sich „Auge um Auge" („TIT FOR TAT") herausgestellt. Dies bedeutet, der Gefangene verhält sich in seiner nächsten Entscheidung so, wie sein Gegenüber in der Runde vorher. Solange ein Mitspieler nicht betrügt, wird auch der Andere nicht damit anfangen. Betrügt jedoch der Gegenspieler, wird er in der nächsten Runde ebenfalls betrogen. Der Grund für den Erfolg dieser Strategie liegt darin, dass sie unnötige Konflikte durch eine möglichst lange Kooperation mit dem anderen Spieler vermeidet. Provokation in Form von Betrug, wird nur nach vorangegangenem Betrug des Anderen angewendet, danach wird diesem aber sofort wieder vergeben. Schließlich ist wichtig, dass das Handeln für den anderen „durchsichtig" und damit kalkulierbar bleibt.

Dies führt zu vier Regeln für eine gute Strategie, bei der sowohl die eigenen Ziele, wie auch die Bedürfnisse des anderen berücksichtigt werden:

1. Sei nicht neidisch
2. Sei nicht der erste, der betrügt
3. Erwiedere sowohl Kooperation, wie auch Betrug
4. Sei nicht zu clever

Auffällig ist der letzte Punkt, widerspricht er doch der Gewohnheit, auf Probleme mit weiteren Regeln - und damit komplexeren Handlungsmustern – zu reagieren. Die Frage stellt sich, wie ein Gesamtsystem entworfen werden muss, dass sich möglichst viele Teilnehmer auf diese Verhaltenmuster einlassen. Axelrod führt mehrere Punkte an, die ein System als ganzes positiv unterstützen sollte, um langfristig sich „sozial" verhaltende Agenten zu fördern:

1. Mache die Zukunft wichtiger als die Gegenwart
2. Erhöhe den Anreiz für eine langanhaltende Zusammenarbeit
3. Bringe den Teilnehmern bei, auf den anderen zu achten
4. Vermittle den Teilnehmern die Wechselwirkung ihrer Interaktion
5. Verbessere die Fähigkeit zur Wiedererkennung anderer Teilnehmer

Dieses Verhalten ermöglicht es einem Teilnehmer nicht unbedingt „der Beste" im Gesamtsystem zu werden, wie man dies in einem Spiel wünschen würde. Doch lässt sich eine wirkliche Welt selten als ein System von „Gut" und „Schlecht" beschreiben, dessen Summe sich gerade aufhebt. Vielmehr überwiegt eine Seite und es kommt als Mitspieler darauf an, viele Punkte auf der „Gut" Seite zu sammeln. Mit den oben beschriebenen Mitteln wird genau dies erreicht. In welcher Form diese Ansätze durch ein Agentensystem forciert werden können, müssen weiteren Untersuchungen zeigen.

## 4.3. Workflow- Unterstützung

In den bisherigen Kapiteln wurde davon ausgegangen, dass sich die Komponenten „auf eine geeignete Art und Weise" verständigen können. Speziell wurde nicht hinterfragt, was passiert, wenn bis dato unbekannte Komponenten miteinander auf ein Ziel hinarbeiten müssen. Dies ist gar nicht so ungewöhnlich und Beispielsweise dann der Fall, wenn ein Besprechungsplaner mit den Teilnehmern einer Besprechung nach einem gemeinsamen, freien Zeitraum für die Besprechung sucht. Hierbei wird der Planer mit unbekannten und eigene Ziele verfolgenden Personen bzw. deren Stellvertretern verhandeln um sein gewünschtes Ziel (die Besprechung) zu realisieren. Seine Wahrnehmung der relevanten Umwelt (bisherige Termine der Gesprächspartner, freie Zeitbereiche) ist lückenhaft, eventuell sogar inkonsistent oder falsch. Seine Möglichkeiten, die Umwelt zu beeinflussen, sind begrenzt und selbst eine Einigung bedeutet das nicht, dass die abgesprochenen Fristen auch eingehalten werden. Er muss also – wie im richtigen Leben – kurz vor dem Termin noch mal sicherstellen,

dass alle Teilnehmer wirklich Bescheid wissen. Die Modellierung der Arbeitsabläufe eines Unternehmens in Software zeigt noch zusätzliche Aspekte. So kann man Planungsaufgaben meist auch auf ein automatisiertes System abbilden, wobei Planer, Stellvertreter und Besprechungsräume eigene, selbständig agierende Agenten sind, die gemeinsam auf die Planungsziele hinarbeiten. Man kann diese Aufgabe mit den „klassischen" Ansätzen zur Programmierung in einem verteilten System lösen, doch würde es spätestens bei einem Wechsel der Abläufe zu Änderungen im Programmcode der bereits bestehenden Komponenten kommen um die neue Strategie (zum Beispiel muss eine zusätzliche Instanz die Planung ebenfalls absegnen, bevor sie gültig wird) zu realisieren. Diese Änderungen sind aber nur das Symptom: An Stelle einer neuen Komponente müssen bereits bestehende Komponenten angepasst werden. Eine weiterer Grund für eine unabhängige, neue Komponente ist, dass bisherige Varianten parallel genutzt werden können. In diesem Umfeld spielen Agenten den Vorteil, dass ihr Design speziell auf derartige Probleme „passt", soll heißen: Das Believe-Desire-Intention [1] Modell kann gerade dazu genutzt werden, um dynamische und nicht in jedem Detail vorhersagbare Handlungsabläufe zu modellieren, und auch in einer Umgebung zu agieren, in der ständig neue und andersartige Abläufe parallel zu bereits eingesetzten Methoden eingesetzt werden müssen. Für die Unterstützung von Planungsabläufen gibt es umfangreiche Forschungen. Dabei kann unterschieden werden zwischen Systemen, die eine Planungsphase als eigenständigen Punkt vorsehen, und solchen Systemen, bei denen die Planung zu jeder Zeit den Gegebenheiten angepasst werden kann. Stellvertretend für letzteres sei hier [71] genannt, ein System zur Unterstützung eines Software Entwicklungsprozesses inklusive möglicher Änderungen des Plans während der aktiven Projektphase.

Was zeichnet eine Unterstützung eines Workflows aus? Um diese Frage speziell für das Broadcast Umfeld zu beantworten, betrachte man als Beispiel einen Arbeitsablauf, der den Weg vom Einlaufen des Rohmaterials, über dessen Aufbereitung bis hin zum Grobschnitt des fertigen Beitrags nach einer Suche durch den Redakteur umfasst. Im folgenden wird exemplarisch der Ablauf dieses Workflows aus der Sicht mehrerer Komponenten vorgestellt [72] [73]. Das System besteht aus den folgenden Einheiten:

- Feed – Über diesen Eingabepunkt erreicht neues Material die Redaktion. Mehrere Feeds können existieren, jedoch kann zu einem Zeitpunkt nur ein Beitrag gleichzeitig über einen Feed übertragen werden.
- LowRes Kopie – Wird in der Regel durch Liveencoding der angeschlossenen Signalquelle realisiert. Am Eingang liegt das sendefähige Material, am Ausgang wird die niedrig bitratige Kopie bereitgestellt.
- HiRes Kopie – Diese Kopie wird durch das Aufzeichnen des Signals auf einem Videorecorder bereitgestellt. Dazu muss ein Band im Recorder liegen, und am Eingang das aufzuzeichnende Signal.
- EssenceManager – hier wird das gesamte LowRes Material abgelegt und verwaltet. Intern wird das Material auf einem HSM System gespeichert und die aktuell notwendigen Teile Online bereitgestellt.
- ContentManager – ist eine Kombination aus EssenceManager und Datenbank zur Beschreibung des Objekts auf Metaebene. Dies umfasst generierbare Informationen, wie zum Beispiel Übertragungsrate, Auflösung, und Anzahl der Audiokanäle, wie auch Metainformationen, wie Aufnahmeort, Aufnahmezeitpunkt, Generation des Ausgangsmaterials, und Einsteller des Materials.
- SearchEngine – Verwaltet das gesamte textuelle Material und erlaubt die Suche nach bestimmten Einträgen.
- AV-Network – Repräsentiert als Komponente das Video- und Audionetzwerk und erlaubt das automatisierte Schalten von Verbindungen zwischen zwei beliebigen Punkten in diesem Netz.

Und aus den folgenden Arbeitsschritten:

- Annotation – Ist ein Arbeitsschritt, bei dem die auf dem Material sichtbaren Bilder textuell beschrieben werden. Als Voraussetzung muss zumindest ein Teil des zu annotierenden Materials bereits im System vorhanden sein (also im EssenceManager). Als Ausgabe liefert dieser Schritt für die Suchmaschinen verwertbare Eingaben in Form von Textdokumenten.
- VideoAnalyse – Hierbei wird das Material automatisch wieder in die Schnittfolge zerlegt, in der es ursprünglich montiert wurde. Als Eingabe benötigt dieser Schritt Zugriff auf das Material. Als Ergebnis liefert es neues Material (Bilder, die wichtige Bildinhalte beschreiben) das wiederum abgelegt werden muss, sowie textuelle Informationen über die gefundenen Schnitte, die ebenfalls abgelegt werden müssen.
- Cataloging – Erlaubt dem Dokumentar die Manipulation der Schnitte, sowie die nachträgliche Korrektur der bei der Annotation eingegebenen Daten. Die Ausgaben ersetzen die Originaldaten und werden wieder an den gleichen Stellen im System abgelegt.
- Kostenstellen – Berechnen den Nutzern Gebühren für die Inanspruchnahme der durch sie repräsentierten Dienste.

Die **Aufgabenstellung** sei das Einstellen eines neuen Eintrags im System mit anschließender Annotation und Zerlegung für den Zugriff durch die Redakteure. Cataloging soll nur bei Bedarf durchgeführt werden.

Der **Nutzer** des Systems erwartet den folgenden Ablauf: Zu Beginn läuft das Rohmaterial über einen Feed (Video- und Audio Standleitung) in der Abteilung ein. An diesem Verbindungspunkt zur Außenwelt wird bereits entschieden, ob das Material später über den Computer erschließbar sein soll. Ist dies nicht der Fall, wird der Feed nur auf einem hochauflösenden Videorekorder für die spätere Verwendung aufgezeichnet. Soll das Material hingegen elektronisch recherchierbar werden, wird parallel zur ebenfalls laufenden HiRes Kopie eine Lowres-(Edit-) Kopie in einem computerlesbaren Format erzeugt und im EssenceManager abgelegt. Falls computerlesbares Material vorliegt, ist der nächste Schritt die inhaltliche Erschließung. Dazu wird das Material durch einen Menschen gesichtet und annotiert, was bedeutet, dass die jeweils sichtbaren Bildinhalte, Personen und Abläufe textuell beschrieben werden. Dazu nutzt der Anwender Strata, die unabhängig voneinander den gleichen zeitlichen Bereich beschreiben. Die Ergebnisse dieser Inhaltlichen Erschließung werden verbunden mit den Ergebnissen der automatisierten Erschließung des Materials auf technischer Ebene. Dabei wurde das Material durch die Videoanalyse in Schnitte unterteilt und klassifiziert. Beide Resultate werden durch einen Menschen kontrolliert und gegebenenfalls noch einmal korrigiert. Dies ist Notwendig, da speziell eine automatisierte Schnitterkennung, abhängig vom Ausgangsmaterial, nicht immer korrekte Ergebnisse liefert. Danach werden die gefundenen Daten in die Suchmaschine eingebracht, wodurch das Objekt für andere Nutzer sichtbar wird. Eventuell wurde dieser Schritt bereits zu einem früheren Zeitpunkt ausgeführt um aktuelles Material um den Preis einer unvollständigen Erschließung frühestmöglich zugreifbar zu machen. Das beendet die für den Nutzer sichtbaren Teile eines möglichen Workflows.

Die für das **Material** relevanten Teile dieses Workflows sehen so aus: Vor dem Beginn der Übertragung muss der Feed bereits reserviert und geschaltet werden. Dazu müssen im AV Netzwerk die korrekten Verbindungen aktiviert werden und diese Konfiguration für den gewünschten Zeitraum reserviert werden. Im weiteren müssen Reservierungen für den Videorekorder und Encoder für die Lowres-(Edit-)Kopie geschaltet werden. Reservierungen haben teilweise Unter-Reservierungen zur Folge, in deren Verlauf Speicherplatz im EssenceManager und Einträge in den Materialdatenbanken reserviert werden. Während des Einstellvorgangs werden die durch die Reservierung belegten Geräte genutzt. Nachdem das Einstellen beendet wurde, werden die Ressourcen für die Geräte und das AV Netzwerk freigegeben. Dafür müssen nun die Reservierungen im Materialsystem verlängert werden, damit die bis-

her temporär belegten Ressourcen dauerhaft zugeordnet bleiben. Für die Inhaltliche Erschließung durch einen Menschen müssen zum Zeitpunkt des Start der Bearbeitung, Reservierungen im Materialsystem und AV Netzwerk für die Übertragung der Daten zwischen EssenceManager und dem lokalen System beim Nutzer reserviert und geschaltet werden. Nachdem alle Ergebnisse vorliegen, wird, abhängig Ergebnis des Cataloging, der Speicher im EssenceManager weiterhin (nun dauerhaft) belegt oder es wird das Material gelöscht und die bisher belegten Ressourcen wieder freigegeben. Damit sind alle das Material betreffenden Teile des Workflows abgeschlossen. Nicht näher betrachtet wurden die Fälle, bei denen das System die bereits aufgezeichneten Daten wieder löschen muss. Dies ist Beispielsweise dann der Fall, wenn auf dem Feed das Falsche gesendet wurde, oder die Ergebnisse der Extraktion und Annotation nicht den Wünschen der Anwender entsprochen hatte.

Schließlich das **Abrechnungssystem**: Schon vor Beginn der eigentlichen Übertragung werden Gebühren für die gewünschte Materialquelle, die gewünschten Zielformate, sowie Speicherdauer – unabhängig vom physischen Transport des Materials – fällig. Dazu kommen Abgaben für die benutzten Geräte, beispielsweise Gebühren für Videorekorder, belegte Videoleitungen, eventuell ein Encoder für das Lowres Material. Der Onlinespeicher muss für die gewünschte Dauer ebenfalls reserviert – und damit bezahlt – werden. Sollte die gemietete Zeit ohne Kontraktverlängerung ablaufen, müssen durch das Abrechnungssystem Mahnungen verschickt werden; eventuell wird das Material zwangsweise freigegeben. Nach dem Einstellen des Materials wird zu dessen Inhaltlicher Erschließung menschliche Hilfe in Form eines Dokumentars benötigt. Dazu muss die Ressource Mensch für diese Aufgabe gebucht werden, wobei sowohl Startzeitpunkt und Dauer dem System im voraus nicht bekannt sind. Zusätzlich muss die (verbrauchte/bereitgestellte) Bandbreite im Netzwerk bezahlt werden, die während der Bearbeitung des Materials benötigt wurde. Die genaue Menge lässt sich allerdings erst nach Beendigung der Arbeit feststellen, was die Vorgabe einer oberen Grenze der verursachten Kosten erschwert. Schließlich muss für die dauerhafte Einstellung des Materials im EssenceManager bzw. für den Stellplatz der Videokassette im Archiv ein Betrag an den Dienstanbieter gezahlt werden.

Die unterschiedlichen Sichten vermitteln jede für sich immer nur ein unvollständiges Bild des Ablaufs, die für diese Aufgabe notwendig sind. Es fällt auf, dass sich jeder Ablauf durch mehrere einfache Schritte beschreiben lässt. Dabei wiederholen sich diese Schritte, oder treten leicht abgewandelt mit anderen Parametern erneut auf. Oft lassen sich die Teilschritte der verschiedenen Sichtweisen **nicht** zeitlich aufeinander abstimmen. Manche überlappen sie sich, auch ist Beispielsweise aus Nutzersicht der Workflow beendet, während er für das Abrechnungssystem noch läuft. Und schließlich ist die Dauer eines einzelnen Schrittes nicht immer deterministisch. Das gilt unter anderem für den Zeitraum nach der physischen Erfassung bis zum Start der Annotation: Hier müssen die Reservierungen weiterhin belegt und ggf. verlängert werden, während weder der Nutzer noch das Material aktiv sind. Daher erscheint der Versuch wenig sinnvoll, einen globalen Workflow „über" diese Sichten zu legen. Zu schnell wären die sich aus der statischen Modellierung ergebenden Abhängigkeiten, nicht mehr handhabbar. Vielmehr muss der Plan während der Abarbeitung der Teilziele an die resultierenden Gegebenheiten angepasst werden.

Betrachten wir den Workflow aus einem anderen Blickwinkel: Zwischen Nutzer und Ressource kommt man auf diese Beziehungen:

1. Es müssen Ressourcen reserviert und diese Reservierungen gegebenenfalls verlängert oder vorzeitig freigegeben werden.
2. Teilweise muss auf das Ergebnis einer Aktion gewartet werden, bevor der nächste Schritt gemacht werden kann.
3. Nicht immer ist die Dauer und Umfang der Nutzung einer Ressource vorher bekannt, was häufig während der Interaktion mit menschlichen Nutzern vorkommt.
4. Der nächste Schritt ist unter Umständen abhängig vom Ergebnis einer menschlichen Aktion, deren Anfang und Dauer nicht durch das System vorhergesagt werden kann.
5. Der nächste Schritt ist nicht immer im voraus planbar, da auch unerwartete Ereignisse eintreten können (während der Aufnahme bricht der Feed ab. Sollen die bereits aufgenommenen Daten verworfen werden oder soll der Feed trotzdem weiter verwendet werden?). Dadurch wird eine instanz-basierte Ressourcenplanung erschwert.

Obwohl dies mit regelbasiertem Schließen lösbar ist, bieten sich hierfür aber Agenten gleich aus mehreren Gründen an:

1. Durch ihre „Intelligenz" sind sie die natürliche Antwort auf „Wünsche" des Benutzers.
2. Autonomes Handeln sichert das Lösungsbestreben auch ohne die direkte Überwachung durch eine übergeordnete Instanz.
3. Eine gemeinsame Sprache sichert die Zusammenarbeit von Agenten, selbst wenn konkrete Instanzen zu Beginn der Implementierung des Systems nicht existierten.
4. Durch die Abstraktion von Implementierung (schon gegeben durch die Objektorientierte Programmierung), sowie vom konkreten Verhalten (gegeben durch die Nutzung einer gemeinsamen Sprache, an Stelle eines gemeinsamen Interfaces) Gewinnung eines zusätzlichen Freiheitsgrades.

Es gibt bereits umfangreiche Forschungen auf dem Gebiet der Workflowunterstützung, wobei auch schon der Übergang auf einer verteilte Ausführung [74] mit Hilfe von Agenten vollzogen worden ist [75], wobei in diesem Zusammenhang ein Agent meist eine (reale) Person bezeichnet, die für einen Arbeitsschritt verantwortlich ist, nicht aber eine Komponente der Workflowunterstützung selbst.

# 5. Fazit: Sind Agenten überhaupt sinnvoll?

Dieses Kapitel befasst sich bewusst stark polarisierend mit den aus den vorangegangenen Kapiteln beschriebenen Unzulänglichkeiten und Eigenschaften von Agenten und den sie beherbergenden Systemen. Hierdurch soll der Anlass gegeben werden, mögliche, bereits gefassten Meinungen zu reflektieren.

Die erste These wendet sich gegen den Versuch, durch Nutzung von (Agenten-)Sprachen die Vielfalt bekannter Remote-Zugriffsmöglichkeiten zu verringern. Denn dieser Ansatz, eine neue Sprache zu definieren, ist der typische Versuch, Probleme durch das Einfügen einer Zwischenschicht zu lösen. Man behandelt die Symptome, und nicht die Ursache. Es wird suggeriert, dass allein durch das Verstehen der Sprache eines Anderen automatisch dessen Funktionalität erschlossen werden kann. Nur kann man eine Sprache nicht „einfach erweitern" und dann ernsthaft glauben, dass damit neue Funktionalität „einfach so" und „automatisch" durch andere genutzt werden wird. Bestehende Komponenten müssen erweitert werden, wenn sie von den Vorteilen der erweiterten Kommunikationsprimitive profitieren wollen und diejenigen, die mit anderen kommunizieren möchten, müssen immer damit rechnen, dass sie auf jemandem treffen, der den neuesten „Dialekt" der Sprache nicht implementiert. Für den Fall muss der Fragende immer auch die alten Kommunikationsprimitive nutzen können, wodurch eine wirkliche Evolution ausbleiben wird. Die Folge: Entweder, man einigt mich eben doch auf einen gemeinsamen Sprachkern, den man nur noch erweitern, aber nicht mehr umdefinieren darf. Oder, Entwickler nachfolgender Systeme haben wieder die gleichen Probleme, die er auch schon bei der Evolution von Interfaces hatten: Bei Interfaces muss jeder Nutzer n Varianten seines Ursprunginterfaces verstehen und jeder Anbieter bedienen (bsp. In Form von ItheInterface, ItheInterface2, etc.). Ein Nutzer muss jedoch bei Dialekten zusätzlich für jede unterstützte Sprache (und genau da liegt die Komplexität) Regeln besitzen, die eine Anfrage erst erlauben: es ist nicht damit getan, „nur" die richtigen Vokabeln zu benutzen, vielmehr muss der Sprecher unter Umständen mehrere Sätze in der eine Sprache sprechen, um das gleiche zu erreichen, das er in einer anderen Sprache mit einem Wort ausdrücken kann. Die Erweiterung einer Sprache (im Sinne vom Computersprache, Interface) führt oftmals Konstrukte ein, die aus sprachlicher Sicht „ein Wort" sind und Befehlsfolgen einer früheren Revision ablösen. Damit kann natürlich ein alter Sprecher mit neuen Zuhörern sprechen (weil diese ja aus Kompatibilität die alten Sätze weiter verstehen müssen, aber es muss auch jeder Sprecher alle bisherigen Sprachen weiterhin sprechen können, will er sich in einem bestehenden, gewachsenem System nicht den Zugang zu älteren Komponenten verbauen.

Zusammengefasst, kommt man auf Probleme, die es auch bei realen Sprachen gibt:

1. Eine Sprache ist nicht „statisch", sondern verändert sich im Laufe ihrer Nutzungszeit.
2. Eine Sprache ist nicht „eindeutig". Viele Konstrukte haben eine Kontextabhängige Bedeutung.
3. Selbst innerhalb einer Sprache gibt es parallel „Dialekte", die oft nur durch Deuten und nicht durch Regeln auf den gemeinsamen Sprachstamm abgebildet werden können.

Es gibt viele Versuche, eine gemeinsame Sprache zu finden. Aus Neid, Profit, politischen Erwägungen, falsch verstandenen Geltungsdrang und vorgeschobenen Gründen wird dies jedoch – obwohl bestritten – immer wieder torpediert. Warum sollte das anders werden, nur weil es jetzt eine Metasprache für die Sprache selbst gibt? Wird es nicht vielmehr eine Menge von Metasprachen geben, die alle um die Gunst des Anwenders konkurrieren?
**These: Menschen wollen keine gemeinsame Sprache! Und da Menschen Agenten entwerfen, werden Agenten keine gemeinsame Sprache sprechen.**

Ähnlich sieht es mit der Sicherheit in einem Agentensystem aus. Sichere Systeme kann es nicht geben, einfach weil ein angreifendes System die gerade auf ihm laufenden Prozesse immer durch einfaches Lesen der Daten im Speicher ausspionieren, und damit im nächsten Schritt auch manipulieren, kann.
**These: Agenten werden niemals sicher (voreinander, bzw. vor einem angreifenden Agentensystem) sein.**

Auch wird immer wieder vorgehalten, dass die Performance bestehender Systeme durch die Migration der aktiven Komponenten verbessert werden kann. Doch bleiben die Autoren den Beweis meist schuldig oder schränken die Randbedingungen unzulässigerweise ein. Beispielsweise wird als lohnender Anwendungsfall gerne das Netzwerkmanagement angeführt und dessen hohen Latenzzeiten bei der zentralen Abfrage des Systems angeführt um den Einsatz mobiler Agenten zu rechtfertigen. Aber genau hier ist der Gewinn durch die Migration minimal, legt man den generischen Weg der Migration zu Grunde – ohne Caching und ohne Pre-Migration des Codes auf die Zielsysteme. So groß sind die Zeit- und Kapazitätsverluste durch die Migration des Codes, sowie die Verluste durch Verpackung und Neustart des Agenten, dass sie die vermeintlichen Performancegewinne durch die große zu übertragende Datenmenge negieren. Vergleichbares gilt – ohne Optimierungen – für die Verminderung der Latenzzeiten, da an Stelle der vielen kleinen Abfragen nun ein relativ großer Codeblock übertragen werden muss oder auf jedem Knoten ein eigenständiger Agent instanziiert werden muss. Auch der Ansatz, das Routing durch Agenten zu vereinfachen, ist nur auf den ersten Blick naheliegend. Wenn man hier annimmt, dass letztendlich die Konfigurierung bestehender Netze durch diese Technik vereinfacht werden soll, bedeutet dies, dass jeder Knoten jedem einkommenden Agenten trauen müsste. Das wird kein Systemadministrator zulassen (wollen). Und schließlich verhindern grundlegende Eigenschaften den wirklich verbreiteten Einsatz mobiler Agenten. In einem System wie ebay [76] wird eine Auktionsplattform geboten, auf der potentielle Käufer miteinander um ein Produkt konkurrieren. Hier wäre es sinnvoll, wenn das System Agenten von Benutzern akzeptieren würde, die nach eigenen Regeln bei einzelnen Auktionen mitbieten um einen für den Bieter akzeptablen Preis zu erzielen. Würde nun ebay seinen Kunden erlauben, ihre Agenten auf die eigene Plattform zu migrieren, käme es (vorsichtig geschätzt) zu mehreren 1000 Agenten, die gleichzeitig aktiv wären und daher auch Ressourcen (Thread, Speicher) benötigen. Mit heutigen Systemen wäre dies nicht realisierbar und selbst auf zukünftigen Systemen sei dahingestellt, ob sie 10000 parallel laufende Threads unterstützen können. Gleiches gilt für den durch diese Anzahl Prozesse belegten Speicher. Hier soll nicht in Abrede gestellt werden, dass man Agenten nutzen könnte, um die in der jeweiligen Fragestellung definierten Ziele zu erreichen. Was allerdings in Abrede gestellt wird, ist, dass dies immer sinnvoll im Hinblick auf die durch die Autoren definierten Ziele ist.
**These: Agenten können bestehende Probleme nicht „besser" lösen, speziell bringt allein die Nutzung von Agententechnik keinen Performancegewinn!**

**Ein Irrglaube: „Es gibt bereits Killerapplikationen für mobile Agenten“**. Es wird immer wieder behauptet, dass sich auch für die allgemeine Anwendung von mobilen Agenten eine Must-Have Applikation finden lässt, welche erst durch die vielen Vorteile mobiler Agenten dem Anwender Nutzen bringen würde und die ohne mobile Agenten entweder gar nicht, oder zumindest nicht so effektiv hätte implementiert werden können. Bisherige Applikationen scheitern an den nicht allgemein akzeptierten und meist propietären Schnittstellen zwischen den Komponenten. Vorschläge, wie DAML und OIL versuchen diese Punkte zu entschäfen, indem sie in keiner direkten Relation stehenden Subjekten die semantische Erschließung der Sprache des jeweils anderen ermöglichen. Durch diesen Schritt wird zwar keine direkte Erweiterung der Funktionalität des Einzelnen erreicht, jedoch kann so eine Komponente die gewünschten Eigenschaften einer anderen Komponente lokalisieren, selbst wenn sie nicht über den gleichen „Zugriffspfad“ erreichbar sind [6] [15]. Möglicherweise erlauben Agenten die Lösung einer neuen Klasse von Problemen, die auf der lokalen Optimierung, basierend auf verteiltem Wissen, in einem dynamischen Umfeld basieren. Beispielsweise ist das Routing in Ad-Hoc Netzwerken ähnlich dem Aushandeln von Terminen. Im allgemeinen scheint es immer dann sinnvoll, mobile Agenten einzusetzen, wenn das Bilden globaler Sichten „teuer“ ist. Dieser Ansatz scheint logisch, doch enthält er einen Fehler: Was hier mit Agenten bezeichnet wird, sind „nur“ verteilte Algorithmen, die verteiltes Wissen sammeln und dieses Wissen an die Nachbarn weitergeben. Beim Routing kommt es darauf an, die eigenen Nachbarn zu finden und dieses Wissen um die Nachbarn im restlichen System zu verbreiten. Da die einzelnen Knoten kein eigenes Interesse an diesem Wissen haben (anders, als die möglicherweise darauf zurückgreifenden Komponenten!), sind sie typischerweise keine Agenten. Vielversprechender ist da schon das Aushandeln von Terminen. Hierbei haben die Teilnehmer in der Regel nur lokales Wissen und müssen sich in (möglicherweise) mehreren Runden auf einen gemeinsamen Termin einigen. Hierbei sind auch kommunikative Komponenten gefragt, da die Terminplaner zum Beispiel Menschen bzw. Softwarekomponenten verschiedener – untereinander konkurrierender – Hersteller sein können. Auch in Netzen, die historisch gewachsen, deren Protokolle inkompatibel, nicht alle Varianten bekannt sind, und die dennoch gemeinsam genutzt werden, kommt es zu einer Situation, die ähnlich dem Abgleich einer Besprechung ist: Knoten müssen zusammengeschaltet werden, Verbindungsparameter über bislang unbekannte, möglicherweise ungeeignete, Verbindungen garantiert werden und Vereinbarungen ständig auf Einhaltung überwacht werden. Das Wissen hierzu liegt verteilt auf den Knoten. Der Agent selbst ist nicht mehr Teil des Netzes, sondern Beauftragter eines Nutzers der ein eigenes, möglicherweise anderen Agenten zuwiderlaufendes, Ziel verfolgt.

Zusammenfassend lassen sich folgende Voraussetzungen für die erfolgreiche Nutzung von Agententechnik aufstellen:

1. Das Wissen ist verteilt.
2. Der Fragende verfolgt ein persönliches Ziel.
3. Das zu nutzende Wissen ist „per Definition“ nicht generalisiert zugänglich.

Selbstorganisierende, lokale Systeme profitieren im besonderen von Agenten. Dies jedoch nicht aus Performanceerwägungen oder weil dies eine effektivere Implementierung eines bestehenden Konzepts ist, sondern weil sie vollständig neue Ansätze für die Programmierung verteilter Systeme darstellen. Kein bisheriger Ansatz hat ernsthaft Teile seiner Macht an ein Computerprogramm abgeben von dem nicht garantiert werden kann – ja selbst nicht einmal sicher wird – dass es seine Aufgabe „besser“ macht, als die bisherige Lösung. Erst, wenn dieses Vertrauen geschaffen ist, kann es „echte“ Lösungen geben.

Dies führt zum Abschluss zu einer bisherigen Überlegungen nicht unbedingt zu erwartenden These:

**Es gibt sinnvolle Anwendungen, die nicht ohne Agenten realisierbar sind.**

# 6. Zusammenfassung

In der vorangegangenen Kapiteln wurde ein Überblick über das Gebiet der mobilen Agenten gegeben. Dabei stellte sich heraus, dass es zwei Ebenen bei der Betrachtung von Agenten gibt. Zum einen aus Sicht der KI, zum anderen aus Sicht der Verteilten Betriebssysteme. Die KI beschäftigt sich mit der Kommunikation und Interaktion von Agenten miteinander, während man sich bei den verteilten Systemen auf die Bereitstellung der zur Kommunikation notwendigen Infrastruktur und der Kommunikation der Agenten mit dem System beschränkt. Im weiteren wurden Aspekte der Infrastruktur vorgestellt und deren spezifischen Probleme diskutiert. Speziell wurden hierbei die Migration von Ressourcen und der Zugriff darauf, sowie das Suchen (und Finden) von Ressourcen in einem verteilten, möglicherweise mobilen System, vorgestellt. Dabei stellte sich heraus, dass bereits bestehende Lösungen zur Standortbestimmung den Anforderungen von mobilen Objekten nicht genügen. In weiteren Arbeiten sollen Lösungen für die Lokationsbestimmung, sowie Ressourcenverwaltung erarbeitet werden, wobei nicht die Schaffung eines (weiteren) Agentensystems angestrebt wird, sondern die Schaffung orthogonaler Pakete, die keine besondere Umgebung voraussetzen.

# 7. Referenzen


[1] A. S. Rao and M. P. Georgeff, "BDI Agents: From Theory to Practice", 1995. "http://citeseer.nj.nec.com/rao95bdi.html", last visited June, 2001

[2] L. N. Foner, "What's an agent anyway? - a sociological case study", MIT Media Lab, ftp report Agents Memo 93-01, 1993.

[3] C. E. Hewitt, "Viewing control structures as patterns of passing messages", *Journal of artificial intelligence*, vol.8(3), pp.323-364, 1977.

[4] H. S. Nwana, "Software agents: an overview", *Knowledge engineering review*, vol.11(3), pp.205-244, 1996.

[5] T. Finin, J. Weber, G. Wiederhold, and M. Genesereth, "Specification of the KQML Agent-Communication Language": DARPA Knowledge Sharing Initiative External Interfaces Working Group, 1993. "http://www.cs.umbc.edu/kqml/papers/", last visited June, 2001

[6] DAML, "The DARPA Agent Markup Language (DAML) Homepage": DARPA, 2000. "http://www.daml.org/", last visited June, 2001

[7] A. Wood, "Towards a Medium for Agent-Based Interaction", in *School of Computer Science*. Birmingham: University of Birmingham, 1994, pp. 29.

[8] H. Reiser and G. Vogt, "Security Requirements for Management Systems using Mobile Agents", presented at Proceedings of the Fifth IEEE Symposium on Computers and Communications: ISCC 2000, Antibes, France, 2000.

[9] AMETAS, "AMETAS, System mobiler, autonomer Softwareagenten": Universität Frankfurt, 2000. "http://www.vsb.cs.uni-frankfurt.de/ametas/", last visited March, 2001

[10] Tracy, "Tracy": Universität Jena, 2000. "http://tracy.informatik.uni-jena.de/", last visited March, 2001

[11] Mole, "Mole": Universität Stuttgart, 2000. "http://mole.informatik.uni-stuttgart.de/", last visited March, 2001

[12] M. Wooldridge and N. R. Jennings, "Agent theories, architectures, and languages: A survey",, vol. 890, M. Wooldridge, Ed., Lecture Notes in Computer Science ed. Berlin: Springer Verlag, 1995.

[13] M. R. Genesereth and S. P. Ketchpel, "Software agents", presented at Communications of the ACM, 1994.

[14] C. Nicholas, "KQML as an agent communication language", presented at Conference on Information and Knowledge Management (CIKM), 1994.

[15] F. v. Harmelen and I. Horrocks, "Reference description of the DAML+OIL ontology markup language", 2001. "http://pride.daml.org/2000/12/reference.html", last visited June, 2001
[16] J. Bradshaw, "Introduction to software agents", in *Introduction to software agents*, J. Bradshaw, Ed. Menlo Park, CA: The MIT Press, 1996, pp. 3-46.
[17] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice", *The Knowledge Engineering Review*, vol.10(2), pp.115-152, 1995.
[18] C. F. Steketee, "Process Migration and Load Balancing in Amoeba", presented at Australasian Computer Science Conference, 1999.
[19] Sun, "Sun RPC (RFC1057)": Sun Microsystems, 1988. "http://www.ietf.org/rfc/rfc1057.txt", last visited June, 2001
[20] Sun, "JavaTM Remote Method Invocation (RMI)": Sun Microsystems, 1995. "http://java.sun.com/products/jdk/rmi/", last visited June, 2001
[21] OMG, "CORBA Specification": OMG, 1998. "http://www.omg.org/technology/documents/formal/corbaiiop.htm", last visited June, 2001
[22] HP, "PCL Specification", 1990. "http://www.sxlist.com/techref/language/pcl/index.htm, http://www.hp-developer-solutions.com", last visited June, 2001
[23] Adobe, "Postscript Language Reference": Adobe, 1985. "http://www.adobe.com/products/postscript/pdfs/PLRM.pdf", last visited June, 2001
[24] J. Gosling, F. Yellin, and T. J. Team, "JavaTM API Documentation Version 1.0.2": Sun Microsystems, 1996. "http://java.sun.com/products/jdk/1.0.2/api/", last visited June, 2001
[25] U. M. Borghoff and K. Nast-Kolb, "Distributed Systems: A Comprehensive Survey", Institut für Informatik, Technische Universität München, München TUM-I8909, November 1989 1989.
[26] R. L. Rivest, A. Shamir, and L. M. Adelman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", MIT Laboratory for Computer Science, Technical Memo MIT/LCS/TM-82, April 1977.
[27] A. S. Tanenbaum, *Verteilte Betriebssysteme*, 1995.
[28] Nehmer and H. Assenmacher, "Panda/Rapid": AG Nehmer, 1993. "http://wwwagss.informatik.uni-kl.de/Projekte/Panda/", last visited June, 2001
[29] P. Homburg, "The Architecture of a Worldwide Distributed System", in *Advanced School for Computing and Imaging*. Amsterdam: Vrije Universiteit, 2000, pp. 388.
[30] Sun, "JiniTM network technology", 1999. "http://www.sun.com/jini/", last visited June, 2001
[31] A. S. Tanenbaum, M. v. Steen, and P. Homburg, "The Architectural Design of Globe: A Wide-Area Distributed System", Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam 1997.
[32] OMG, "CORBA Forwarding Specification. General Inter-ORBProtocol, Section 15. CORBA V2.3.1": OMG, 1999. "http://cgi.omg.org/cgi-bin/doc?formal/98-02-64.pdf", last visited March, 2001
[33] J. G. Sempere, "An overview of the GSM system": Communications Division, Department of Electronic&Electrical Engineering, University of Strathclyde, 1997. "http://www.comms.eee.strath.ac.uk/~gozalvez/gsm/gsm.html", last visited June, 2001
[34] B. Gray, "Soldiers, Agents and Wireless Networks: A Report on a Military Application", presented at PAAM2000, Manchester, England, 2000.
[35] M. D. Association, "Mobile Data Association", 1994. "http://www.mda-mobiledata.org/", last visited June, 2001
[36] A. K. Salkintzis, "A Survey of Mobile Data Networks", IEEE Communication Surveys, The University of British Columbia 1999.
[37] Sun, "JavaCard - Java Smartcard Technology", 2001. "http://java.sun.com/products/javacard/", last visited June, 2001
[38] H.-W. Kisker, "Sicherer Zugang zu Daten und Informationen durch Smart-Karte und elektronisches Buch", *Zentrum für Informationsverarbeitung der Universität Münster*, vol.24(3), 2000.

[39] T. Imielinsky and B. R. Badrinath, "Querying in highly mobile distributed environments", presented at Proceedings of the 18th Very Large Databases (VLDB) Conference, Vancouver, British Columbia, Canada, 1992.
[40] L. Cluster, "MOSIX", 2001. "http://www.mosix.org/", last visited July, 2001
[41] ACCC, "Advanced Cluster Computing Consortium", 2001. "http://www.tc.cornell.edu/AC3/Memberships/", last visited July, 2001
[42] Sun, "XDR Data Representation (RFC1832)": Sun, 1995. "http://www.ietf.org/rfc/rfc1832.txt", last visited June, 2001
[43] W3C, "HTTP Version 1.1": W3C, 1999. "ftp://ftp.isi.edu/in-notes/rfc2616.txt", last visited June, 2001
[44] IETF, "Official Internet Protocol Standards": IETF, 2001. "http://www.rfc-editor.org/rfcxx00.html", last visited June, 2001
[45] J. B. Postel, "Simple Mail Transfer Protocol": IETF, 1982. "ftp://ftp.isi.edu/in-notes/rfc821.txt", last visited June, 2001
[46] B. Kantor, "Network News Transfer Protocol": IETF, 1986. "ftp://ftp.isi.edu/in-notes/rfc977.txt", last visited June, 2001
[47] IBM, "IBM Aglets Workbench": IBM, 2000. "http://www.aglets.org", last visited March, 2001
[48] M. Electric, "CONCORDIA": Mitsubishi Electric Information Technology Center America (ITA), 1997. "http://www.concordiaagents.com/", last visited June, 2001
[49] Grasshopper, "Grasshopper Mobile Agents": Grasshopper, 1998. "http://www.grasshopper.de", last visited March, 2001
[50] L. M. Silva, L. Almeida, P. Simones, G. Soares, P. Martins, V. Batista, C. Renato, and N. Stohr, "JAMES: A Plattform of Mobile Agents for the Management of Telecommunication Networks", presented at PAAM, Coimbra, 1999.
[51] Ara, "Ara": Universität Kaiserslautern, 2000. "http://wwwagss.informatik.uni-kl.de/Projekte/Ara/index.html", last visited March, 2001
[52] T. Sakamoto, T. Sekiguchi, and A. Yonezawa, "Bytecode Transformation for Portable Thread Migration in Java", Department of Information Science, Facullty of Science, University of Tokyo, Tokkyo 2000.
[53] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable Support for Transparent Thread Migration in Java", Department Computerwetenschappen, K.U. Leuven, Heverlee 2000.
[54] L. M. Silva and L. Almeida, "The Advantages of Using Mobile Agents in Software for Telecommunications", presented at iccc99, Coimbra, Portugal, 1999.
[55] F. Hohl, "A Protocol to Detect Malicious Hosts Attacks by Using Reference States", Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR), Fakultät Informatik, Universität Stuttgart, Stuttgart Bericht Nr. 1999/09, 1999.
[56] T. Dierks and C. Allen, "The TLS Protocol": Certicom, 1999. "ftp://ftp.isi.edu/in-notes/rfc2246.txt", last visited June, 2001
[57] E. Bonabeau, F. Henaux, S. Guérin, D. Snyers, P. Kuntz, and G. Theraulaz, "Routing in telecommunications networks with "smart" ant-like agents",, 1998.
[58] N. Minar, K. H. Kramer, and P. Maes, "Cooperating Mobile Agents for Dynamic Network Routing", MIT Media Lab, Cambridge MA 1999.
[59] G. D. Caro and M. Dorigo, "Mobile Agents for Adaptive Routing", IRIDIA, Universitè Libre de Bruxelles, Bruxelles 1997.
[60] R. Schoonderwoerd, O. Holland, J. Bruten, and L. Rothkrantz, "Ant-based load balancing in telecommunicataions networks", HP Laboratories Bristol, Delft University of Technology, The Nederlands, Delft 1996.
[61] D. B. Terry, "Distributed Name Serves: Naming and Caching in Large Distributed Computing Environments", Computer Science Division Univ. of Berkeley, Berkeley ICB/CSD 85/228, March 1985.
[62] B. Wellington, "Secure Domain Name System (DNS) Dynamic Update": IETF, 2000. "http://www.ietf.org/rfc/rfc3007.txt", last visited June, 2001
[63] V. Roth, "Scalable and Secure Global Name Services for Mobile Agents", presented at 6th ECOOP workshop on mobile object systems, Sophia Antipolis, France, 2000.

[64] M. v. Steen, A. S. Tanenbaum, A. Baggio, and G. Ballintijn, "Efficient Tracking of Mobile Objects in Globe", Vrije Universiteit, Amsterdam IR-481, November 2000.
[65] A. Gulbrandsen and P. Vixie, "A DNS RR for specifying the location of services (DNS SRV)", 1996. "ftp://ftp.isi.edu/in-notes/rfc2052.txt", last visited June, 2001
[66] Seti, "Seti@Home", 2001. "http://www.seti.org/", last visited June, 2001
[67] D. Net, "Distributed Net - RC5 Contest", 1999. "http://www.distributed.net/rc5/", last visited June, 2001
[68] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, "Experience with Software Watermarking", Purdue University, West Lafayette 2000.
[69] F. Hohl, "An approach to solve the problem of malicious hosts", Fakultät Informatik, University of Stuttgart, Germany, Stuttgart Bericht Nr. 1997/03, 1997.
[70] Axelrod, *The Evolution of Cooperation*. New York: Basic Books, 1984.
[71] J. Petry, "Milos Homepage": AG Richter, 2000. "http://wwwagr.informatik.uni-kl.de/~milos/index.htm", last visited September, 2001
[72] P. Thomas, "MediaArchive Whitepaper": tecmath AG, 1999. "http://www.media-archive.de", last visited June, 2001
[73] D. Slack, "ContentShare Introduction": GrassValley Group, 1999. "http://www.contentshare.com/", last visited June, 2001
[74] S. Goldmann, J. Münch, and H. Holz, "Distributed Process Planning Support with MILOS", *Int. Journal of Software Engineering and Knowledge Engineering*, pp.12, 2000.
[75] S. Goldmann, J. Münch, and H. Holz, "MILOS: A Model of Interleaved Planning, Scheduling, and Enactment", presented at ICSE 99 Workshop on Software Engineering over the Internet, 1999.
[76] Ebay, "Ebay Auction Platform", 1996. "http://www.ebay.com", last visited June, 2001