INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Entwurf eines Java/CORBA-basierten Mobilen Agenten

Bearbeiter:	Bernhard Kempter
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Boris Gruschke Alexander Keller

INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Entwurf eines Java/CORBA-basierten Mobilen Agenten

Bearbeiter: Bernhard Kempter

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Boris Gruschke
Alexander Keller

Abgabetermin: 15. August 1998

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. August 1998

....
(Unterschrift des Kandidaten)

Zusammenfassung

Durch die schnell wachsende Größe und Komplexität von bestehenden verteilten Systemen, ist das zentrale Management, wie es heute noch sehr weit verbreitet ist, aus vielfältigen Gründen inadäquat. Ein verteiltes, flexibles Management ist erforderlich, um den Anforderungen gerecht zu werden. Beim verteilten Management kommen Agenten zum Einsatz, die jeweils eine bestimmte Aufgabe zu bewältigen haben. Ein Agent ist ein autonom handelndes, kommunikationsfähiges Softwaresystem.

In dieser Arbeit ist eine Agentenarchitektur entworfen und implementiert worden, die den Ansprüchen des verteilten Managements entspricht. Die wichtigsten Eigenschaften der Agenten sind Mobilität und Kommunikationsfähigkeit mit anderen Agenten. Jeder Agent bietet darüberhinaus eine graphische Bedienoberfläche an. Die Agenten werden auf Agentensystemen, ihrer Laufzeitumgebung, ausgeführt, die das Management der Agenten übernehmen.

Als Kommunikationsinfrastruktur für die Agenten wird CORBA verwendet. Agenten, Agentensystem und Bedienoberfläche sind in Java implementiert. Ein Webbrowser übernimmt die Visualisierung der graphischen Bedienoberflächen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Durch die immer größer werdenden Rechnernetze, die sich durch ihre Heterogenität in Hardware und Software charakterisieren lassen, wird das systematische Management dieser Rechnernetze immer schwieriger – aber auch wichtiger.

Dieses Kapitel gliedert sich in folgende Abschnitte:
Der Bedarf für *Mobile Agenten* im Management wird anhand eines Beispielszenarios motiviert. Dabei werden für gängige Managementansätze Lösungsmöglichkeiten beschrieben und die sich daraus ergebenden Probleme diskutiert. Anschließend wird ein weiterführender Ansatz erörtert, welcher als Grundlage für die zu entwerfende Agentenarchitektur dienen wird.

Im darauffolgenden Abschnitt werden die Aufgabenstellung und die wichtigsten Ziele dieser Arbeit skizziert.

Abschließend werden im Abschnitt 'Gliederung' die einzelnen Kapitel der Diplomarbeit kurz vorgestellt.

1.1 Motivation

Die Motivation für Mobile Agenten ergibt sich aus der gegenwärtigen Situation des Netz- und Systemmanagements in typischen Rechnernetzen. Um die Problematik des Managements anschaulich darzustellen, wird als zu lösende Aufgabe die Erstellung einer Zugriffsstatistik eines Webservers als Beispielszenario vorgestellt. Anschließend werden verschiedene Lösungsansätze für das Beispielszenario vorgestellt. Dabei finden sich aus dem zentralen und verteilten Management Lösungsansätze. Die Begriffe 'zentral' und 'verteilt' beziehen sich in diesem Zusammenhang auf die Intelligenz. Beim zentralen Management liegt die gesamte Intelligenz bei einem Manager, während beim verteilten Management die Intelligenz auf mehrere Manager oder Agenten verteilt ist.

1.1.1 Beispielszenario einer Managementaufgabe

Als Managementaufgabe soll die Zugriffsstatistik eines Webservers ermittelt werden, dessen Logdatei als Datenbasis dient. Große Websites werden aus Gründen der Lastverteilung und Ausfallsicherheit auf mehrere Rechner verteilt. Damit gibt es nicht nur eine Logdatei, sondern mehrere im Netz verteilte Logdateien, die alle zur Erzeugung der Zugriffsstatistik herangezogen werden müssen. Die Logdateien sind oft mehrere Megabytes groß.

1.1.2 Zentrales Management

Das Management eines Rechnernetztes wird heutzutage noch oft durch eine zentrale Managementeinheit durchgeführt. Dabei wird meist als zugrundeliegende Architektur das Internet Management eingesetzt, welches, wegen seiner Einfachheit und geringem Implementierungsaufwands, eine weite Verbreitung gefunden hat.

Simple Network Management Protocol (SNMP)

SNMP (vgl. [Ros94]) hat sich zum De-facto-Standard des Internet Managements entwickelt. Die Managementeinheit, oder kurz Manager, von dem es genau einen im zu managenden Netz gibt, trifft alle Entscheidungen allein. Er wird durch einfache Agenten, die auf den Netzkomponenten ausgeführt werden, mit Informationen versorgt. Auf diese Informationen greift der Agent über eine *Management Information Base (MIB)* zu. Der Manager kann über seine Agenten auch Werte in der MIB manipulieren. Manager und Agenten haben eine feste Rollenverteilung zueinander, welche Client-Server-Beziehungen sind, wobei der Manager die Rolle des Clients und der Agent die Rolle des Servers einnimmt.

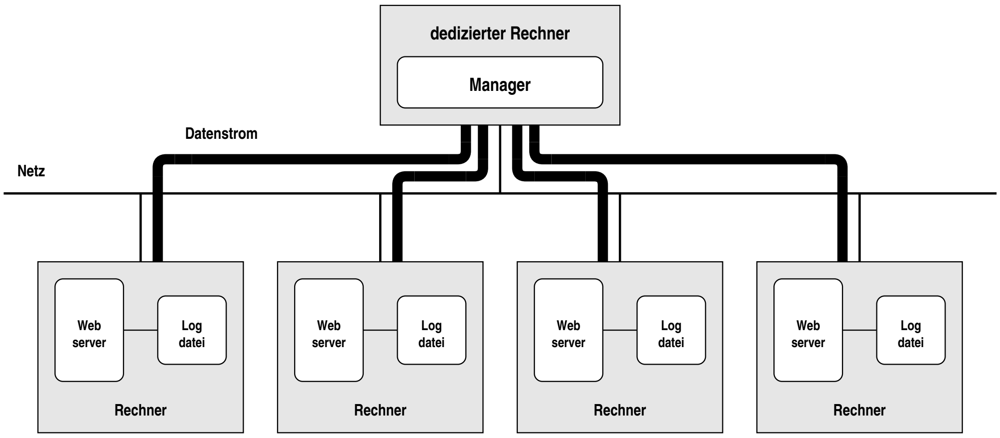

Abbildung 1.1: Lösungsansatz von SNMP für das Beispielszenario

Abbildung 1.1 verdeutlicht den Lösungsansatz des zentralen Managements für das Beispielszenario:
Da nur der Manager in der Lage ist, die Statistik zu erstellen, müssen alle Logdateien auf den Rechner des Managers transferiert werden. Dort wird dann die Statistik erstellt. Der Transfer der Logdateien verursacht einerseits eine erhebliche Netzlast und andererseits eine eventuell Überlastung des Managers, da er in der Regel auch noch viele andere Aufgaben zur gleichen Zeit zu bewältigen hat.

Es ergeben sich drei grundsätzliche Problembereiche, die mit zentralem Management nicht effizient gelöst werden können:

- ein zentraler Manager: *single point of failure*
- Heterogenität und Anzahl der zu managenden Netzkomponenten (Router, Switches, Hosts, ...), Dienste (WWW, Mail, NFS, Drucker, ...) und unterstützte Protokolle (IP, TCP, HTTP, SNMP, ...)
- ständige, dynamische Änderungen im zu managenden Netz

Der Manager muß auf einem dedizierten Rechner ausgeführt werden. Fällt dieser Rechner aus (*single point of failure*), so kann das gesamte Rechnernetz nicht mehr verwaltet werden.

Die Größe und Heterogenität des Netzes zeigt sich in mehrfacher Hinsicht als problematisch. Die Managementapplikationen werden sehr groß und unübersichtlich in der Implementierung. Es kann leicht zu einer Überlastung der Managementeinheit kommen. Genauso wird das Verhältnis der Netzlast, die alleine durch das Management verursacht wird, zur Netzlast des normalen Netzverkehrs bei steigender Anzahl der Komponenten im Netz immer schlechter.

Durch das eingesetzte Managementverfahren ist man nicht in der Lage, die Dynamik der Änderungen adäquat zu beherrschen:
Wenn eine neue Managementaufgabe zu lösen ist, muß in der Regel eine neue Applikation (oder Modul) implementieren werden, welche auf der Managementplattform kompiliert werden muß. Das erfordert häufig auch eine Rekonfiguration des Managers, oder auch eine Neukompilierung desselben. Das bedeutet eine Unterbrechung des Managements für jedes neue Modul. Darüberhinaus muß oft noch eine Portierung des Moduls für verschiedene Betriebssysteme erfolgen.

1.1.3 Statische, verteilte Ansätze

Das Hauptproblem des zentralen Managements ist, daß die ganze Intelligenz bei einem Manager liegt. Eine Verteilung der Intelligenz des Managers kann zur Behebung der Problematik beitragen. Dabei haben sich zwei Wege der Verteilung herausgebildet:

- Verschiebung von Intelligenz vom Manager auf die Agenten
- Kooperation der Manager untereinander

Die angebotenen Kooperationsmöglichkeiten, sowie die Funktionalität, die vom Manager auf die Agenten verteilt wird, ist vor der Laufzeit des Managementsystem bereits festgelegt.

Remote Network Monitoring MIB (RMON)

RMON ist ein Beispiel für die Verschiebung von Intelligenz vom Manager auf die Agenten. Mit RMON kann ein Agent die im Rechnernetz übertragenen Pakete überwachen (monitoring) und über Filtermechanismen eine Vorverarbeitung der Netzdaten vornehmen. Die Filter für den Agenten sind statisch vordefiniert und es gibt keinen Weg, diese während der Laufzeit zu ändern.

Manager-to-Manager MIB (M2M)

Die Manager-to-Manager MIB wurde mit dem Ziel entwickelt, eine Koordination zwischen den Managern zu erreichen. Damit kann eine Verteilung von Aufgaben auf verschiedene Manager erfolgen, was besonders in sehr großen Netzen von Bedeutung ist. Ein Manager stellt Dienste bereit, die von einem anderen Manager genutzt werden können. Diese Dienste sind statisch vordefiniert. Falls ein Manager neue Dienste anbieten will, muß er neu implementiert und kompiliert werden. Das gleiche gilt für die Manager, die diese Dienste nutzen wollen.

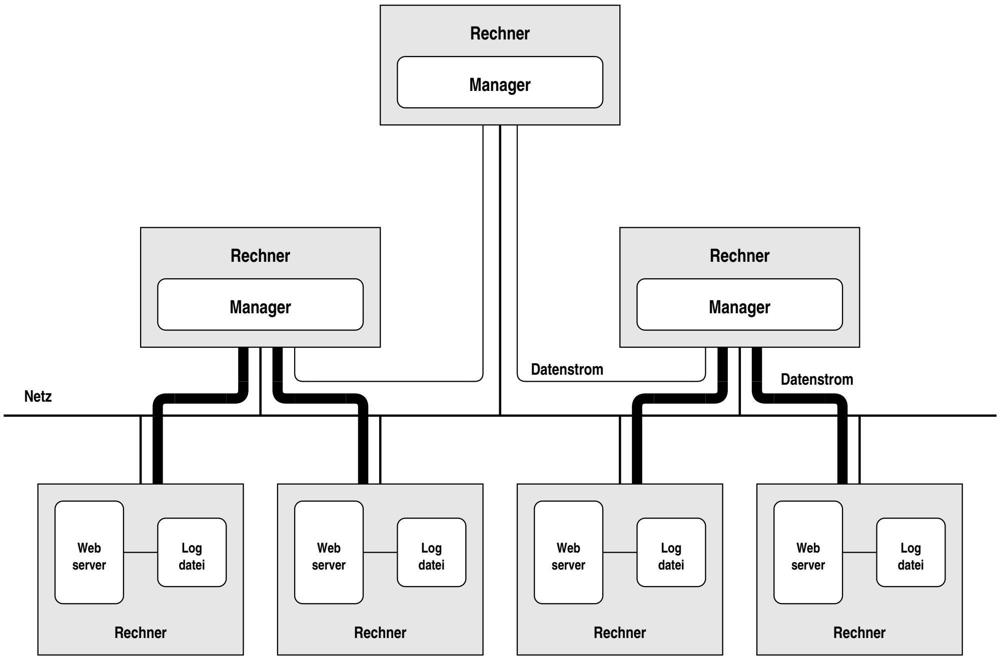

Abbildung 1.2: Lösungsansatz für das Beispielszenario anhand M2M

Der Lösungsansatz für das Beispielszenario sieht beim M2M wie folgt aus (vgl. Abbildung 1.2):
Es wird eine Schicht von Managern eingeführt, die gegenüber dem zentralen Manager die Rolle des Agenten übernehmen. Diese Zwischen-Manager bieten dem zentralen Manager den Dienst der Erstellung der Zugriffsstatistik für ihren Netzbereich an. Der zentrale Manager fordert die Teilergebnisse bei den Zwischen-Managern an und erstellt daraus die Gesamtstatistik. Man erreicht dadurch eine Entlastung des zentralen Managers, der nur noch die Teilstatistiken zusammenzufassen braucht. Auch kann sich die gesamte Netzlast bei günstiger Subnetzbildung reduzieren, falls der Datenfluß aus den Logdateien zu den Zwischen-Managern in getrennten Netzsegmenten abläuft.

Auch das statisch, verteilte Management erfordert bei einer neuen Managementaufgabe in der Regel eine Neuimplementierung, Kompilierung und Rekonfiguration des Managementsystems. Dies führt zu einem hohen Aufwand bei Änderungen und zu Ausfallzeiten des Managementsystems.

1.1.4 Dynamische, verteilte Ansätze

Um Änderungen im Management während des Betriebs des Managementsystems in den Griff zu bekommen, ist ein dynamisches und verteiltes Management notwendig. Ein vielversprechender Ansatz wird nun vorgestellt:

Management by Delegation (MbD)

In [Mou97a] (Seite 79) wird *Management by Delegation* wie folgt beschrieben:
„*MbD allows, therefore, dynamic embedding of functionality in distributed agents. Delegation consists of downloading programs to a remote agent, binding them with its local environment and executing them under local or remote control. Delegated programs provide extensibility of the management and control capabilities embedded in networked systems.*“

Die Dynamik bezieht sich hier darauf, daß **während** der Laufzeit eines Agenten dieser neue Funktionalität hinzubekommt. Dieses Modell ist sehr allgemein gehalten und es gibt deshalb verschiedene Realisierungensmöglichkeiten dieses Modells, wobei nun als Beispiel die Mid-Level-Manager MIB vorgestellt wird.

Mid-Level-Manager MIB

Der Mid-Level-Manager ist ein erweiteter SNMP Manager, der als Manager *und* Agent zur selben Zeit agieren kann. Es wurde im wesentlichen eine *SNMP scripting language*, eine Interpretersprache entwickelt, die SNMP-Operationen ausführt. Die Managementskripte werden mit Hilfe von SNMP Version 2 (SNMPv2) delegiert. SNMPv2 eignet sich nicht als Delegierungsprotokoll, da die Paketgröße stark beschränkt ist. So muß das Managementskript zeilenweiße auf die MIB des Agenten übertragen werden, was mit einem beachtlichen Mehraufwand verbunden ist.

1.1.5 Weiterführende Realisierung von MbD

Den bisherigen Realisierungen des MbD-Paradigma fehlt die Flexibilität im Ansatz, d. h. es wird zwar das Management dynamisch verteilt, es werden aber zusätzlich Kommunikationsmöglichkeiten und die Mobilität für Agenten benötigt. Agenten haben beim verteilten Management nur eine eingeschränkte Sicht auf das Managementsystem. Durch Kommunikation können Agenten sich die Informationen beschaffen, die sie zur Lösung ihrer Managementaufgabe benötigen. Die Mobilität bietet eine große Flexibilität des Agenten. Der Agent kann sich frei im Netz bewegen, um seine Managementaufgabe zu erfüllen. Falls ein Host, auf dem sich ein Mobiler Agent befindet, abgeschaltet werden muß, so kann der Agent auf einen anderen Rechner migrieren und seine Aufgabe weiter, ohne Neustart erfüllen. Agenten werden nun auch dynamisch geladen und es bedarf keiner Rekonfiguration des Managers. Durch die Mobilität kann auch ein Lastausgleich durchgeführt werden, in dem Agenten von überlasteten Rechnern auf weniger belastete Rechner transferiert werden.

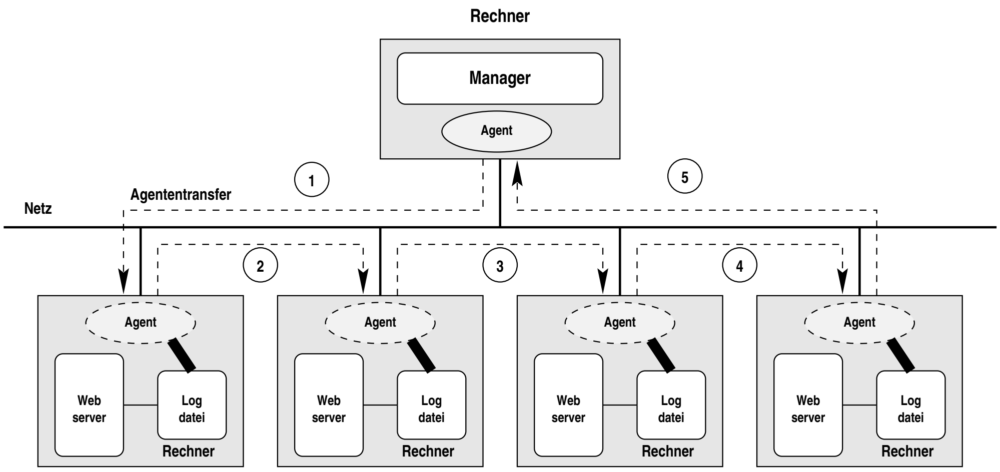

Abbildung 1.3: Lösungsansatz mit Mobilen Agenten für das Beispielszenario

Ein weiterer Vorteil Mobiler Agenten wird aus dem Lösungsansatz für das Beispielszenario ersichtlich (Abbildung 1.3). Der Manager beauftragt einen Agenten, die Zugriffsstatistik zu erstellen. Der Agent wandert von Rechner zu Rechner, erstellt jeweils Teilstatistiken, in dem er lokal auf die Logdateien zugreift und kehrt zum Manager mit der Gesamtstatistik zurück. Der Manager liest abschließend die Statistik aus dem Agenten aus. Man kann davon ausgehen, daß die Größe des Agenten viel kleiner ist, als die Größe der Logdateien, da der Agent nur die relevanten Daten der Statistik mit sich führt.

1.2 Aufgabenstellung

Nachdem Mobile Agenten in einem dynamischen, verteilten Ansatz motiviert sind, wird nun die Aufgabenstellung vorgestellt.

Es ist eine Agentenarchitektur zu konzipieren, realisieren und implementieren. Das Einsatzszenario dieser Architektur ist das Netz- und Systemmanagement. Dabei soll als Basisansatz, für die zu entwickelnde Architektur, das MbD-Paradigma dienen. Wie aus dem vorherigen Abschnitt hervorgegangen ist, bietet der Ansatz der kommunizierenden, mobilen Agenten eine vielversprechende Realisierungsmöglichkeit. Deshalb wird besonderer Wert auf die Erreichung der Ziele Kommunikation und Mobilität gelegt.

Als Grundlage für die Implementierung soll der *Flexible Management Agent*, der im Abschnitt 2.3 vorgestellt wird, dienen.

Da es sich bei der zu entwickelnden Agentenarchitektur der Aufgabenstellung, um eine Managementarchitektur handelt, werden diese Begriffe im weiteren synonym verwendet.

1.2.1 Eingrenzung des Agentenbegriffs

Da es weder in der Industrie, noch in der Wissenschaft einen einheitlichen Agentenbegriff gibt, werden nachfolgend charakteristische Eigenschaften eines Agenten angegeben, um zu einem gemeinsamen Verständnis von Agenten zu gelangen:

- ein Agent handelt **zielgerichtet**
- ein Agent ist **autonom**, handelt ohne direkte Einflußnahme des Menschen
- ein Agent kann mit anderen Agenten **kommunizieren**
- ein Agent kann seine Umwelt **wahrnehmen** und darauf reagieren

Der so festgelegte Agentenbegriff wird als Grundlage für diese Arbeit genommen und in Kapitel 2 weiter ausgeführt.

Es wird nochmals darauf hingewiesen, daß es sich bei den betrachteten Agenten um solche handelt, deren Einsatzgebiet das Netz- und Systemmanagement in einem **begrenzten** Rechnernetz ist und deshalb sogenannte *Internet Robots* und *Spiders*, die im gesamten Internet ihre Aufgabe erfüllen, nicht betrachtet werden [Che96].

Intelligente Agenten, wobei 'intelligent' im Sinne von lernfähig zu verstehen ist, spielen im Bereich der Künstlichen Intelligenz (KI) eine immer größere Rolle. Diese Art der Agenten und die Anforderungen, die sich daraus ergeben, werden in dieser Arbeit nicht näher untersucht.

1.3 Gliederung

Die weitere Struktur dieser Diplomarbeit ist wie folgt:
In Kapitel 2 wird die Anforderungsanalyse, welcher die zu entwerfende Agentenarchitektur genügen soll, erarbeitet. Zwei Agentenarchitekturen werden vorgestellt und auf ihre Tauglichkeit bezüglich der Anforderungsanalyse betrachtet.

Eine Software-Entwicklungsmethode, eine Kommunikationsinfrastruktur (Middleware) und eine Implementierungssprache werden als mögliche Techniken der Realisierung in Kapitel 3 vorgestellt.

Nachdem die Grundlagen gelegt sind, kann ein erster, konzeptioneller Entwurf des Agentensystems erfolgen. Die wichtigsten Designentscheidungen werden in Kapitel 4 beschrieben.

Aus der Konzeption heraus entsteht in Kapitel 5 die Realisierung der Agentenarchitektur. Die Realisierung besteht aus dem Objektmodell, das in den enthaltenen Abschnitten detailliert erklärt wird.

In Kapitel 6 wird die Implementierung der Agentenarchitektur anhand der wichtigsten Klassen und Algorithmen beschrieben. Die Entwicklung neuer Agenten und deren Einbindung in die Architektur wird erläutert.

Abschließend findet in Kapitel 7 die Zusammenfassung der Arbeit und ein Ausblick auf mögliche Erweiterungen der implementierten Agentenarchitektur statt.

Kapitel 2

Anforderungen, bestehende Agentenarchitektur und Agenten-Standards

Agentenarchitekturen können nur sinnvoll verglichen und bewertet werden, wenn man eine einheitliche Beschreibung der Architekturen wählt und Anforderungen bzw. Kriterien aufstellt, anhand derer die Bewertung erfolgen kann. In den nächsten Abschnitten wird das Beschreibungsmodell vorgestellt, sowie die Anforderungen spezifiziert. Anschließend findet eine Beschreibung und Bewertung der beiden Agentenarchitekturen, *Flexible Managenent Agent* und *Mobile Agents Facility* statt.

2.1 Beschreibung von Managementarchitekturen

Zur Beschreibung von Managementarchitekturen sind in [HA93] vier Teilmodelle definiert worden, welche nun kurz vorgestellt werden:

- **Informationsmodell**: Konzepte zur Beschreibung der Management Information (MI) und Managed Objects (MOs)[1]; deren Auswahl und Zugriff
- **Kommunikationsmodell**: Festlegung der kommunizierenden Partner, Syntax und Semantik der Austauschformate und Einbettung in den Protokollstack
- **Funktionsmodell**: Aufteilung des gesamten Managements in Aufgabenbereiche, z. B. Abrechnungs-, Fehler- und Konfigurationsmanagement
- **Organisationsmodell**: Festlegung von Rollen und Beziehungen der kommunizierenden Partner sowie deren Gruppierung (Domänenbildung)

[1]Ein Managed Object ist die Abstraktion einer realen Ressource

In der nun folgenden Anforderungsanalyse wird darauf eingegangen, inwiefern die 4 Teilmodelle, zumindest schon grob, festgelegt werden können.

2.2 Anforderungsanalyse

In diesem Abschnitt werden Anforderungen, die an eine Agentenarchitektur gestellt werden, festgelegt. Im nächsten Unterabschnitt werden grundlegende Anforderungen aufgestellt, aus denen anschließend Folgerungen abgeleitet werden. Die Gesamtheit aus grundlegenden Anforderungen und Folgerungen bildet die Anforderungsanalyse.

2.2.1 Grundlegende Anforderungen

Die folgenden fünf Anforderungen an eine Agentenarchitektur werden als Grundlage dienen.

Die ersten drei Anforderungen sind aus [Mou97a] entnommen:

- Jeder Agent besitzt als Aufgabe das **Management einer Ressource**. Die Agenten unterscheiden sich damit in ihren verschiedenen Aufgabenbereichen. Durch eine Aufgabe wird die Funktionalität eines Agenten festgelegt.

  Die Eingliederung der Agenten in das oben beschriebene Funktionsmodell ist damit bereits erfüllt.

- Agenten können zur Realisierung einer Managementaufgabe in **Gruppen** zusammengefaßt werden. Dabei kann ein Agent mehreren Gruppen angehören. Gruppen können nach verschiedenen Gesichtspunkten gebildet werden. Beispiele für Gruppierungen sind alle Agenten für eine bestimmte Aufgabe, oder Agenten, die voneinander abhängig sind, zu einer Gruppe zusammenzufassen.

  Die Gruppenbildung ist ein Aspekt des Organisationsmodells.

- Um die Flexibilität und die Verteilung der Agentenarchitektur zu gewährleisten, soll ***Management by Delegation*** unterstützt werden.

- Jeder Agent soll **managebar** sein, d. h. sein gesamter Lebenszyklus kann gesteuert werden.

- Jeder Agent soll eine **graphische Benutzerschnittstelle (GUI)** zur komfortablen Eingabe von Anfragen und zur visuellen Darstellung der Ergebnisse bereitstellen.

- **Sicherheit** muß für folgenden Fälle gewährleistet werden:
  - ein Agent kann vertrauliche Daten halten und muß deswegen vor unautorisiertem Zugriff geschützt werden

– Ressourcen des zu managenden Systems müssen vor dem Zugriff eines Agenten geschützt werden können

2.2.2 Folgerungen

Damit ein Agent die ihm gestellte Aufgabe erfüllen kann, muß er mit anderen Agenten **kommunizieren** können. Die Möglichkeit der Kommunikation ist essentiell, da jeder Agent nur einen bestimmten Teilbereich des zu managenden Netzes wahrnimmt und deshalb eventuell Dienste eines anderen Agenten beanspruchen muß.

Bei der Kommunikation handelt es sich um den Austausch von Informationen und es soll die symmetrische und asymmetrische Kommunikation unterstützt werden. Zur Durchführung der Kommunikation wird eine **Kommunikationsinfrastruktur** benötigt. Damit ist das Kommunikationsmodell festgelegt.

Um einen möglichst flexiblen Ansatz zu erreichen, werden bei den kommunizierenden Partnern keine festen Rollen im vorhinein festgelegt, sondern jeder Agent soll je nach Situation verschiedene Rollen annehmen können.

Damit die Agenten kommunizieren können, muß jeder Agent eindeutig **identifizierbar** und **lokalisierbar** sein. Dazu ist ein **Verzeichnisdienst** notwendig, der eine Abbildung von einem Identifikator, z. B. einem Namen, zum Aufenthaltsort des Agenten vornimmt.

Das *Management by Delegation* Paradigma wird durch die **Mobilität** der Agenten erreicht. Der Agent, der Funktionalität kapselt, kann von Rechner zu Rechner transferiert werden. Es wird damit Funktionalität delegiert.

Aus der Mobilität folgt die **Serialisierbarkeit** eines Agenten. Serialisierbarkeit bedeutet, daß aus dem Agenten, einem ablaufenden Programm, ein Datenstrom erzeugt wird, der über ein Rechnernetz transferiert werden kann. Die Möglichkeit der **persistenten Speicherung** eines Agenten nutzt auch die Serialisierbarkeit aus. Die Speicherung ermöglicht z. B. den Neustart eines Rechners mit anschließenden Wiedereinspielen des Agenten. Dabei wird u. a. der Zustand des Agenten wiederhergestellt.

Damit Agenten kommunizieren können erhält jeder Agent eine **Schnittstelle**, an der seine **Dienste** offeriert werden. Ein Dienst besteht aus einer Operation zur Identifizierung des Dienstes und aus der angebotenen Managementinformation. Somit ist das Informationsmodell des Agenten festgelegt.

Aus der Forderung der Managebarkeit, der Sicherheit und der Mobilität der Agenten folgt, daß eine **Laufzeitumgebung** nötig ist. Die Laufzeitumgebung kann die oben genannten Sicherheitsbedingungen durchsetzen, den Agenten managen, serialisieren, sowie den serialisierten Agenten zu einer anderen Laufzeitumgebung senden. Damit der Agententransfer zwischen Laufzeitumgebungen stattfinden kann, müssen diese, wie der Agent, identifizierbar und lokalisierbar sein.

Aus den gestellten Anforderungen ergibt sich eine Reihe von Basisdiensten:

- **Verzeichnisdienst (Naming Service)**: Unterstützt die Identifizierung und die Lokalisierung von Agenten und Laufzeitumgebungen.
- **Ereignisdienst (Event Service)**: Bietet Agenten die Möglichkeit der asynchronen Kommunikation. Die Möglichkeit der Filterung von Ereignissen soll für jeden Agenten vorhanden sein.
- **Gruppierungsdienst**: Organisiert die Gruppierung von Agenten und stellt einen Gruppenkommunikationsmechanismus zur Verfügung.

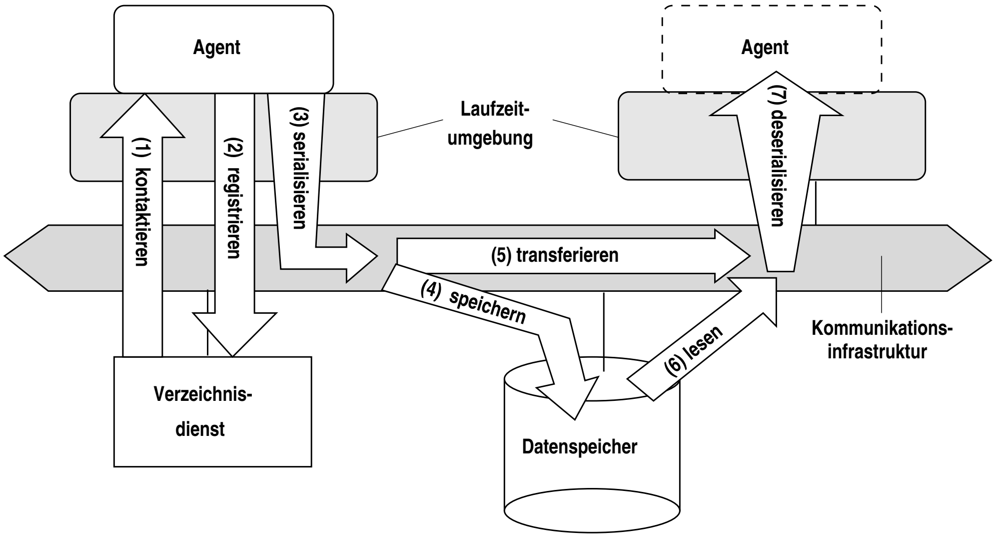

Abbildung 2.1: Architektur der Anforderungsanalyse

Abbildung 2.1 zeigt die prinzipielle Architektur der Anforderungsanalyse und mögliche Aktionen des Agenten, wobei die Pfeile die Richtung des Informationsflusses anzeigen. Wenn ein Agent erzeugt wurde, kontaktiert (1) er den Verzeichnisdienst und registriert (2) sich bei diesem. Damit ist der Agent für andere Agenten auffindbar. Wenn der Agent persistent gespeichert (4) oder transferiert (5) werden soll, so muß das Laufzeitsystem den Agenten zuerst serialisieren (3). Auf der Ziellaufzeitumgebung wird anschließend der Agent deserialisiert (7), wobei der serialisierte Agent entweder von einem Quelllaufzeitsystem transferiert wird (5), oder aus einem Datenspeicher gelesen wird (6).

Die Komponenten der Agentenarchitekturen sind festgelegt. Damit die verschiedenen Komponenten auf eine definierte Weise interagieren können, müssen Schnittstellen definiert werden:

- Schnittstelle zwischen Agenten und Laufzeitumgebung:
  - der Agent zeigt den Transferwunsch der Laufzeitumgebung an
  - die Laufzeitumgebung bietet den Agenten Informations- und Manipulationsmöglichkeiten

- Schnittstelle zwischen den Laufzeitumgebungen: zur Übertragung des serialisierten Agenten
- Schnittstelle zwischen Basisdiensten und Agenten bzw. Laufzeitumgebung: die Basisdienste müssen zu ihrer Benutzung eine Schnittstelle anbieten
- Schnittstelle zwischen zu managenden Ressourcen und Agenten: damit der Agent auf die Ressource zugreifen kann
- Schnittstelle zwischen Agenten und Benutzer: ist die graphische Benutzerschnittstelle (GUI)

Einen Manager, wie er im zentralen Ansatz des Managements existiert, gibt es nicht mehr. Der Agent übernimmt nun neben der Bereitstellung von Information, auch die Verarbeitung und führt Managementaktionen aus. Das Management der Agenten wird vom Laufzeitsystem übernommen. D. h., daß die Funktion des Managers von Agenten und von den Laufzeitumgebungen übernommen werden.

Folgende Problemfelder können identifiziert werden, die sich speziell für die Architektur der Anforderungsanalyse ergeben und die nicht im zentralen Management auftreten:

- Wie erfolgt die technische Realisierung der Serialierung/Deserialisierung?
- Gibt es bereits standardisierte Schnittstellen zwischen den Komponenten?
- Wie wird der Verzeichnisdienst bei der Mobilität von Agenten konsistent gehalten?
- Wie wird die graphische Benutzeroberfläche bei Mobilen Agenten realisiert?
- Wie erlangt der Administrator eine globale Sicht auf das Managementsystem?
- Wie können die Sicherheitsanforderungen, die durch die Mobilität der Agenten auftreten, durchgesetzt werden? (Ein Mobiler Agent könnte auch ein Angreifer sein!)

Die Antworten auf diese Fragen werden in den weiteren Kapiteln erarbeitet und zusammengefaßt in Abschnitt 4.7 vorgestellt.

Da man von der Heterogenität eines zu managenden Netzes ausgehen muß, ist die Kommunikationsinfrastruktur, sowie die Implementierungssprache der Laufzeitumgebung und der Agenten so zu wählen, daß ein flächendeckendes Management möglich ist. Besonders die Wahl der Implementierungssprache ist unter dem Aspekt der Mobilität der Agenten von großer Bedeutung.

Nun folgt die Beschreibung und Bewertung zweier Managementarchitekturen, die als Basis der zu entwerfenden Managementarchitektur dienen können.

2.3 Flexible Management Agent (FMA)

Die Architektur des ***Flexible Management Agent (FMA)*** wurde im Rahmen einer Dissertation [Mou97a] entworfen und im Rahmen zweier Diplomarbeiten ([Wen97],[Hol97]) am Lehrstuhl für technische Informatik – Rechnernetze der Technischen Universität München implementiert. Diese Architektur und deren Implementierung bildet die Grundlage dieser Diplomarbeit, von der aus eine neue Architektur zu entwickeln ist. Die Analyse der FMA ist deshalb besonders wichtig. Die Architektur besteht aus drei verschiedenen Typen von Managementeinheiten, die hierarchisch gegliedert sind:

- **Manager**: delegiert Aufgaben an flexible Agenten und kontrolliert diese
- **Flexible Agenten**: eine Einheit, die eine bestimmte Aufgabe erfüllen soll. Dazu können sie untereinander kommunizieren, kooperieren und auch Aufgaben delegieren. Zur Erfüllung der Aufgaben stützt man sich auf
- **Agenten**: die Managementinformationen liefern, z. B. SNMP-Agenten

Flexible Agenten können zu Gruppen zusammengefaßt werden.

2.3.1 Realisierung von FMAs

Die Implementierung ist im wesentlichen eine Erweiterung des *Java Agent Template (JAT)*, welches an der Stanford University [Fro97] entwickelt wurde. Die Architektur, siehe Abbildung 2.2, besteht aus einem ***Flexible Management Agent***, einer Laufzeitumgebung, die vor allem die Kommunikation mit anderen FMAs und die Fähigkeit zur Laufzeit neue Funktionalität hinzuzunehmen, bereitstellt. Neue Funktionalität ist in einem **Interpreter** gekapselt, der dem Agentenbegriff der Anforderungsanalyse entspricht.

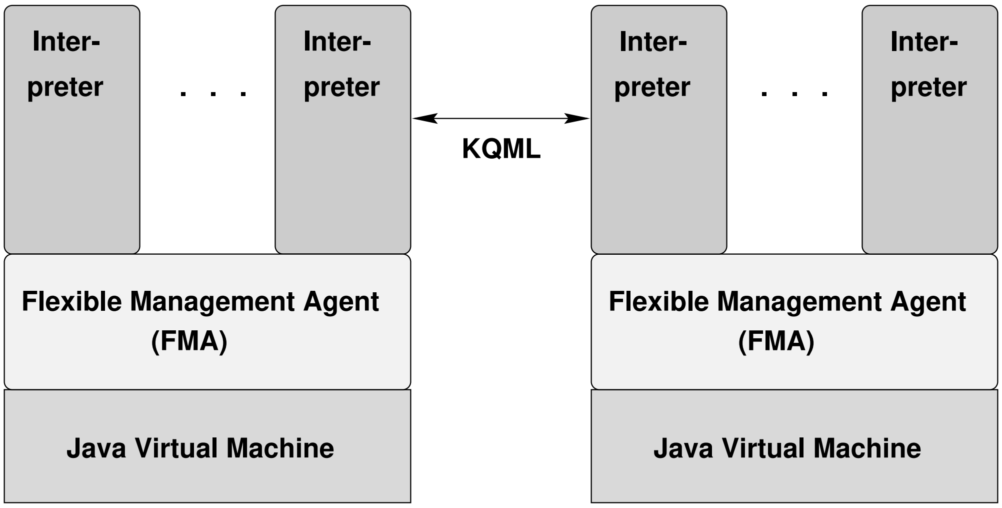

Abbildung 2.2: FMA-Architektur

Der FMA und die Interpreter sind in Java implementiert und werden innerhalb der Java Virtual Machine (JVM) ausgeführt. Eine Beschreibung der Managementinformationen fehlt bei dieser Architektur vollkommen. Als Kommunikationsprotokoll wird die ***Knowledge Query and Manipulation Language (KQML)*** verwendet. KQML wurde speziell für die Kommunikation von Agenten entwickelt. Eine KQML-Nachricht besteht aus Schlüssel-Werte-Paaren, angelehnt an die Programmiersprache Lisp.

Das Funktionsmodell besteht aus Naming und Event Service, die als Interpreter realisiert sind. Weiterhin sind Interpreter für Prozeß-, Maschinen-, Webmonitoring, etc. implementiert. Interpreter kommunizieren miteinander, um eine bestimmte Aufgabe zu bewältigen. Die Rollen und Beziehungen der Interpreter sind nicht fest vordefiniert.

Beim Naming Service wird jeder Interpreter angemeldet. Der Naming Service muß auf einem fest vordefinierten Host ausgeführt werden, damit ihn die Agenten erreichen können. Dieser Ansatz ist sehr inflexibel und störanfällig (*single point of failure*). Der implementierte Naming Service bietet weiterhin keine Domänenbildung für Agenten an.

Ereignisse werden an den Event Service geschickt und haben ebenfalls das Format einer KQML-Nachricht. Interpreter, die an einem bestimmten Ereignis interessiert sind, können sich für dieses beim Event Service anmelden (subscribe). Ist der Interpreter an dem Ereignis nicht mehr interessiert, so kann er sich beim Event Service abmelden (unsubscribe). Der Event Service ist zentral, d. h. es gibt genau einen Event Service im zu managenden Netz.

Jeder FMA kann zwar neue Interpreter hinzunehmen, die Fähigkeit, Interpreter von einem FMA zu einem anderen FMA zu senden, gibt es nicht.

2.3.2 Fazit

Die größte Schwäche der Realisierung ist die fehlende Schnittstellenspezifikation der Interpreter. Ein Client muß wissen, welche Nachrichten ein Interpreter versteht bzw. sendet. Dieses Wissen kann er nur aus einer Dokumentation erhalten.

KQML wurde für die Kommunikation von intelligenten Agenten entwickelt. Mit KQML kann man verschieden Datenformate übertragen, es existieren aber nur Implementierungen, die lediglich in ASCII verfaßte KQML-Nachrichten verarbeiten können. Damit wird die Übertragung von komplexen Datenstrukturen, wie z. B. Graphen unkonfortabel, da der Programmierer das Ein- und Auspacken (Marshalling) der Nachrichten selbst implementieren muß.

Auch bei der Implementierung der FMA-Architektur gibt es Defizite. Die in der Anforderungsanalyse gestellte Forderung nach Mobilität der Agenten wird nicht unterstützt.

Der Naming Service muß auf einem festen Host über einen festgelegten Port erreichbar sein. Dieser Ansatz ist sehr inflexibel und ein *single point of failure*. Darüberhinaus unterstützt der Naming Service keine Domänenbildung von Agentennamen.

Es ist eine mächtigere Kommunikationsinfrastruktur wünschenswert, die folgende Eigenschaften hat (vgl. Abschnitt 3.3):

- Kommunikation über festgelegte Schnittstellen
- Werkzeugunterstützung zur Generierung von Stubs damit der Programmierer nicht das Marshalling übernehmen muß
- Bereitstellung von Standard-Basisdiensten wie Verzeichnis- und Ereignisdienst

2.4 Mobile Agents Facility (MAF)

Es wird nun eine weitere Architektur für Mobile Agenten vorgestellt, die als mögliche Lösung der gestellten Aufgabe in Betracht kommt. Die meisten der hier vorgestellten Konzepte fließen in den Entwurf dieser Arbeit ein.

2.4.1 Motivation und Überblick

Diese Spezifikation einer Architektur für Mobile Agenten wurde von der Object Management Group (OMG) im November 1997 herausgegeben [GMD97]. Ziel der Spezifikation ist eine Standardisierung von Agentensystemen. Ein Agentensystem entspricht der Laufzeitumgebung der Anforderungsanalyse. Durch die Standardisierung soll eine möglichst hohe Interoperabilität zwischen verschiedenen Agentensystemen, die diesen Standard einhalten, erreicht werden. Wichtigste Standardisierungsbereiche dieser Spezifikation sind:

- das **Agentenmanagement**, damit z. B. ein Administrator Agenten auf unterschiedlichen Typen von Agentensystemen mit den gleichen Operationen managen kann
- der **Agententransfer** zur Ermöglichung der Mobilität von Agenten
- die **Namensgebung** für Agenten und Agentensysteme, dient der Identifizierung
- die **Lokalisierung** von Agenten und Agentensystemen, damit Kommunikationspartner gefunden werden können

Wichtige Bereiche, die **nicht** spezifiziert werden sind:

- Inter-Agentenkommunikation
- Ausführung von Agenten
- Serialisierung/Deserialisierung von Agenten

- Schnittstelle zwischen Agentensystem und darauf ablaufenden Agenten

Diese Architektur ist nicht speziell für das Management konzipiert worden, sondern allgemein für alle Bereiche, die für den Einsatz von Mobilen Agenten geeignet sind.

2.4.2 Terminologie, Konzepte und Architektur

Es wird nun auch ausführlich auf die Terminologie eingegangen, da diese in der Realisierung Verwendung findet. Ein ***Agent*** ist ein Programm, das innerhalb eines eigenen Thread ausgeführt wird, somit autonom ist und im Auftrag einer ***Authority*** (Person oder Organistion) handelt. Ein ***Stationary Agent*** ist ein Agent, der nur auf dem Host auf dem er gestartet wurde, ausgeführt wird. Im Gegensatz dazu kann ein ***Mobile Agent*** von Agentensystem zu Agentensystem wandern, um seine gestellte Aufgabe zu erfüllen.

Ein ***Agent System*** stellt eine Laufzeitumgebung für Agenten dar, besitzt einen eindeutigen ***Agent System Type***[2] und wird ebenfalls von einer *Authority* gestartet. Agenten befinden sich innerhalb des Agentensystems in einer Ausführungsumgebung, den sogenannten ***Places***. Ein *Place* kann eine Umgebung sein, in der z. B. bestimmte Zugriffsrechte auf Ressourcen gelten.

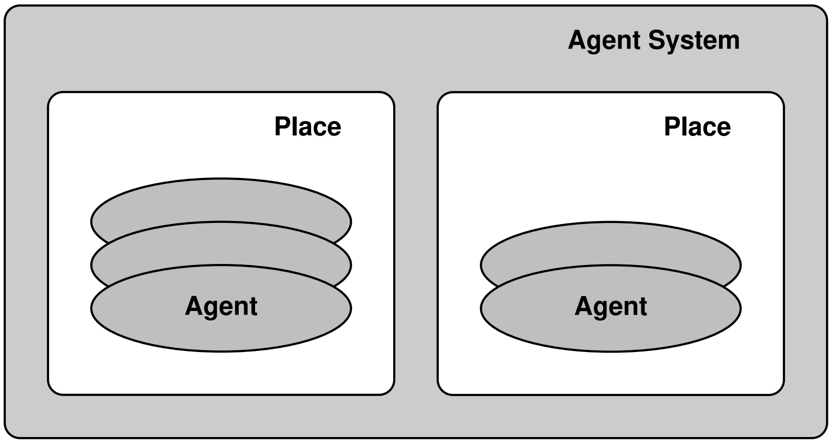

Abbildung 2.3: Agentensystem nach der MAF-Spezifikation

Agenten und Agentensysteme besitzen jeweils einen eindeutigen Identifikator (***Name***), wobei ein *Name* aus folgenden Teilen besteht:

- ***Authority***: Name der Person oder Organistion
- ***Identity***: ein eindeutiger Wert innerhalb eines Agentensystems
- ***Agent System Type***: Typ des Agentensystems

[2] Dieser Typ ist eine Zahl und wird von der OMG vergeben (z. B. Aglets, Mobile Agenten von IBM haben '1' als *Agent System Type*.)

Der so zusammengesetzte *Name* muß global eindeutig sein.

Mehrere Agentensysteme mit derselben *Authority* können zu einer ***Region*** zusammengefaßt werden. Mit einer *Region* kann vor allem eine Lastverteilung erreicht werden.

Agentensysteme kommunizieren untereinander mit Hilfe einer ***Communication Infrastructure***, welche jedoch nicht näher bestimmt ist.

Die Funktion des Naming Service übernimmt der ***MAFFinder***. Pro *Region* wird genau ein *MAFFinder* benötigt. Bei ihm werden Agenten, Agentensysteme und *Places* registriert, deregistriert und gesucht. Als Ergebnis einer Suchanfrage nach einem Agenten, wird eine ***Location*** zurückgegeben. Eine *Location* besteht aus der Adresse des Agentensystems und dem *Place*, entweder in Form eines ***Uniform Resource Locators (URL)*** oder eines ***Uniform Resource Identifiers (URI)***. D. h. man bekommt **nicht** die Referenz auf einen Agenten, sondern immer nur die Adresse des Agentensystems, auf dem der Agent läuft.

Es werden in der MAF-Spezifikation zwei Schnittstellen mit Hilfe der ***Interface Definition Language (IDL)***, einer Schnittstellenbeschreibungssprache (vgl. Abschnitt 3.3) definiert:

- *`MAFAgentSystem`*: für das Agentenmanagement und Agententransfer
- *`MAFFinder`*: Naming Service für Agenten, Agentensysteme und *Places*

Agenten müssen keine IDL-Schnittstelle definieren.

2.4.3 Funktionen eines Agentensystems

Im Folgenden werden exemplarisch drei wichtige Funktionen eines MAF-konformen Agentensystems, nämlich Erzeugung, Transfer und Terminierung eines Agenten, welche alle in der IDL-Schnittstelle *`MAFAgentSystem`* beschrieben sind, genauer vorgestellt. Dieser Lebenszyklus eines Agenten wird auch in den nachfolgenden Kapiteln immer wieder aufgegriffen.

Während des Lebenszyklus eines Agenten kann dieser sich in verschiedenen Zuständen befinden. Abbildung 2.4 veranschaulicht die Zustandsübergänge und die zugehörige Beschreibung folgt in den nachfolgenden Unterabschnitten.

Erzeugung eines Agenten

Agenten können von einem beliebigen Client, der Zugriff auf die IDL-Schnittstelle *`MAFAgentSystem`* mit der Operation `create_agent()`, erzeugt werden. Der Client muß sich gegenüber dem Zielagentensystem authentifizieren, wobei das Authentifizierungsverfahren nicht näher festgelegt wird.

Der Agent wird innerhalb eines *Places* erzeugt. Ein Thread wird instanziiert, in dem der Agent ablaufen kann. Falls nötig, wird ein innerhalb der *Region*

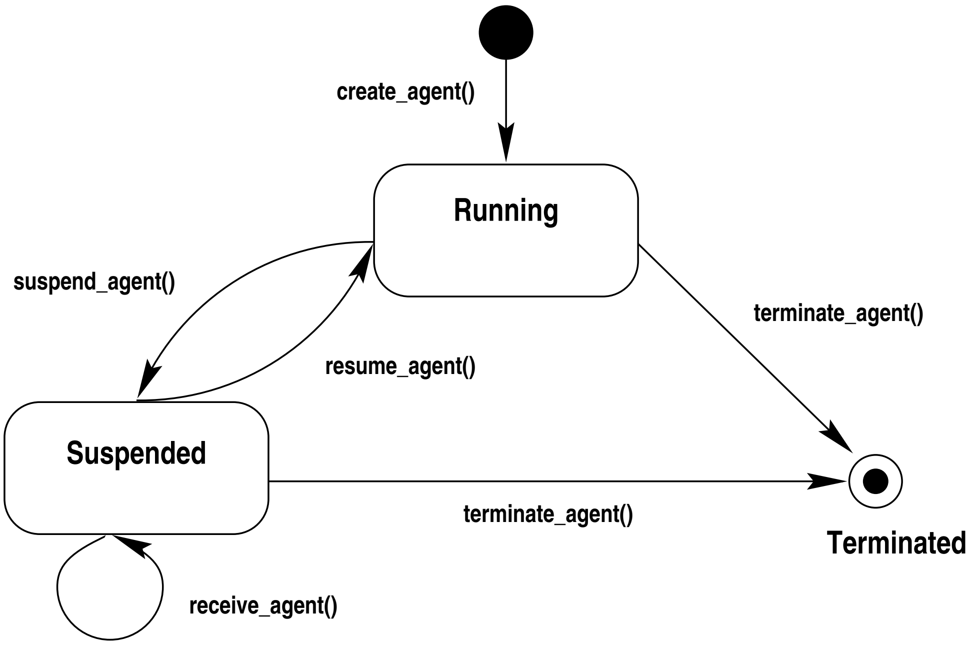

Abbildung 2.4: Zustandsübergangsdiagramm eines Agenten

global eindeutiger *Name* generiert. Schließlich wird der Agent in seinem Thread gestartet. Der Agent befindet sich im Zustand `Running`.

Transfer von Agenten

Ein Mobiler Agent, welcher auf ein anderes Agentensystem migrieren will, muß diesen Wunsch dem Agentensystem, auf dem er gerade läuft, durch einen internen API-Aufruf mitteilen. Dazu muß der Mobile Agent sein Zielagentensystem in Form einer *Location*, sowie die benötigte Qualität der Kommunikation, (z. B. Bandbreite), seinem Ausgangssystem mitteilen. Jegliche Quality of Service (QoS)-Anforderungen werden in der Spezifikation nicht näher beschrieben. Das Ausgangs-Agentensystem muß nun folgende Schritte durchführen: Der Agent wird gestoppt (`suspend_agent()`). Die zu transferierenden Attribute des Agenten werden bestimmt. Der Agent wird serialisiert und für das gewählte Transportprotokoll kodiert. Die beiden beteiligten Agentensysteme müssen sich gegenseitig authentifizieren. Nun kann der Agent übertragen werden (`receive_agent()`).

Das Zielagentensystem muß zuerst feststellen, ob es den Agenten interpretieren kann. Der Agent wird dekodiert und deserialisiert. Nun erfolgt die Wiederherstellung des Agentenzustandes. Abschließend wird die Ausführung des Agenten fortgesetzt (`resume_agent()`). Der Agent befindet sich wieder im Zustand `Running`.

Terminierung eines Agenten

Nachdem der Agent seine Aufgabe erfüllt hat, kann er über die IDL-Schnittstelle `MAFAgentSystem` terminiert werden. Dabei werden sämtliche Ressourcen, die der Agent beansprucht hat, freigegeben und der Agent vollständig aus dem Agentensystem entfernt. Der Lebenszyklus des Agenten ist damit beendet.

2.4.4 Sicherheit

Die Spezifikation beschreibt ausführlich die möglichen Bedrohungen des Systems, z. B. unautorisierter Zugriff auf Informationen, Spamming, Spoofing, etc. Die wichtigsten Sicherheitsanforderungen werden nun vorgestellt:

- Authentifizierung von Clients, welche keine Agentensysteme sind. Dieser Authentifizierungsvorgang kann durch eine Paßwortabfrage erfolgen.
- Gegenseitige Authentifizierung von Agentensystemen für z. B. einen Agententransfer: Hier kann die Authentifizierung nicht durch eine Paßwortabfrage erfolgen, da die Agentensysteme autonom handeln. Ein Möglichkeit wäre, daß bei der **Erzeugung** des Agentensystems der Administrator über eine Paßwortabfrage dem Agentensystem Zugriff auf geheime Informationen, z. B. einem vertraulichen Schlüssel, gibt, der dann zur Authentifizierung herangezogen werden kann.

  Die gegenseitige Authentifizierung garantiert dem Agenten den Schutz seiner vertraulichen Daten auf dem Zielagentensystem.
- *Security Policies* für Agenten und Agentensysteme: Einem Agenten bzw. Agentensystem soll damit die Möglichkeit gegeben werden, eine Zugriffskontrolle für seine Methoden zu erreichen.
- Stufen der Sicherheit bei der Rechnernetzkommunikation:
  - Vertraulichkeit (Confidentiality) des Kommunikationsweges.
  - Integrität (Integrity) des Kanals, damit Verfälschungen erkannt werden.
  - Gegenseitige Authentifizierung (Authentication) der Agentensystems bevor ein Agent migriert, damit ein unautorisierter Zugriff auf Informationen, die der Agent mit sich führt, verhindert wird.
  - Wiedereinspielung (Replay attack) kann dadurch verhindert werden, daß das Agentensystem einen entsprechenden Algorithmus bereitstellt, der das Duplizieren erkennt.

  Ein Agent/Agentensystem, welches eine Kommunikation initiieren will, bestimmt die Sicherheitsstufe. Entweder die Kommunikationsinfrastruktur kann der Anforderung entsprechen, oder sie lehnt den Kommunikationswunsch ab.

2.4.5 Bewertung der MAF-Spezifikation

Die MAF-Spezifikation deckt wichtige Bereiche der Anforderungsanalyse, wie Laufzeitumgebung, Agentenmanagement und Mobilität von Agenten, ab und ist deshalb als Basis einer Agentenarchitektur gut geeignet.

Als Verzeichnisdienst kann der ***MAFFinder*** verwendet werden.

Die Sicherheitsanforderungen aus Kapitel 2.2 werden in der MAF-Spezifikation eingehend behandelt.

Um die Spezifikation zu komplettieren, müssen folgende Bereiche noch festgelegt werden:

- Festlegung des Agenten, vorallem seine angebotenen Dienste in einer Schnittstelle.
- Interne Schnittstelle zwischen Laufzeitumgebung und Agenten
- GUI der Laufzeitumgebung und Agenten
- benötigte Dienste

2.4.6 MAF-konforme Implementierungen

Es gibt bereits einige Implementierungen der MAF-Spezifikation. Es kann im Rahmen dieser Diplomarbeit nur eine kurze Aufzählung der wichtigsten Implementierungen erfolgen:

1. Aglets von IBM [Agl]
2. Mobile Objects and Agents (MOA) von dem Opengroup Research Institute [MOA]
3. Grasshopper von IKV++ [Gra]

Alle drei Implementierungen verwenden Java als Implementierungssprache. Aglets und MOA verwenden *Remote Method Invocation*, einen RPC-Mechanismus, zur Kommunikation zwischen den Agenten. Grasshopper benutzt die *Common Object Request Broker Architecture (CORBA)* als Kommunikationsinfrastruktur.

2.5 Vergleich von FMA und MAF

Beide Architekturen unterstützen das MbD-Paradigma und sind prinzipiell als Referenzarchitekturen für die zu entwerfende Architektur geeignet. Es können zwei wesentliche Unterschiede der beiden Architekturen identifiziert werden:
Die FMA-Architektur besitzt kein Informationsmodell. Die Management Information ist in den KQML-Nachrichten enthalten, aber nicht in einer öffentlichen

Schnittstelle sichtbar. Dagegen ist bei der MAF-Spezifikation durch die Verwendung von IDL die Management Information an einer Schnittstelle streng typisiert beschrieben.

Aus der Sicht des Anwendungsprogrammierers, der Agenten entwickeln soll, ergibt sich folgendes Bild. In KQML werden alle Nachrichten in ASCII kodiert. Der Programmierer muß die Kodierung und Dekodierung von Nachrichten selbst vornehmen. Es gibt kein Werkzeug, daß ihn dabei unterstützt. Das ergibt einen zusätzlichen Implementierungsaufwand für den Programmierer. Die MAF-Spezifikation beschreibt das Agentensystem mit IDL. Um eine homogene Schnittstellenspezifikation zu erreichen ist es sinnvoll, auch die Agenten mit IDL zu beschreiben. D. h. der Programmierer entwickelt die Agenten auf der Ebene von Objekten und Methoden. Compiler erzeugen für eine gewählte Kommunikationsinfrastruktur die Kodierungs- und Dekodierungsfunktionen. Damit kann sich der Programmierer auf seine wesentliche Aufgabe, die Entwicklung von Agenten konzentrieren.

Kapitel 3

Techniken zur Realisierung

Die weiteren Schritte werden im Vorgehensmodell analysiert. In den folgenden Abschnitten werden die verwendete Objektmodellierungssprache, Middleware und Programmiersprache vorgestellt. Dabei wird eine kurze Einführung in das jeweilige Gebiet gegeben und anschließend unter dem Gesichtspunkt der Eignung für Agentenarchitekturen näher betrachtet.

3.1 Vorgehensmodell

Nachdem nun die Anforderungen definiert sind und zwei verschiedene Architekturen diskutiert wurden, ist das weitere Vorgehen wie folgt:
Es wird eine Methode des Software-Engineering benötigt, die folgende Schritte unterstützt:

- Konzeption der Architektur
- Erstellung des Objektmodells der Architektur
- Implementierung des Objektmodells

Die eingesetzte Software-Engineering-Methode wird im nächsten Abschnitt vorgestellt.

Da bei der Agentenarchitektur sowohl die Agenten, als auch die Agentensysteme sich verteilt im zu managenden Netz befinden, wird als mögliches Framework für die Verteilung CORBA im Abschnitt 3.3 beschrieben.

Auch die einsetzbare Implementierungssprache läßt sich losgelöst von der Architektur behandeln. Es wäre sogar denkbar, für die Laufzeitumgebung und die Agenten unterschiedliche Implementierungssprachen zu verwenden. Java wird als mögliche Implementierungssprache im Abschnitt 3.4 vorgestellt.

3.2 Object Modeling Technique

Bei der ***Object Modeling Technique (OMT)*** [Rum93] handelt es sich um eine objektorientierte Software-Entwicklungsmethode. Dabei kann das zu entwerfende System anhand von drei unterschiedlichen Modellen charakterisiert werden, die im weiteren Verlauf vorgestellt werden.

3.2.1 Objektmodell

Das Objektmodell gibt die statische Struktur des Systems wieder. Dabei wird ein Objekt durch eine Klasse repräsentiert, die wiederum aus Attributen und Methoden besteht. Die Klassen können in verschiedenen Relationen (z. B. Assoziationen) zueinander stehen. Eine spezielle Assoziation ist die Aggregation, oder Enthaltenseins-Relation. Damit wird zum Ausdruck gebracht, daß eine Klasse ein Teil einer anderen Klasse ist. Die Generalisierung (Vererbung) beschreibt eine Relation, die ausdrückt, daß eine Klasse die Verfeinerung einer (Ober)-Klasse ist. Attribute und Methoden werden in die Unterklasse übernommen.

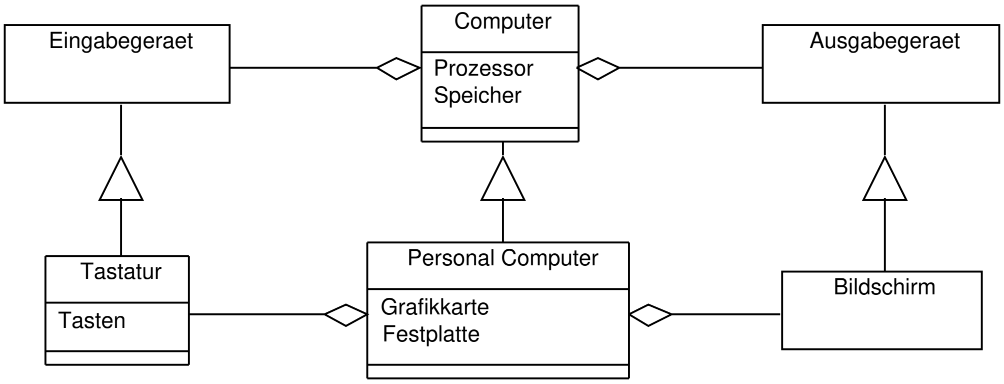

Abbildung 3.1: Beispiel eines Objektmodells in OMT

Im Beispiel der Abbildung 3.1 erbt die Unterklasse `Personal Computer` von der Oberklasse `Computer`. Das wird durch das Dreieck als Symbol, in der Verbindungslinie der beiden Klassen angezeigt. Dabei zeigt die Spitze des Dreiecks immer zur Oberklasse. Die Attribute `Prozessor` und `Speicher` der Oberklasse `Computer` werden an die Unterklasse `Personal Computer` vererbt. Die beiden Verbindungslinien (mit der Raute als Symbol) bei der Klasse `Personal Computer` zeigen an, daß die Klassen `Tastatur` und `Bildschirm` Teil der Klasse `Personal Computer` sind.

3.2.2 Dynamisches Modell

Um die zeitlichen Abläufe in einem System, speziell Zustandsänderungen von Objekten darzustellen, wird das dynamische Modell verwendet. Das dynamische Modell, oder Zustandsdiagramm, ist ein endlicher Automat. Die Ursache für einen Zustandsübergang, oder Transition, ist ein Ereignis. Bei diesem Modellierungsansatz können nun Vererbungshierarchien von Ereignissen oder Zuständen definiert werden.

3.2.3 Funktionales Modell

Beim Funktionalen Modell steht die Datentransformation im Mittelpunkt. Zur Veranschaulichung der Datentransformation werden Datenflußdiagramme erstellt, welche Graphen von Prozessen, Datenflüssen, Datenspeichern und Handlungsobjekten sind. Ein Prozeß wandelt Daten, wobei die Datenflüsse die Daten zum Prozeß bringen. Ein Datenspeicher ist ein Objekt, welches Daten bereithält. Ein Handlungsobjekt erzeugt und verbraucht Daten. Prozesse können weiter aufgefächert werden und müssen auf der untersten Modellierungsebene Methoden entsprechen.

3.2.4 Eignung für den Entwurf Mobiler Agenten Architekturen

OMT ist eine Modellierungsmethode, die so allgemein gehalten ist, daß man damit beim Entwurf eines Systems völlig unabhängig von der später verwendeten Implementierungssprache ist. Durch die drei Teilmodelle kann man die meisten Aspekte eines Systems erfassen und es erhöht sich damit die Robustheit des Entwurfs.

3.2.5 Software through Pictures (StP)

Als Entwicklungswerkzeug zur Erstellung von OMT-Modellen wird Software through Pictures (StP) Version 2.4.2 der Firma Aonix verwendet [Aon96]. Neben der graphischen Erstellung der Modelle kann man in sogenannten *Classtables* den Attributen und Operationen einer Klasse die Schlüsselwörter, wie `public` und `private`, einer Implementierungssprache hinzufügen. StP unterstützt die Generierung von Programm- bzw. Klassenskeletten für verschiedene Spezifikations- und Programmiersprachen.

3.3 CORBA

Die Common Object Request Broker Architecture (CORBA) ist von der Object Management Group (OMG) als Spezifikation herausgegeben worden und liegt inzwischen in der Version 2.2 vom Februar 1998 vor ([COR98]). Hauptziel der Spezifikation ist es, eine von Hardware- und Implementierungssprachen unabhängige, ortstransparente Architektur für verteilte Objekte zu definieren.

3.3.1 Grundlegende Konzepte und Notation

Ein **CORBA-Objekt** ist eine gekapselte Entität, die einem Client Dienste anbietet. Ein Dienst (Service) ist eine Schnittstelle, in der Operationen, die mit der ***Interface Definition Language (IDL)*** deklariert sind, einem Client zum Aufruf bereitgestellt werden. IDL ist eine objektorientierte Schnittstellenbeschreibungssprache. Die IDL-Syntax ist eine Untermenge der C++-Syntax mit zusätzlichen Schlüsselwörten. Außerdem werden alle C++-Präprozessor-Direktiven unterstützt. IDL-Compiler generieren aus einer IDL-Schnittstelle Stubs und Skeletons zu einer Implementierungssprache, z. B. Java, C++, Cobol, Ada etc. Stubs und Skeletons übernehmen das Marshalling[1] von Aufrufen und dienen als Proxy-Objekte für den Client bzw. Server.

Durch die abstrakte IDL wird die Unabhängigkeit von der Implementierungssprache erreicht. Clients und CORBA-Objekte können dadurch in verschiedenen Programmiersprachen implementiert werden.

Die IDL-Schnittstellen können in einer ***Interface Repository***, einer Datenbank für IDL-Schnittstellen, gespeichert werden. Sie ermöglicht vor allem ***Dynamic Method Invocation***, welches der Aufruf einer Operation einer IDL-Schnittstelle ist, der erst zur Laufzeit des Clients erzeugt wird. Ein Client sucht dabei in der *Interface Repository* nach einem, für seine Zwecke geeigneten Interface bzw. Operation, generiert die Parameter und ruft die Operation auf.

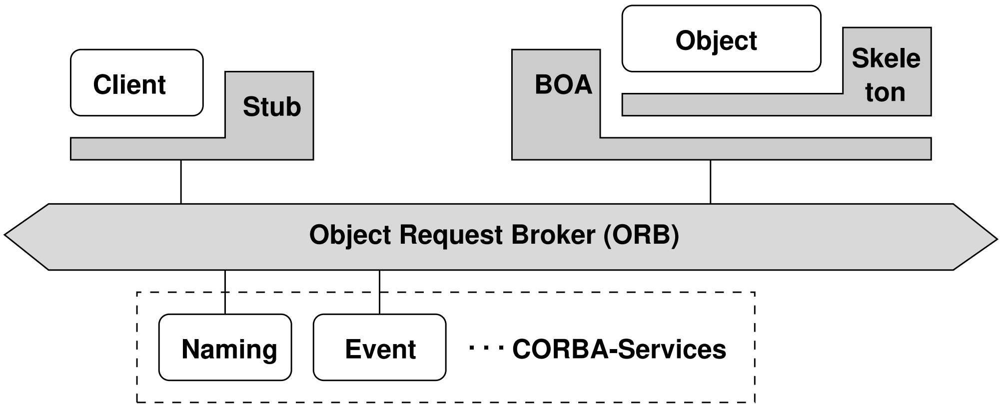

Abbildung 3.2: *Common Object Request Broker Architecture (CORBA 2.2)*

Jedem CORBA-Objekt ist eine eindeutige **Objektreferenz** zugeordnet. Ein Client, muß die Objektreferenz des CORBA-Objekts besitzen um mit diesem

[1]Packen/Entpacken der Operation und deren Parameter in/aus einem Netz-Protokollpaket

kommunizieren zu können. Die Kommunikationsinfrastruktur stellt der ***Object Request Broker (ORB)*** bereit (siehe Abbildung 3.2). Der ORB kann als Objekt-Bus angesehen werden, der die Ortstransparenz der CORBA-Objekte gegenüber den Clients herstellt. Ein CORBA-Objekt wird über einen ***Object Adapter*** an den ORB gebunden. Der Object Adapter stellt die Laufzeitumgebung eines CORBA-Objekts dar. Es leitet die Anfragen an das CORBA-Objekt weiter. Ein Standardadapter, der ***Basic Object Adapter (BOA)*** muß von jedem Hersteller unterstützt werden.

Um ORBs verschiedener Hersteller miteinander verbinden zu können, wurde das ***General Inter-ORB Protocol (GIOP)*** spezifiziert. GIOP ist für verbindungsorientierte Transportprotokolle konzipiert worden. Das ***Internet Inter-ORB Protocol (IIOP)*** spezifiziert, wie GIOP-Nachrichten über ein TCP/IP Netz übermittelt werden. Jeder CORBA 2.2 kompatible ORB muß IIOP unterstützen.

Um die Interoperabilität zwischen ORBs verschiedenen Herstellern zu gewährleisten, ist eine ***Interoperable Object Reference (IOR)*** spezifiziert worden.

CORBA Services sind CORBA-Objekte, welche fundamentale Dienste, wie Naming, Event, Security Service, etc. bereitstellen. Diese Services werden von Applikationsprogrammierern genutzt.

Drei CORBA Services, die in Hinblick auf Agentenarchitekturen interessant sind, werden nun vorgestellt.

3.3.2 Naming Service

Ein *Naming Service* [COR97] ist ein Verzeichnisdienst, mit dem CORBA-Objekte durch Namen identifizieren werden können. Die Namen werden im allgemeinen so gewählt, daß sie für den Menschen gut lesbar sind und eine intuitive Benutzung ermöglichen.

Der CORBA Naming Service bildet eine Assoziation zwischen einem CORBA-Objekt und einem Namen. Diese Assoziation wird Namensbindung (***Name Binding***) genannt. Ein ***Context*** ist ein Behälter für Namensbindungen. Eine Name wird immer relativ zu einem Context gesehen. Innerhalb eines Context ist jeder Name eindeutig. Da ein Context auch ein CORBA-Objekte ist, kann man einem Context einen Namen zuordnen. Damit kann ein *Name Graph* aufgebaut werden. Dabei wird zwischen einem

- *Simple Name*: ein Name, aus genau einer Komponente und
- *Compound Name*: ein Name, der aus mehreren Komponenten besteht

unterschieden. Eine einzelne Komponente besteht aus der ***Identity*** und dem ***Kind***, in Anlehnung an Dateinamenkonventionen der gängigen Betriebssysteme. So ist bei einer Datei 'readme.txt' der erste Teil 'readme' die *Identity* und 'txt' der *Kind*.

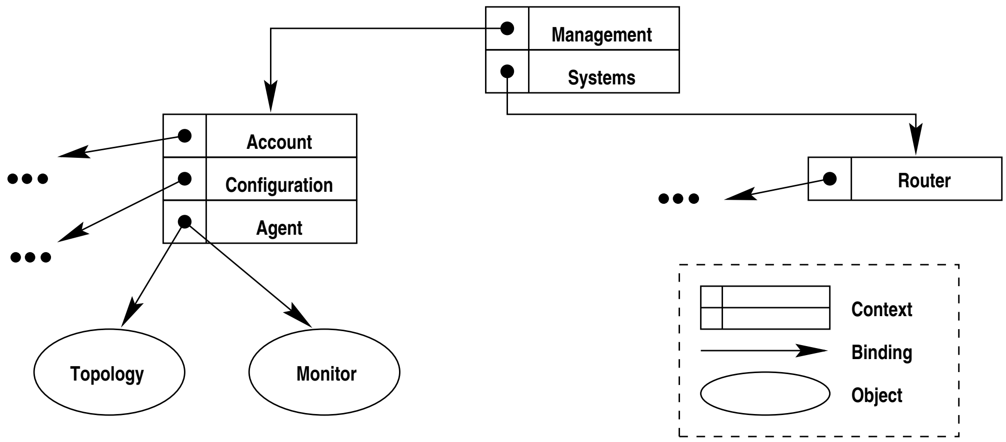

Abbildung 3.3: Beispiel für einen *Name Graph* des CORBA Naming Service

In Abbildung 3.3 ist ein *Name Graph* abgebildet. Ein *Compound Name* ist z. B. `Management/Agent/Monitor`, mit dem Schrägstrich als Trennsymbol, wobei die letzte Komponente `Monitor` ein *Simple Name* ist und die Bindung vom Context `Agent` zu einem CORBA-Objekt beschreibt. Die *Kinds* sind in diesem Beispiel nicht berücksichtigt.

3.3.3 Event Service

Der Event Service [COR97] stellt einen asynchronen, dezentralen Kommunikationsmechanismus zur Verfügung. Es werden zwei Rollen definiert:
Supplier (Erzeuger) sind Objekte, die Events produzieren und ***Consumer*** (Verbraucher) sind Objekte, die Events verarbeiten. Die Entkopplung von *Supplier* und *Consumer* wird durch einen Event Channel (Ereigniskanal) realisiert. An einen Event Channel können sich beliebig viele Supplier und Consumer anbinden.

Das Handling der Events kann durch zwei verschiedene Modelle beschrieben werden. Im **Pushmodell**, sendet der *Supplier* die von ihm erzeugten Events sofort an den Event Channel. Der Event Channel sendet automatisch den Event an die angeschlossenen *Consumer*. Im Gegensatz dazu nimmt beim **Pullmodell** der *Consumer* die aktive Rolle ein. Er fragt beim Event Channel aktiv nach Events, woraufhin der Event Channel bei den *Suppliern* die bereits erzeugten Events abholt. Eine Mischung der beiden Modelle ist auch möglich, z. B. ein *Push Supplier* und ein *Pull Consumer*.

Abbildung 3.4 zeigt ein Beispiel mit zwei Event Channel. *Supplier* und *Consumer* kommunizieren über Event Channels miteinander. Dabei kann ein *Supplier* wie auch ein *Consumer* mit mehreren Event Channels verbunden sein.

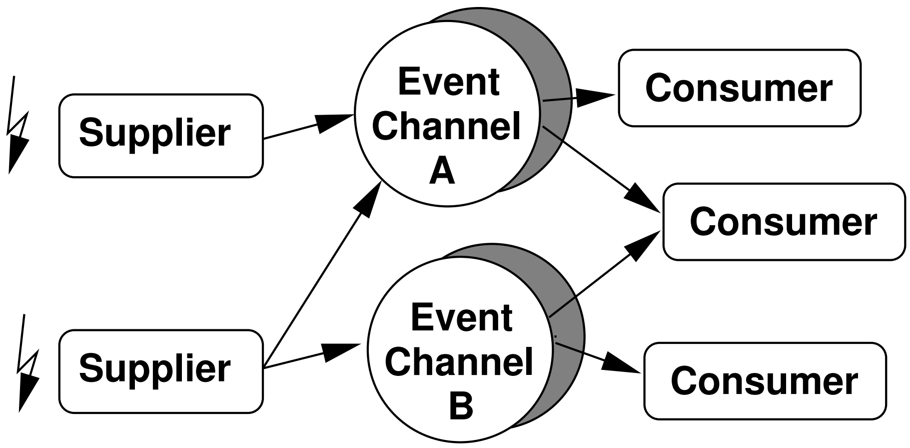

Abbildung 3.4: Pushmodell mit zwei Event Channel des CORBA Event Service

Es existieren zwei Typen von Events:

- ***Any***: Container für einen beliebigen Typ
- ***Typed Event***: der Event besitzt einen fest zugeordneten Typ

Der größte Unterschied zwischen den beiden Typen ist, daß sich der *Consumer* beim Event Channel für einen bestimmten *Typed Event* registrieren kann. Der *Consumer* erhält dann nur Events, die diesem *Typed Event* entsprechen. Diesen Mechanismus gibt es für den Typ *Any* nicht.

3.3.4 Notification Service

Der **Notification Service** [COR97] ist eine Erweiterung des Event Service. Der Notification Service bietet einen Registrierungsmechanismus, mit dem sich ein *Consumer* nicht nur für bestimmte Event-Typen registrieren kann, sondern auch über Filter den Inhalt der Events, an denen er interessiert ist, näher bestimmen. Außerdem kann man Dienstgüteparamter (QoS-Parameter), z. B. Priorität eines Events oder Timeout eines Events, festlegen. Um die neuen Anforderungen Filterung und QoS-Parameter effizient einbringen zu können, wird ein neuer Eventtyp, der ***Structured Event***, eingeführt. Der Aufbau eines *Structured Event* ist fest vorgegeben und besteht aus dem:

- *Event Header*: der vor allem aus Eventtyp und -name beschreibt
- *Event Body*: Schlüssel-Werte-Paare, welche die Filter und deren Werte beschreiben

Um auch feine Filter aufbauen zu können, wird eine *Constraint*-Sprache verwendet, mit der sich komplexe Filterregeln definieren lassen. Als Basis für die *Constraint*-Sprache wird die Spezifikation aus der *Trader Object Service Specification* [COR97] herangezogen, und in einigen Punkten erweitert.

3.3.5 Eignung für Agentenarchitektur

CORBA kann grundsätzlich als Kommunikationsinfrastruktur einer Agentenarchitektur verwendet werden.

Die Hardwareunabhängigkeit ist wegen der Heterogenität des zu managenden Rechnernetzes eine Grundvoraussetzung. Da jeder Agent normalerweise Dienste über eine Schnittstelle anbietet, bietet sich IDL als Beschreibungssprache an. Ebenso bietet sich der Agent als CORBA-Objekt an. Durch IDL wird der Dienst exakt und streng typisiert definiert. CORBA unterstützt die Verteilung von CORBA-Objekten durch den ortstransparenten ORB gut. CORBA-Objekte unterliegen keinen vordefinierten Rollenzuweisungen. Die symmetrische Kommunikation wird vom ORB, die asynchrone Kommunikation wird vom Event bzw. Notification Service unterstützt. Die zahlreichen Services, vor allem aber Naming, Event und Notification Service bieten zusätzlich zum ORB Dienste, die in einer Agentenarchitektur benötigt werden.

Durch den CORBA-Standard ist man auch nicht auf einen ORB eines bestimmten Herstellers festgelegt und kann somit ohne jeglichen Implementierungsaufwand den ORB eines anderen Herstellers verwenden. Mögliche Gründe für einen Wechsel wären z. B. bessere Leistung, oder die Unterstützung von Sicherheitsprotokollen (z. B. Secure Socket Layer (SSL)).

CORBA ist ein Standard, der von der OMG, einer Gruppe, der über 800 Firmen und Institutionen angehören, entworfen wurde, und somit von einer großen Akzeptanz und Unterstützung ausgegangen werden kann. Dafür spricht auch die Integrierung des ORBs in Java ab Version 1.2 des Java Development Kit, sowie die Integrierung in den Webbrowser von Netscape.

3.3.6 Visibroker for Java

Visibroker for Java ist ein von der Firma Inprise implementierter CORBA 2.0 konformer ORB. Er bietet eine Entwicklungsumgebung, die aus folgenden Teilen besteht:

- `idl2java`: Ein Compiler, der aus einem IDL-Schnittstelle die zugehörigen Stubs und Skeletons generiert.
- java2iiop: Einen Compiler, der aus **Java**-Interfaces die zugehörigen IDL-Stubs und Skeletons generiert.

  Das ermöglicht, daß man sich bei der gesamten Softwareentwicklung alleine auf Java abstützten kann.
- `osagent`: der sog. *Smart Agent*, ein ausführbares Programm, enthält einen Lokalisierungsdienst und einen nicht CORBA 2.0 konformen Verzeichnisdienst. Der `osagent` muß auf einem Host im Netz ausgeführt werden. Die Clients erreichen den `osagent` erstmalig über einen UDP-Broadcast.
- `osfind`: Listet alle instanziierten CORBA-Objekte an einem ORB auf.

3.4 Java

Java ist eine objektorientierte Programmiersprache, die von Sun Microsystems 1995 veröffentlicht wurde. Eines der Ziele der Entwicklung von Java war die **Hardwareunabhängigkeit** kompilierter Programme. Java definiert neben der Syntax und Semantik der Sprache, eine umfangreiche Bibliothek, die im weiteren Verlauf des Abschnitts vorgestellt wird.

Der Java Compiler erzeugt **keinen** Maschinencode sondern einen portablen Bytecode, welcher von der *Java Virtual Machine (JVM)* interpretiert wird. Damit kann ein einmal übersetztes Programm auf verschiedenen Plattformen ausgeführt werden, auf der eine JVM gestartet werden kann.

Adreßarithmetik ist in Java, im Gegensatz zu C oder C++, nicht möglich. Damit ist eine böswillige Manipulation von z. B. als 'private' gekennzeichneten Attributen einer Klasse, von außerhalb der Klasse, auf programmiersprachlicher Ebene nicht möglich. Auf der Ebene des zu interpretierenden Bytecode überprüft innerhalb der JVM der *Bytecode Verifier* jeden Speicherzugriff auf seine Korrektheit.

Ein **Applet** ist ein Javaprogramm, welches normalerweise von einem Webserver über das Rechnernetz auf eine Client-Maschine geladen wird. Das Applet wird innerhalb des Browsers in einer *Sandbox*[2] ausgeführt. Ein Applet ist vor allem als GUI geeignet.

Java unterstützt die Serialisierung von Klassen sehr komfortabel. Jede Klasse kann prinzipiell serialisiert werden, indem dies durch einen Indikator in der Signatur der Klasse anzeigt wird. Die Serialisierung erfolgt automatisch und rekursiv. Die aktuellen Werte der Attribute der Klasse werden dabei berücksichtigt. Nur einige Klassen, wie z. B. `java.lang.Thread` sind nicht serialisierbar und müssen deshalb gesondert bearbeitet werden.

Wichtige Klassen der standardisierten Bibliothek:

- `java.lang.Thread`: Erzeugung und Steuerung von Threads
- `java.lang.SecurityManager`: Überwachung aller sicherheitsrelevanten Bereiche, wie Manipulation von Threads, Sockets, Dateisystem etc.
- `java.lang.Classloader`: Laden und Instanziierung von Klassen während der Laufzeit
- Klassen des `java.net`-Pakets: Klassen, zur Unterstützung von Netzverbindungen, wie Sockets oder HTTP-Verbindungen

Abbildung 3.5 zeigt die Ausführungsumgebung einer Java-Applikation, wobei innerhalb des Programms drei voneinander unabhängige Threads ausgeführt werden.

[2]Einer JVM, die dem Applet z. B. den Zugriff auf Systemressourcen verbietet

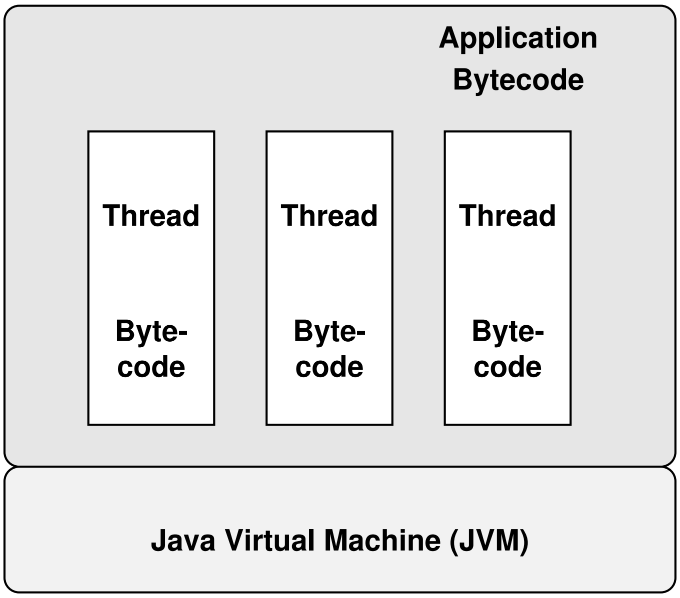

Abbildung 3.5: Ausführungsumgebung für Java Bytecode

3.4.1 Eignung für Agentenarchitekturen

Die vorgestellten Eigenschaften heben Java als Implementierungssprache für eine Agentenarchitektur hervor. Sowohl für die Agenten, als auch für die Laufzeitumgebung ist Java geeignet. Eigenschaften von Java, von der Agenten **und** Laufzeitumgebung profitieren:

- Java ist objektorientiert und unterstützt den modularen Aufbau der Laufzeitumgebung ebenso, wie die Entwicklung eines generischen Basisagenten.
- Durch die Plattformunabhängigkeit und die Unterstützung vieler unterschiedlicher Betriebssysteme ist der Einsatz in einem heterogenen Rechnernetz sinnvoll möglich.
- Der *Bytecode Verifier* verhindert die unautorisierte, gegenseitige Manipulation von Laufzeitumgebung und Agenten.

Eigenschaften von Java, welche die Laufzeitumgebung unterstützt:

- Der *Classloader* ermöglicht das Laden von Klassen und damit Agenten zur Laufzeit.
- Der *SecurityManager* kann gegenüber dem Agenten vor allem Sicherheitsanforderungen bezüglich der lokalen Systemressourcen durchsetzen.

Eigenschaften von Java, welche den Agenten unterstützt:

- Java ist eine interpretierende Sprache und deshalb besonders für Mobile Agenten geeignet, da einfach der Bytecode geladen und ausgeführt wird.

Ansonsten müßte bei jedem Plattformwechsel eine Neukompilierung des Agenten vorgenommen werden.

- Die Serialisierung wird durch Java sehr gut unterstützt, da durch einen einfachen Indikator die meisten Klassen serialisierbar werden.
- Ein Applet ist als graphische Benutzeroberfläche geeignet, da es genauso wie ein Mobiler Agent dynamisch und verteilt geladen und ablaufen kann.

Applets sind nicht als Mobile Agenten geeignet, da keine Attributwerte bei einem Transfer übertragen werden können.

3.5 Weitere Realisierungstechniken

Es gibt neben den vorgestellten Middleware und Implementierungssprache auch noch andere Techniken, auf die kurz verwiesen wird.

3.5.1 Kommunikationsinfrastruktur

Zwei Beispiele für Middlewares sind:

- Distributed Component Object Model Protocol (DCOM) von Microsoft (vgl. [DCO], [Ses97])
- Distributed Computing Enviroment (DCE) der Open Software Foundation (vgl. [DCE], [OSF93]): ein Menge von Produkten unterschiedlicher Hersteller, die in einem Standard integriert wurden: *Remote Procedure Call (RPC), Cell* und *Global Directory Services (CDS* und *GDS), the Security Service, DCE Threads, Distributed Time Service (DTS)* und *Distributed File Service (DFS)*

3.5.2 Implementierungssprachen

Zur Implementierung, vor allem von Agenten, sind Skriptsprachen wegen ihrer Portabilität interessant. Beispiele sind Perl [SCP96] und Tcl [ZP98].

Speziell für Mobile Agenten wurde die Sprache Telescript [tel96] von General Magic entwickelt.

Kapitel 4

Konzeption der Architektur

In diesem Kapitel werden die Erkenntnisse und Folgerungen aus den vorherigen Kapiteln in einem konzeptionellen Entwurf der Architektur eingebracht. Dabei sind die Anforderungsanalyse (Abschnitt 2.2) und die MAF-Spezifikation (Abschnitt 2.4) von besonderer Bedeutung. Der FMA-Prototyp wurde wegen der beschriebenen Mängel (Unterabschnitt 2.3.2) nicht berücksichtigt. Deswegen müssen nun die für die FMA-Realisierung entwickelten Agenten auf die neue Architektur, die im folgeneden beschrieben wird, portiert werden. Die wichtigsten Festlegungen, die in diesem Kapitel erfolgen, sind:

- Implementierungssprache
- Agenten
- Laufzeitumgebung der Agenten
- Kommunikationsinfrastruktur
- Dienste der Kommunikationsinfrastruktur

4.1 Implementierungssprache

Als Implementierungssprache wird Java verwendet. Die wichtigsten Vorzüge werden nun kurz zusammengefaßt:

- Hardwareunabhängigkeit des Bytecodes: ist zum einen für heterogene Netzkomponenten wichtig und zum anderen ermöglicht es erst die Mobilität der Agenten
- Objektorientiertheit der Sprache: ermöglicht eine strukturierte, modulare Implementierung
- Optimale Unterstützung der Serialisierung: reduziert den Implementierungsaufwand dieses Punktes erheblich

- Threads: ermöglichen den nebenläufigen Ablauf der Agenten innerhalb eines Prozesses
- Applets: ermöglichen eine einfache Implementierung der GUI, sowie die Einbettung in eine HTML-Seite

Zur Implementierung wird der Java Development Kit (JDK) Version 1.1.6 von Sun Microsystems verwendet.

4.2 Kommunikationsinfrastruktur

CORBA bietet als Kommunikationsinfrastruktur die meisten Eigenschaften der Anforderungsanalyse:

- Standard, der von vielen Firmen unterstützt wird
- Ortstransparente Architektur für verteilte Objekte
- Unterstützung der symmetrischen und asymmetrischen Kommunikation
- Hardwareunabhängig
- Unterstützt Java als Implementierungssprache
- Clients können in einer anderen, von CORBA unterstützten Implementierungssprache programmiert sein
- Objekte besitzen nach außen eine feste Schnittstelle
- Naming, Event und Notification Service sind im Standard enthalten

Die zwei IDL-Schnittstellen der MAF-Spezifikation werden nach CORBA umgesetzt. Daraus folgt, daß das *MAFAgentSystem* und der *MAFFinder* CORBA-Objekte werden.

Es existieren verschiedene Implementierungen von CORBA ORBs, wobei der Visibroker for Java 3.0 verwendet wird, der auch den Naming und Event Service implementiert.

4.3 Entscheidung für MAF-Spezifikation

Ein wichtiger Punkt für die Wahl der MAF-Spezifikation zur Realisierung ist, daß mit dieser Spezifikation ein **Standard** festgelegt wurde, der die Interoperabilität, in gewissen Grenzen, zwischen verschiedenen Implementierungen des Standards, ermöglicht. Die MAF-Spezifikation deckt den größten Bereich der Anforderungsanalyse in Abschnitt 2.2 ab. Durch die Entscheidung für die MAF-Spezifikation ist der Agent und die Laufzeitumgebung in großen Teilen festgelegt.

4.3.1 Agent

Die Anforderungen an den Agenten, wie sie in der Anforderungsanalyse aufgestellt wurden, entspricht dem Agentenkonzept der MAF-Spezifikation, insbesonders die Mobilität der Agenten, womit das *Management by Delegation*-Paradigma erfüllt ist. Die MAF-Spezifikation definiert nicht die in der Anforderungsanalyse verlangte Schnittstelle nach außen. Diese Schnittstelle wird mit IDL spezifiziert. Da die Kommunikationsinfrastruktur auf CORBA festgelegt ist, liegt es natürlich nahe, daß ein Agent ein CORBA-Objekt ist. Damit ist auch die Forderung nach Kommunikation erfüllt, denn CORBA-Objekte können über den ORB miteinander kommunizieren.

4.3.2 Laufzeitumgebung

Auch für die Laufzeitumgebung der Anforderungsanalyse ist das Pendant das Agentensystem der MAF-Spezifikation. Das Agentensystem verwaltet den Agenten. Es unterstützt die Mobilität der Agenten und setzt Sicherheitsanforderungen durch.

Eine interne API zwischen Agent und Agentensystem ist in der MAF-Spezifikation nicht definiert worden. Damit ein Agent seinem eigenen Agentensystem z. B. den Wunsch einer Migration mitteilen kann, ist eine interne Schnittstelle nötig. Diese interne Schnittstelle muß zusätzlich in die Realisierung aufgenommen werden.

4.3.3 Objektmodell der Konzeption

Aus den vorgestellten konzeptionellen Entscheidungen kann ein erster Entwurf des Objektmodells erfolgen. In Abbildung 4.1 sind im oberen Teil die

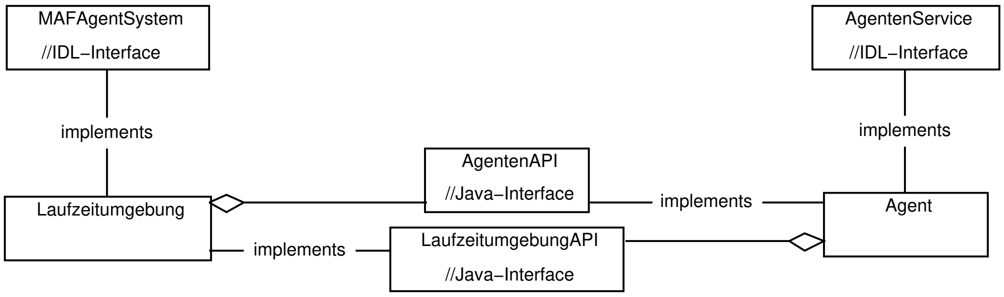

Abbildung 4.1: Objektmodell der Konzeption

IDL-Schnittstellen ***MAFAgentSystem*** und ***AgentenService*** zu sehen. Beide IDL-Schnittstellen werden jeweils von einer Klasse implementiert. Die Klassen `Laufzeitumgebung` und `Agent` sind über die Schnittstellen `AgentenAPI` und `LaufzeitumgebungAPI` miteinander verbunden.

4.3.4 Nicht berücksichtigte Konzepte der MAF-Spezifikation

Zur Lokalisierung der Agenten wurde die *Location* eingeführt und ein *MAFFinder* als Verzeichnisdienst. Diese Konzepte werden nicht benötigt, da ein Agent ein CORBA-Objekt ist und daher über seine IOR eindeutig festgelegt ist, bzw. der Agent über den bereits vorhanden CORBA Naming Service lokalisiert werden kann. Außerdem wird dadurch zusätzlicher Implementierungsaufwand eingespart. Auch in der MAF-Spezifikation [GMD97] (S.26f) wird der mögliche Einsatz des CORBA Naming Service diskutiert.

Das Konzept der *Places* wird ebenfalls nicht aus der MAF-Spezifikation übernommen. *Places* sind die Ausführungsumgebungen von Agenten innerhalb eines Agentensystems. Verschiedene *Places* können z. B. verschiedene Zugriffsrechte auf Ressourcen durchsetzen. Anstatt der *Places* wird das Konzept der *Access Control List (ACL)* eingeführt. Jeder Agent führt eine ACL mit sich, die beschreibt, welche Rechte ein Agent besitzt. Mit einer ACL kann man feingranularer und individueller abgestimmt Rechte an Agenten vergeben, als bei dem Konzept der *Places*.

Eine *Region*, die Gruppierung von Agentensystemen derselben *Authority*, wird ebenfalls nicht implementiert. Denn eine Lastverteilung wie sie als Grund in der MAF-Spezifikation angegeben ist, ist nicht so bedeutend, da viele Agenten genau auf einem bestimmten Host und somit auf einem bestimmten Agentensystem ablaufen müssen.

In der MAF-Spezifikation teilt ein Agent, der transferiert werden will, seinem Agentensystem die QoS-Anforderungen, die er an die Kommunikationsinfrastruktur stellt, mit. Weiter wird in der Spezifikation nicht auf die QoS-Parameter eingegangen. Mögliche QoS-Parameter wären z. B. der minimale Durchsatz der Übertragung, oder die Übertragungssicherheit des Kanals. Die QoS-Parameter haben mit der eigentlichen Aufgabe dieser Arbeit nur am Rande zu tun und bleiben in der Implementierung deshalb unberücksichtigt.

Alle Sicherheitsaspekte, auf die in der MAF-Spezifikation eingegangen werden, können in dieser Diplomarbeit nicht behandelt werden, da der Rahmen sonst gesprengt würde. Einige Sicherheitsaspekte werden aber in einem Fortgeschrittenenpraktikum von Robert Zeilhofer implementiert.

4.4 Graphische Benutzerschnittstelle (GUI)

Jeder Agent hat zur graphischen Benutzerführung eine Homepage, eine Datei in *Hypertext Markup Language (HTML)*-Format, auf die der Benutzer (Administrator) über eine URL zugreifen kann.

Um eine komfortable Benutzerführung zu erreichen, soll eine Abbildung vom Agentensystem und den darauf ablaufenden Agenten, auf die Bedienoberfläche stattfinden. Diese Abbildung kann am einfachsten unterstützt werden, wenn ein Webserver in jedem Agentensystem integriert ist. Der Webserver ist als Agent realisiert. Der Webserver-Agent muß über eine interne API zum Agentensystem die URLs aller Agenten, die sich auf dem Agentensystem befinden, bekommen.

Die Einstiegsseite des Webserver-Agent ist damit die Homepage des Agentensystems, auf der sich Links zu allen auf dem Agentensystem laufenden Agenten befinden. Es ist somit eine direkte Relation von laufenden Agenten zu vorhandenen URLs auf der Webpage des Webserver-Agenten.

Abbildung 4.2 beschreibt die durchgeführten Aktionen, bis ein Administrator die Homepage eines Agenten (hier IPRouting-Agent) bekommt. Zunächst fordert der Webbrowser des Administrators über das *Hypertext Transfer Protocol (HTTP)* die Homepage des Webserver an (Schritt (1)). Die Informationen für

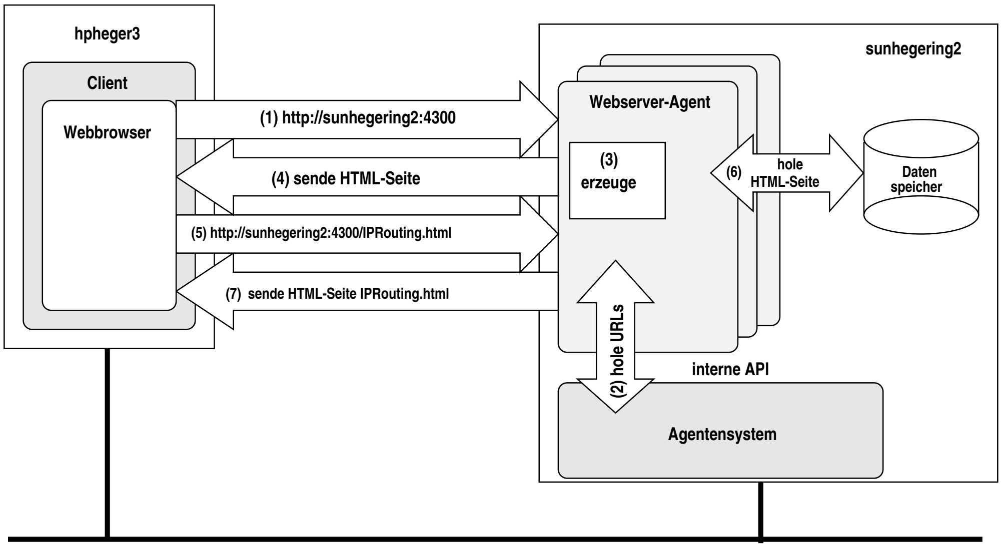

Abbildung 4.2: Laden der Homepage eines Agenten

diese Anfrage sind auf zwei möglichen Wegen zu beschaffen:

- der Administrator hat das Agentensystem erzeugt und kennt deshalb den Rechnernamen ('sunhegering2') und die Portnummer ist allgemein bekannt ('4300'), oder
- der Webserver-Agent stellt über seine IDL-Schnittstelle eine Operation bereit, die diese URL liefert.

Der Webserver-Agent bekommt über einen internen API-Aufruf alle URLs der auf dem Agentensystem ablaufenden Agenten (Schritt (2)). Nun erzeugt der Webserver-Agent dynamisch die HTML-Seite (Schritt (3)), in der die URLs der Agenten als Links angegeben sind, und sendet diese HTML-Seite zum Webbrowser (Schritt (4)).

Der Administrator wählt einen Link aus und der Webbrowser fragt die URL des IPRouting-Agenten nach (Schritt (5)). Der Webserver-Agent liest die HTML-Seite aus seinem Datenspeicher (Schritt (6)) und sendet sie zum Webbrowser des Administrators (Schritt (7)). Die gesendete HTML-Seite wird in der Regel ein Applet enthalten, welches die eigentliche graphische Benutzerschnittstelle zum Agenten darstellt. Der Verbindungsaufbau von Applet zu dem zugehörigen Agenten wird in Unterabschnitt 6.4.3 erläutert.

4.5 Dienste

Die in der Anforderungsanalyse definierten Dienste werden in den folgenden Unterabschnitten vorgestellt, wobei ein Gruppierungsdienst nicht realisiert wird (vgl. Abschnitt 4.8).

4.5.1 Verzeichnisdienst

Der Verzeichnisdienst der Anforderungsanalyse wird vom CORBA Naming Service übernommen. Er bietet zum einen die eindeutige Identifizierung durch einen Namen und zum anderen die Lokalisierung durch die CORBA-Objektreferenz. Es kann prinzipiell eine gewisse Gruppierung von Agenten durch den Naming Service vorgenommen werden. Allerdings sind keine gruppenbezogenen Kommunikationsmechanismen vorhanden.

Es wird die Implementierung des Naming Services von Visibroker for Java 3.0 verwendet.

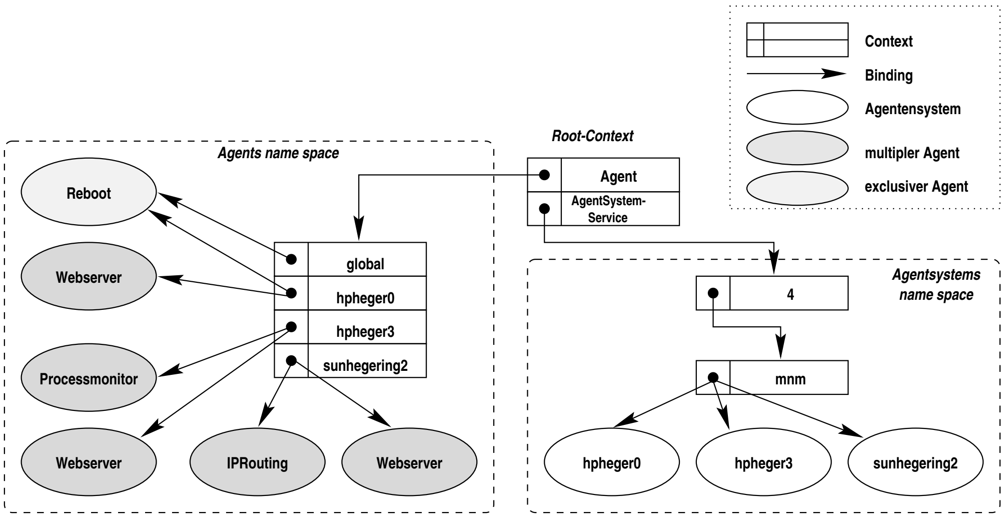

Abbildung 4.3: *Name Graph* der Architektur

Bei der Namensvergabe steht die Eindeutigkeit, sowie die Einfachheit und Intuitivität der Namensvergabe im Vordergrund.

Namensgebung für Agentensysteme

Die MAF-Spezifikation definiert, wie ein eindeutiger *Name* strukturiert ist. Er besteht aus dem *Agent System Type*, der *Authority* und der *Identity*. Diese drei Komponenten bilden den *Compound Name* des Agentensystems. Deshalb wird in der Abbildung 4.3 pro Komponente ein *Naming Context* aufgebaut. Die OMG übernimmt die autorisierte Vergabe des *Agent System Type*. In dem Beispiel wird der *Agent System Type* willkürlich auf die erste freie Nummer '4' festgelegt. Als *Authority* kann eine einzelne Person oder eine Organisation fungieren und wird hier auf 'mnm' festgelegt. Die *Identity* besteht immer aus dem Rechnernamen des Hosts, auf dem das Agentensystem gestartet wird. Die *Identity* wurde so gewählt, da es für viele Agenten notwendig ist, auf einem bestimmten Host gestartet oder transferiert zu werden.

Um die CORBA-Objektreferenz des Agentensystems, der sich auf dem Host 'hpheger0' befindet, zu bekommen, muß man, ausgehend vom Root Context, eine Anfrage an den Naming Service mit dem *Compound Name* 'AgentSystem-Service/4/mnm/hpheger0' stellen. Es wird pro Host höchstens ein Agentensystem erlaubt. Damit ist die oben vorgeschlagene Namensgebung eindeutig und intuitiv für die Benutzer nachvollziehbar. Außerdem ist damit die Portnummer des Webserver-Agenten auf jedem Agentensystem gleich.

Namensgebung für Agenten

Die Namensgebung für Agenten ist problematischer, da sich drei teilweise gegensätzliche Problematiken ergeben:

1. Multiple Agenten: Agenten, von denen sich mehrere Instanzen im Netz befinden dürfen
2. Exklusive Agenten: Agenten, von denen höchstens eine Instanz sich im Netz befinden darf, da sie z. B. eine Ressource exklusiv beanspruchen müssen
3. Um den Administrator, sowie dem Programmierer ein leichtes Identifizieren der Agenten zu ermöglichen, soll zusätzlich gelten, daß Multiple Agenten, z. B. der Webserver-Agent, immer unter dem gleichen Namen auf jeden Agentensystem ansprechbar ist

Um die Anforderungen der Exklusiven Agenten durchzusetzen, ist der Verzeichnisdienst die einzig mögliche Alternative, da es die einzige Komponente in der Agentenarchitektur ist, die von ihrer logischen Struktur her, zentralisiert ist.

Aus der dritten Forderung folgt, daß es pro Agentensystem nur eine Instanz eines bestimmten Agenten geben darf. Diese Beschränkung muß für die Einfachheit der Namensvergabe in Kauf genommen werden.

Es wird eine **feste** Zuordnung von der *Identity* zur einer Klasse getroffen. Das ist die Grundvoraussetzung um überhaupt die Konfliktfälle für Exklusive Agenten erkennen zu können und gleiche Namen für Agenten desselben Typs durchgesetzt werden können. Die Bestandteile *Authority* und *Agent System Type* müssen damit beim Eintrag in den Verzeichnisdienst unberücksichtigt bleiben.

Der eindeutige Namen für Agenten ist nun erreicht. Jetzt müssen diese Namen geeignet in den *name graph* eingehängt werden. Dabei müssen die Agenten unterschiedlich behandelt werden, siehe Abbildung 4.3:

- Multiple Agenten: hier muß erreicht werden, daß mehrere Agenten in den *Name Graph* eingehängt werden können. Dazu wird für jedes Agentensystem ein zusätzlicher Context im *Agent Name Space* eingefügt (vgl. Abb. 4.3), hinter dem diese Agenten eingehängt werden. Als Beispiel ist der Webserver-Agent zu nennen, der auf dem Agentensystem 'hpheger0' läuft. Der *Compound Name* des Agenten vom Root Context aus lautet: 'Agent/hpheger0/Webserver'. Diese Strategie für die Namensvergabe hat zur Folge, daß sich bei Mobilen Agenten, die das Agentsystem wechseln, der Name ändert.

- Exklusive Agenten: diese werden zweimal in den *Name Graph* eingehängt. Zum einen wie jeder Multiple Agent unter dem Context seines Agentensystems und zum anderen, um die Einmaligkeit des Agenten zu garantieren, wird er unter dem zentralen Context 'global' eingehängt. Als Beispiel dient dazu der fiktive Reboot-Agent. Die beiden *Compound Names* des Agenten vom Root Context aus lauten:
'Agent/hpheger0/Reboot' und 'Agent/global/Reboot'.

Um den korrekten Aufbau des *Name Graph* und das richtige Einhängen der Agenten zu gewährleisten, werden diese Aufgaben ausschließlich vom Agentensystem vorgenommen. Sollte z. B.der Versuch unternommen werden, eine zweite Instanz eines Exklusiven Agenten auf einem Agentensystem zu erzeugen, so überprüft das Agentensystem, ob bereits ein entsprechender Eintrag im Naming Service, unter 'Agent/global' gemacht wurde. Da dies der Fall ist, wird die Erzeugung des Agenten abgelehnt.

Die Konfliktsituation bei exklusiven Ressourcen für Agenten ist hier nur für einen Spezialfall gelöst worden. Der allgemeine Fall ist, daß Agenten verschiedenen Typs exklusiv eine Ressource beanspruchen wollen. Dieses Problem ist mit dem Verzeichnisdienst nicht lösbar. Einen möglichen Lösunganansatz würden Policies bieten, in denen jeder Agent beschreibt, welche Voraussetzungen für seine Erzeugung erfüllt sein müssen, um seine Aufgabe korrekt zu erfüllen. Die Lösung dieses Problems ist nicht Gegenstand dieser Arbeit.

4.5.2 Ereignisdienst

Der Ereignisdienst der Anforderungsanalyse wird durch den Notification Service von CORBA realisiert. Er bietet die geforderte Möglichkeit der Filterung von Ereignissen. Implementierungen des Notification Service sind derzeit nicht erhältlich. Parallel zu dieser Diplomarbeit werden Teile der Spezifikation des

Notification Service in einem Fortgeschrittenenpraktikum von Oliver Maul implementiert, so daß künftig darauf zurückgegriffen werden kann.

Als Provisorium wird deshalb der CORBA Event Service als Ereignisdienst verwendet. Damit entfallen die Filtermöglichkeiten.

4.6 Schichtenmodelle und Zugriffssichten

Aus den bis jetzt getroffenen Designentscheidungen ergeben sich die nun folgenden Sichteweisen auf die Agentenarchitektur.

4.6.1 Java-Schichtenmodell

Aus den vorigen Abschnitten ergibt sich nun ein Schichtenmodell aus Java-Sicht.

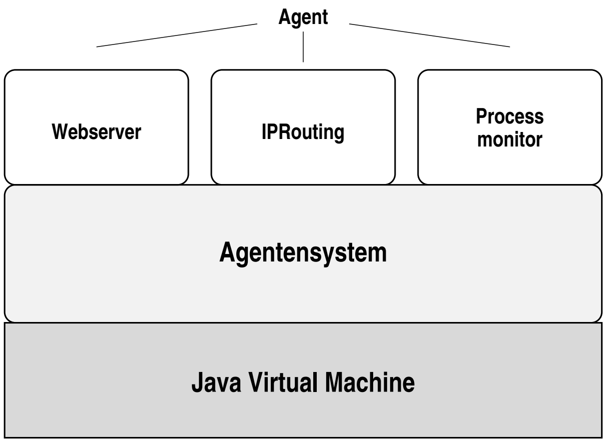

Abbildung 4.4: Softwarearchitektur

Wie in Abbildung 4.4 zu sehen, werden die Agenten auf dem Agentensystem ausgeführt und von diesem verwaltet. Das Agentensystem wiederum wird auf der *Java Virtual Machine* ausgeführt.

4.6.2 CORBA-Zugriffssicht

Abbildung 4.5 zeigt den schematischen Aufbau der Agentenarchitektur aus der CORBA-Zugriffssicht. Die Skeletons und BOAs weisen darauf hin, daß die Komponenten, mit denen sie verbunden sind, CORBA-Objekte sind. Agenten wie Agentensysteme werden in den Naming Service eingetragen und Clients können über den Naming Service auf die jeweiligen Objektreferenzen zugreifen.

In den IDL-Schnittstellen der MAF-Spezifikation fehlt die Möglichkeit, wenn ein Client die CORBA-Objektreferenz eines Agenten besitzt, die zugehörige

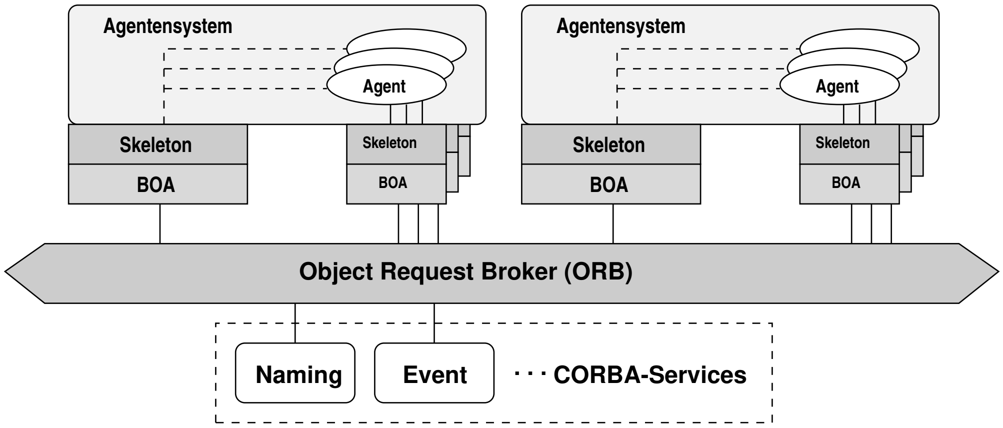

Abbildung 4.5: CORBA-Zugriffssicht auf die Architektur

CORBA-Objektreferenz des Agentensystems herauszufinden. Deshalb bietet jeder Agent eine Operation an, die nicht in der MAF-Spezifikation vorhanden ist, um das zugehörige Agentensystem zu kontaktieren. Eine 'symmetrische' Operation offeriert auch jedes Agentensystem, damit findet eine Entlastung des Naming Service statt. Dieser Umstand wird in der Abbildung durch die gestrichelten Verbindungslinien vom Agentensystem zu den Agenten angezeigt.

4.6.3 Zugriffssicht des Benutzer

In der Abbildung 4.6 ist die Zugriffsicht des Benutzers zu sehen. Von der Homepage des Agentensystems kann der Benutzer über Links auf die Webseite eines Agenten gelangen. Zur Visualisierung dieser Webseiten, wird ein Webbrowser verwendet.

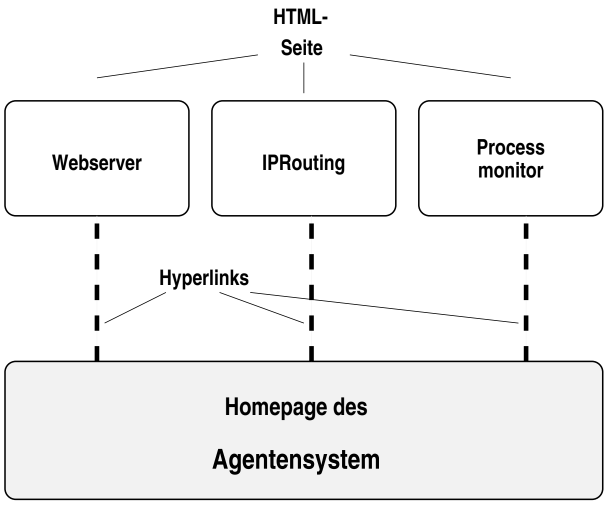

Abbildung 4.6: Zugriffssicht des Benutzers

4.7 Fragen der Anforderungsanalyse

Nun können auf die Fragen der Anforderungsanalyse aus Unterabschnitt 2.2.2 Antworten geben werden:

- Wie erfolgt die technische Realisierung der Serialierung/Deserialisierung? Die Serialierung/Deserialisierung wird von Java übernommen.
- Gibt es bereits standardisierte Schnittstellen zwischen den Komponenten? Die MAF-Spezifikation definiert eine Schnittstelle zwischen Laufzeitumgebungen. Der Naming und Event Service von CORBA bieten als Basisdienste Schnittstellen an.
- Wie wird der Verzeichnisdienst bei der Mobilität von Agenten konsistent gehalten?
  Da die Agentensysteme die Migration der Agenten übernehmen, sind sie auch für die Konsistenz des Verzeichnisdienstes zuständig.
- Wie wird die graphische Benutzeroberfläche bei Mobilen Agenten realisiert?
  Der Webserver des Agentensystems erzeugt bei einer Anfrage die HTML-Seite dynamisch, womit der Mobilität der Agenten Rechnung getragen wird. Applets sind gerade für Mobile Agenten als GUI geeignet, da sie über das Netz ladbar sind.
- Wie erlangt der Administrator eine globale Sicht auf das Managementsystem?
  Eine globale Sicht kann der Administrator nur über den Naming Service erhalten. Ein Agent könnte die Visualisierung des *Name Graphs* und den Zugriff über CORBA auf die dort verzeichneten Agenten und Laufzeitumgebungen bereitstellen.
- Wie können die Sicherheitsanforderungen, die durch die Mobilität der Agenten auftreten, durchgesetzt werden? (Ein Mobiler Agent könnte auch ein Angreifer sein!)
  Die genaue Betrachtung von Sicherheitsanforderungen ist nicht Gegenstand dieser Arbeit. Die MAF-Spezifikation [GMD97] geht aber ausführlich darauf ein. und in Unterabschnitt 2.4.4 werden die wichtigsten Aspekte dargestellt.

4.8 Nicht erfüllte Forderungen der Anforderungsanalyse

Eine Gruppierung von Agenten zur Gruppenkommunikation oder Abstraktion von Agenten nach außen, wird nicht unterstützt. Eine Gruppenkommunikation (Multicast-Kommunikation) ist auf eine Unicast-Kommunikation abbildbar und damit falls nötig nachbildbar. Außerdem ist bis jetzt noch kein konkreter

Anwendungsfall für Gruppenkommunikation vorhanden. Lediglich die organisatorische Gruppierung innerhalb eines Agentensystems wird durch den Aufbau des *Name Graph* im Naming Service unterstützt.

Kapitel 5

Realisierung: Mobile Agent System Architecture (MASA)

Aus dem konzeptionellen Teil der Arbeit muß nun ein Objektmodell entwickelt werden. Um das Objektmodell von anderen Modellen und Architekturen klar zu trennen, hat der Autor den Namen *Mobile Agent System Architecture (MASA)* gewählt.

Im weiteren wird eine Einführung in das Objektmodell gegeben, anschließend werden die Teilmodelle getrennt vorgestellt. Der Schnittstelle zwischen den Teilmodellen, sowie bereits entwickelte Agenten werden in separaten Abschnitten besprochen. Abschließend wird der Lebenszyklus eines Agenten anhand der im Objektmodell betroffenen Schnittstellen und Klassen erläutert, um das Verständnis für das Objektmodell zu vertiefen.

Hinweis zu den Abbildungen in diesem Kapitel:
Da OMT keine graphische Unterscheidung von Schnittstellen und Klassen unterstützt, wird mit entsprechenden Kommentaren in den Abbildungen darauf hingewiesen. Alle IDL-Schnittstellen sind mit `'//IDL-Interface'` und Java-Schnittstellen mit `'//Java-Interface'` gekennzeichnet. Zusätzlich werden alle Schnittstellen und (generierten) Klassen, die aus der MAF-Spezifikation stammen, mit `'//MAF'` beschriftet.

5.1 Einführung in das Objektmodell

In Kapitel 4 sind grundsätzliche Designentscheidungen getroffen worden, die sich auch im Objektmodell wiederspiegeln. So besteht die Architektur für Mobile Agenten aus folgenden Teilmodellen (siehe Abbildung 5.1, vgl. Abbildung 4.1):

- **Agentensystem-Modell**: die Laufzeitumgebung für Agenten, ist auf der linken Hälfte der Abbildung zu sehen (vgl. Abschnitt 5.2)

- **Basisagent-Modell**: die Objekte, die den Basisagenten bilden, sind auf der rechten Hälfte der Abbildung zu sehen (vgl. Abschnitt 5.3)

- **Schnittstelle zwischen den beiden Teilmodellen**: die im mittleren Teil der Abbildung sich befindenden Objekte `Migrate` und `MigrateInfo`, sowie die Objekte `AgentManager` und `Agent` bilden die Schnittstelle

Damit das Agentensystem und die Agenten die Vorteile von CORBA nutzen können (vor allem die symmetrische Kommunikation untereinander) werden sie als CORBA-Objekte realisiert. Beide Teilmodelle besitzen IDL-Schnittstellen, die oberen Teil in Abbildung 5.1 zu sehen sind. D. h., daß jedes Agentensystem und jeder Agent seine Dienste über eine IDL-Schnittstelle anbietet.

Anmerkung:
In den folgenden Abschnitten wird zur einfacheren Darstellung und Erklärung folgender Sachverhalt nicht völlig exakt erläutert:
Java-Klassen implementieren IDL-Schnittstellen nicht direkt, sondern nur im übertragenen Sinne. Aus einer IDL-Schnittstelle wird eine Java-Schnittstelle generiert, und **diese** wird dann von der entsprechenden Java-Klasse implementiert. Im Abschnitt 6.1 wird die exakte Vererbungshierarchie nachgereicht.

Beide Teilmodelle werden ausgehend von den jeweiligen IDL-Schnittstellen vorgestellt.
Hinweise zur Syntax:

- IDL-Schnittstellen beginnen mit einem Großbuchstaben und sind in *`kursivem Typewriter`* geschrieben

- Java-Schnittstellen und -Klassen beginnen mit einem Großbuchstaben und sind in `Typewriter` geschrieben

- Methoden beginnen mit einem Kleinbuchstaben

- Attribute beginnen mit einem *Underscore* '_', gefolgt von einem Kleinbuchstaben

5.2 Agentensystem-Modell

In diesem Abschnitt werden die Schnittstellen und Klassen, die das Agentensystem bilden, erklärt.

Die definierten IDL-Schnittstellen, *`MAFAgentSystem`* und *`AgentSystemService`*, werden durch verschiedene Klassen implementiert. Dabei sind die Klassen `AgentSystem` und `AgentManager` von zentraler Bedeutung. Die Signaturen der Klassen sind in Java-Syntax angegeben.

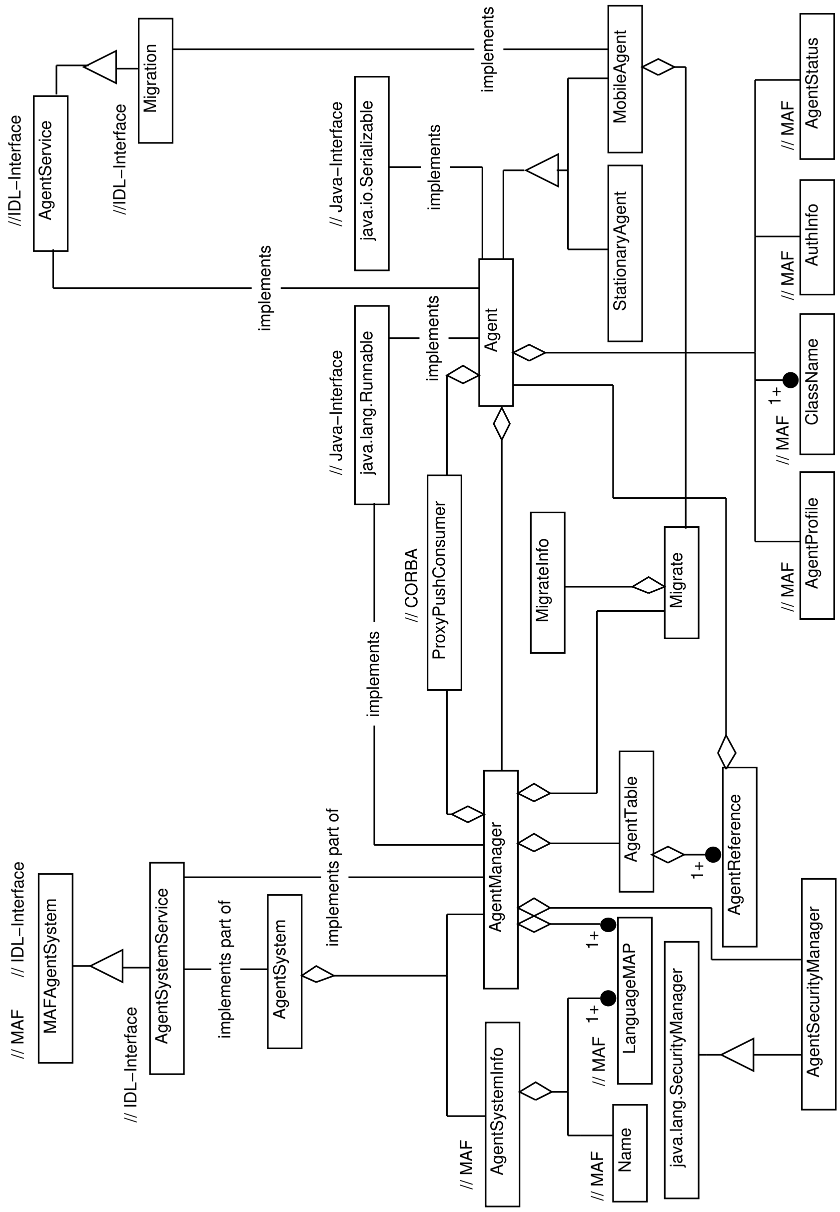

Abbildung 5.1: *Mobile Agent System Architecture* – Gesamtübersicht

5.2.1 IDL-Schnittstelle MAFAgentSystem

Die IDL-Schnittstelle ***`MAFAgentSystem`*** wird in [GMD97] spezifiziert und die Operationen detailliert beschrieben (vgl. auch Unterabschnitt 2.4.3 und 6.3.2).

Die Methoden dieser Schnittstelle lassen sich grob in die folgenden Bereiche einteilen:

- Agentenmanagement:
  - `create_agent(...)`

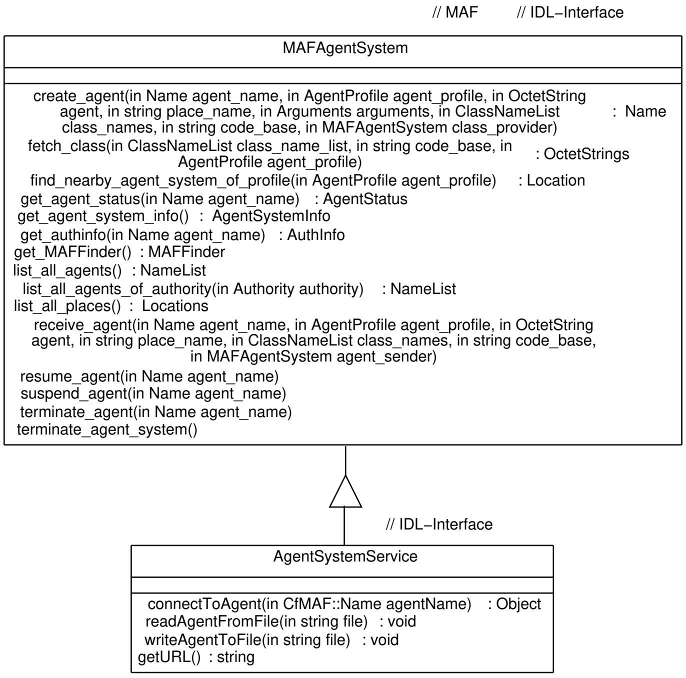

Abbildung 5.2: Teilmodell: Agentensystem – IDL-Schnittstellen

- `receive_agent(...)`
- `terminate_agent()`
- `suspend_agent()`
- `resume_agent()`

- Informationen über das Agentensystem und den darauf laufenden Agenten abfragen:
  - `get_agent_status(...)`
  - `get_authinfo()`
  - `list_all_agents()`
  - `list_all_agents_of_authority()`
  - `get_agent_system_info()`
- Terminierung des Agentensystems: `terminate_agent_system()`
- folgende Operationen werden nicht implementiert (vgl. Abschnitt 4.8):
  - `list_all_places()`
  - `get_MAFFinder()`
  - `find_nearby_agent_system_of_profile()`

5.2.2 IDL-Schnittstelle AgentSystemService

Die MAF-Spezifikation setzt nicht voraus, daß Agenten oder Agentensysteme CORBA-Objekte sind. Deshalb fehlt auch in der IDL-Schnittstelle *MAFAgentSystem* eine Methode, die als Rückgabewert die CORBA-Objektreferenz eines Agenten liefert. Diese Lücke wird von *AgentSystemService* geschlossen. Damit kann eine direkte Verbindung zu einem Agenten auf einem Agentensystem aufgenommen werden, ohne immer CORBA Naming Service kontaktieren zu müssen (siehe dazu auch Unterabschnitt 4.6.2). Die Methode *connectToAgent(...)* liefert die CORBA-Objektreferenz eines Agenten auf dem angesprochenen Agentensystem. Der Agent wird über seinen eindeutigen Namen bestimmt. Falls der Agent nicht vorhanden ist, wird die Ausnahmebehandlung *CfMAF.AgentNotFound* eingeleitet.

Mit Hilfe der Operationen *readAgentFromFile(...)* und *writeAgentToFile(...)* können Agenten aus einer Datei gelesen, oder in eine Datei persistent abgespeichert werden.

Jedes Agentensystem besitzt eine Webseite (vgl. Abschnitt 4.4). Mit der Methode *getURL()* wird der Uniform Resource Locator (URL) des Agentensystems in Form eines Strings zurückgegeben.

5.2.3 Klasse AgentSystem

Die Klasse `AgentSystem` implementiert die Schnittstelle *AgentSystemService* und damit auch *MAFAgentSystem*. Sie ist die wichtigste Klasse des Agentensystems. Damit diese Klasse überschaubar bleibt, werden alle zu implementierenden Methoden, welche die Handhabung der Agenten betreffen, von der Klasse `AgentManager` übernommen (siehe Abbildung 5.3). So wird z. B. die Methode `get_agent_status()` aus der Schnittstelle *MAFAgentSystem* nur 'durchgeschleift',

```
public AgentStatus get_agent_status(Name agent_name)
  throws AgentNotFound {
  return _agentManager.get_agent_status(agent_name);
}
```

wobei die Variable `_agentManager` ein Attribut von `AgentSystem` und vom Typ `AgentManager` ist. Das erklärt auch die Bezeichnung 'implements part of' bei der Assoziation zwischen `AgentSystem` bzw. `AgentManager` und der IDL-Schnittstelle *AgentSystemService*. Die Attribute der Klasse `AgentSystem` sind:

- `_url`: die Homepage des Agentensystems
- `_boa`: Objektreferenz des BOAs, zum Binden des Agentensystems an den ORB.
- `_agentSystemService`: CORBA-Objektreferenz des Agentensystems

- `_initContext`: Objektreferenz des CORBA Naming Service
- `_agentSystemInfo`: in [GMD97] spezifizierte Informationen über das Agentensystem
- `_agentManager`: Verwaltung der Agenten (siehe Unterabschnitt 5.2.4)

`AgentSystem` implementiert die Methode `main()`, ist also das Hauptprogramm der gesamten Implementierung. Pro JVM wird genau eine Instanz der Klasse `AgentSystem` gestartet. Die Durchsetzung dieser Anforderung wird einfach dadurch erreicht, daß der Konstruktor `AgentSystem()` den Sichtbarkeitsindikator 'private' hat und somit von keiner anderen Klasse aufgerufen werden kann. `AgentSystem` dient als Laufzeitumgebung, auf dem beliebig viele Agenten ausgeführt werden können.

Informationen über das Agentensystem kann man über die Operation *`get_agent_system_info()`* erlangen. Man bekommt als Rückgabewert eine Instanz der Klasse `AgentSystemInfo`. Diese Klasse wird in der MAF-Spezifikation festgelegt und enthält noch die Klassen `Name` (der Name dieser Instanz) und die Klasse `LanguageMap`, welche eine Liste der unterstützten Ausführungssprachen und Serialisierungsarten enthält.

5.2.4 Klasse AgentManager

Die wichtigste Aufgabe der Klasse `AgentManager` ist die tatsächliche Implementierung der Methoden aus den Schnittstellen *`MAFAgentSystem`* und *`AgentSystemService`*, die die Handhabung der Agenten betreffen. Die Attribute der Klasse sind:

- `_agentThreadGroup`: alle Agenten werden in diese Thread-Gruppe eingetragen
- `_ownAgentContext`: ist der Context, in den die Multiplen Agenten und die Exklusiven Agenten eingetragen werden.
- `_agentGlobalContext`: ist der Context in dem die Exklusiven Agenten eingetragen werden
- `_initContext`: der Root Context des *Agent Name Space*
- `_orb`: CORBA-Objektreferenz des ORB
- `_agentSystemService`: CORBA-Objektreferenz des Agentensystems
- `_agentTable`: Tabelle, welche die Agenten enthält
- `_migrate`: über diese Attribut erfolgt die Synchronisation mit den Agenten, die transferiert werden wollen.
- `_proxyPushConsumer`: Anbindung an den Standard Event Channel als *Supplier*

Die Methoden der Klasse sind:

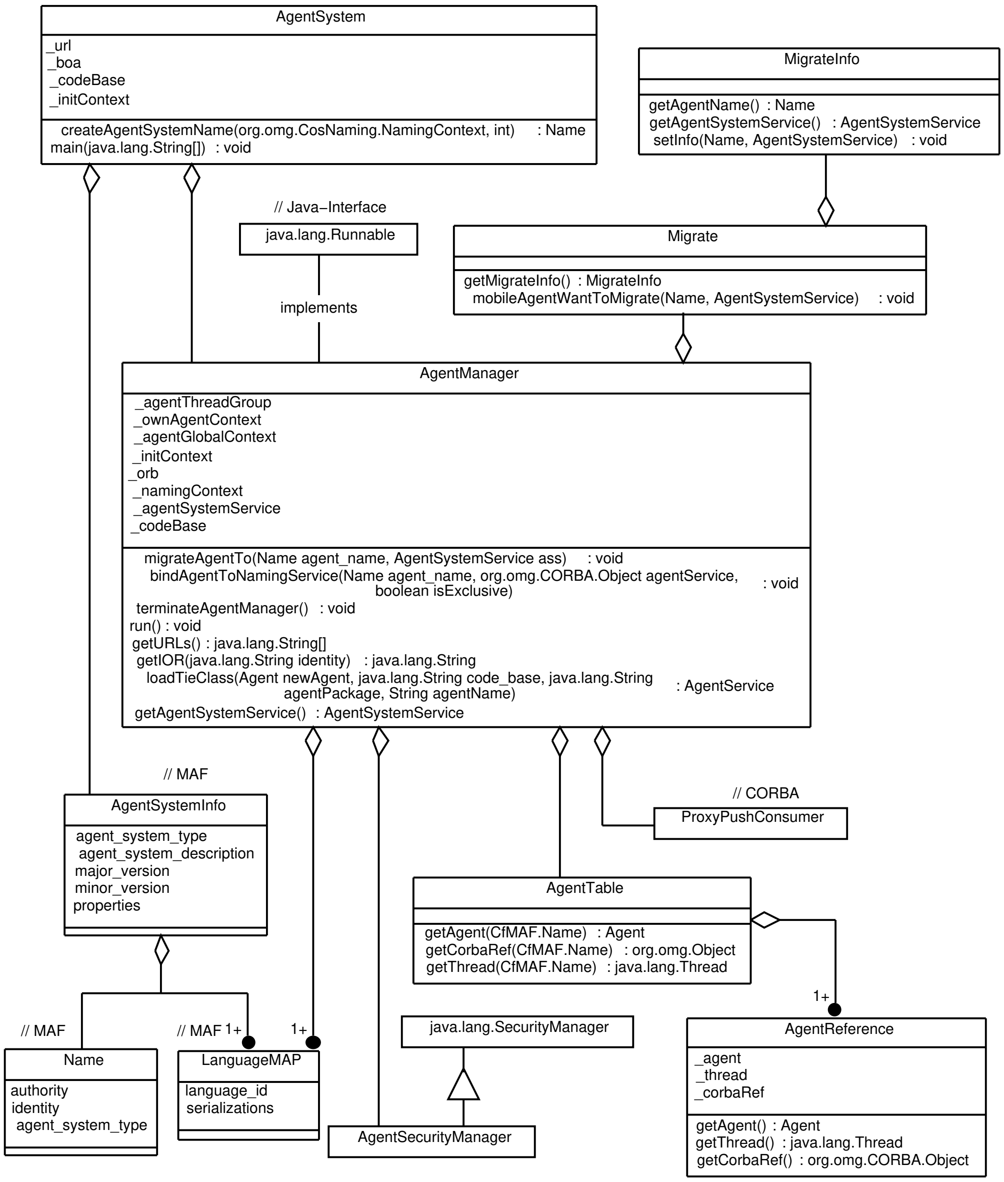

Abbildung 5.3: Teilmodell: Agentensystem

- `getURLs()`: liefert alle URLs der auf dem Agentensystem laufenden Agenten

- `getIOR(...)`: liefert die IOR zu einem bestimmten Agenten. Diese Methode wird vor allem vom Webserver-Agenten benötigt

- `bindAgentToNamingService(...)`: bindet einen Agenten an den Naming Service

- `loadTieClass(...)`: dynamisches Laden der Tie-Klasse mit Hilfe eines *Classloaders* (vgl. 6.1)

- `getAgentSystemService()`: liefert die CORBA-Objektreferenz des Agentensystems zurück. Wird von den Agenten benötigt.
- `migrateAgentTo(...)`: koordiniert den Ablauf des Transfers eines Agenten
- `terminateAgentManager()`: Terminierung des `AgentManagers`
- `run()`: Methode, die beim Start des `AgentManager` als Thread abgearbeitet wird

5.2.5 Klasse AgentTable

Die vom `AgentManager` verwalteten Agenten werden in der von `java.util.Hashtable` verfeinerten Klasse `AgentTable` gespeichert. Als Schlüssel für die Hashtabelle wird der Name des Agenten benutzt. Abgespeichert wird in der Hashtabelle jeweils eine Instanz der Klasse `AgentReference`. Diese Klasse definiert zusätzlich Methoden, die die jeweilige Referenz zurückgeben, um eine umständliche Typkonvertierung zu vermeiden[1]. Eine Hashtabelle eignet sich besonders für die Speicherung von Agenten, da man ohne Probleme neue Objekte einfügen kann (gegenüber Feldern) und Objekte über beliebige Schlüssel identifiziert werden (gegenüber der Indizierung von Feldern und Vektoren).

5.2.6 Klasse AgentReference

Diese Klasse speichert alle Referenzen eines Agenten. Die Referenzen sind:

- Java-Objektreferenz: erzeugt, durch Instanziierung einer Agentenklasse
- CORBA-Objektreferenz: erzeugt, durch die Zuweisung der Instanz einer Agentenklasse an seine IDL-Schnittstelle
- Thread-Referenz: erzeugt durch die Instanziierung eines Thread mit der Instanz der Agentenklasse

Durch die Kapselung der drei Referenzen eines Agenten genügt es, **eine** Hashtabelle im `AgentManager` zu halten.

5.2.7 Klasse AgentSecurityManager

Alle Agentklassen werden über einen *Classloader* geladen. Alle Klassen, die keine Systemklassen sind, werden von der *Java Virtual Machine* als kritisch eingestuft. Deshalb muß ein *SecurityManager* installiert werden, der alle aus Java-Sicht kritischen Methodenaufrufe überwacht, z. B. Lesen und Schreiben auf das lokalen Dateisystem, Abhören von Ports, Ausführung von Systemkommandos auf Betriebssystemebene, etc.

[1] Der Rückgabetyp eines Wertes von `java.util.Hashtable` ist immer `java.lang.Object`

Die Klasse `AgentSecurityManager` prüft die sicherheitskritischen Methodenaufrufe der Agenten und kann gegebenenfalls die Ausführung der Methode verbieten. Im Rahmen der Diplomarbeit wird nur eine prototypische Implementierung der Klasse vorgenommen, die aber in dem Fortgeschrittenenpraktikum von Robert Zeilhofer verfeinert wird.

5.3 Basisagenten-Modell

Beim Entwurf des Agentenmodells kommt es darauf an, ein Rahmenwerk zur Verfügung zu stellen, das eine einfache Einbindung von konkreten Agenten ermöglicht, sowie die Basisklassen aller Agenten zu entwerfen.

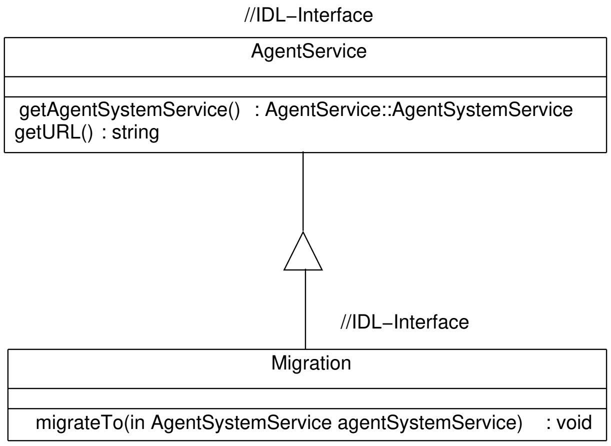

Abbildung 5.4: Teilmodell: Basisagent – IDL-Schnittstellen

5.3.1 IDL-Schnittstelle AgentService

Die IDL-Schnittstelle *AgentService* kann als Gegenstück zur IDL-Schnittstelle *AgentSystemService* angesehen werden (siehe Abbildung 5.4). Die Methode *getAgentSystemService()* hat als Rückgabewert die CORBA-Objektreferenz auf das Agentensystem, auf dem der Agent abläuft.

Wie das Agentensystem, besitzt auch jeder Agent eine Homepage, deren URL mit der Methode *getURL()* geliefert wird.

Die IDL-Schnittstelle *AgentService* sieht folgendermaßen aus:

```
#ifndef _AgentService_idl_
#define _AgentService_idl_

#include "AgentSystemService.idl"

module agent {

  interface AgentService
```

```
  {
    agentSystem::AgentSystemService getAgentSystemService();
    string getURL();
  };
};
#endif
```

Ein `module` bietet die Möglichkeit einer hierarchischen Gliederung und einer funktionalen Zusammenfassung von IDL-Schnittstellen.

5.3.2 IDL-Schnittstelle Migration

Diese IDL-Schnittstelle verfeinert die IDL-Schnittstelle ***AgentService*** um die Möglichkeit der Migration von Agenten. Über die Operation ***migrateTo(...)*** wird dem Agenten von außen angezeigt, auf ein anderes Agentensystem zu migrieren. D. h., daß ein Agent nicht nur von sich aus migrieren kann, sondern diese Aktion auch von einem Client angestoßen werden kann.

5.3.3 Klasse Agent

Die abstrakte Klasse `Agent` ist die zentrale Klasse dieses Teilmodells, da sie zum einen die Schnittstelle ***AgentService*** implementiert und zum anderen die Grundfunktionalität eines Agenten bereitstellt (siehe Abbildung 5.5). Sie ist somit auch die Basisklasse aller Agenten. Die Klasse `Agent` implementiert zusätzlich noch die Java-Schnittstellen `java.lang.Runnable`, damit ein Agent als Thread gestartet werden kann, und `java.io.Serializable`, damit der Agent serialisierbar ist[2]. Mit der Methode `init(...)` erfolgt die Initialisierung der Attribute der Klasse.

Es wird kein Konstruktor verwendet, da sonst die Parameterliste auch in allen Konstruktoren der Subklassen vorkommen und diese Parameter im Konstruktor korrekt an die Oberklasse übergeben werden müßten. Um die Einbindung neuer Agenten so einfach wie möglich zu halten, wurde dieser Weg, über die `init(...)`-Methode gewählt. Der Nachteil der Initialisierung über die init(...)-Methode ist, daß während der Initialisierung eines konkreten Agenten durch einen Konstruktor, nicht auf die Attribute der Basisklasse `Agent` zurückgegriffen werden kann.

Alle Attribute der Klasse sind als 'protected' deklariert, damit nur die Subklassen auf diese direkt zugreifen können. Die Attribute der Klasse sind:

- _url: Webseite des Agenten
- _codeBase: das Basisverzeichnis als URL, von dem die Klassen für diesen Agenten geladen werden
- _orb: Referenz des ORB

[2]Die Schnittstelle `java.io.Serializable` enthält keine Methoden, da sie nur als Indikator, daß eine Klasse serialisierbar sein soll, dient.

- _boa: Referenz des BOA
- _proxyPushConsumer: Bindung an den Event Channel. Der Agent kann Events, die er erzeugt, in diesen Event Channel schicken.
- _isExclusive: zeigt an ob es sich bei dem Agenten um einen Exklusiven Agenten handelt.
- _agentProfile, _className,_ authInfo, _agentStatus, _name werden in [GMD97] beschrieben.

Zu den einzelnen Attributen werden get-Methoden implementiert, damit der `AgentManager` darauf zugreifen kann. Die Methoden `readObject()` und `writeObject()` sind für die Serialisierung bzw. Deserialisierung des Agenten verantwortlich. Die Signaturen dieser Methoden sind fest vorgeschrieben (siehe [Fla97] S.434). Die Methode `initTransient(...)` wird vom `AgentManager` aufgerufen, um die Attribute, die mit 'transient' gekennzeichnet sind, zu initialisieren. Als 'transient' gekennzeichnete Attribute werden nicht serialisiert.

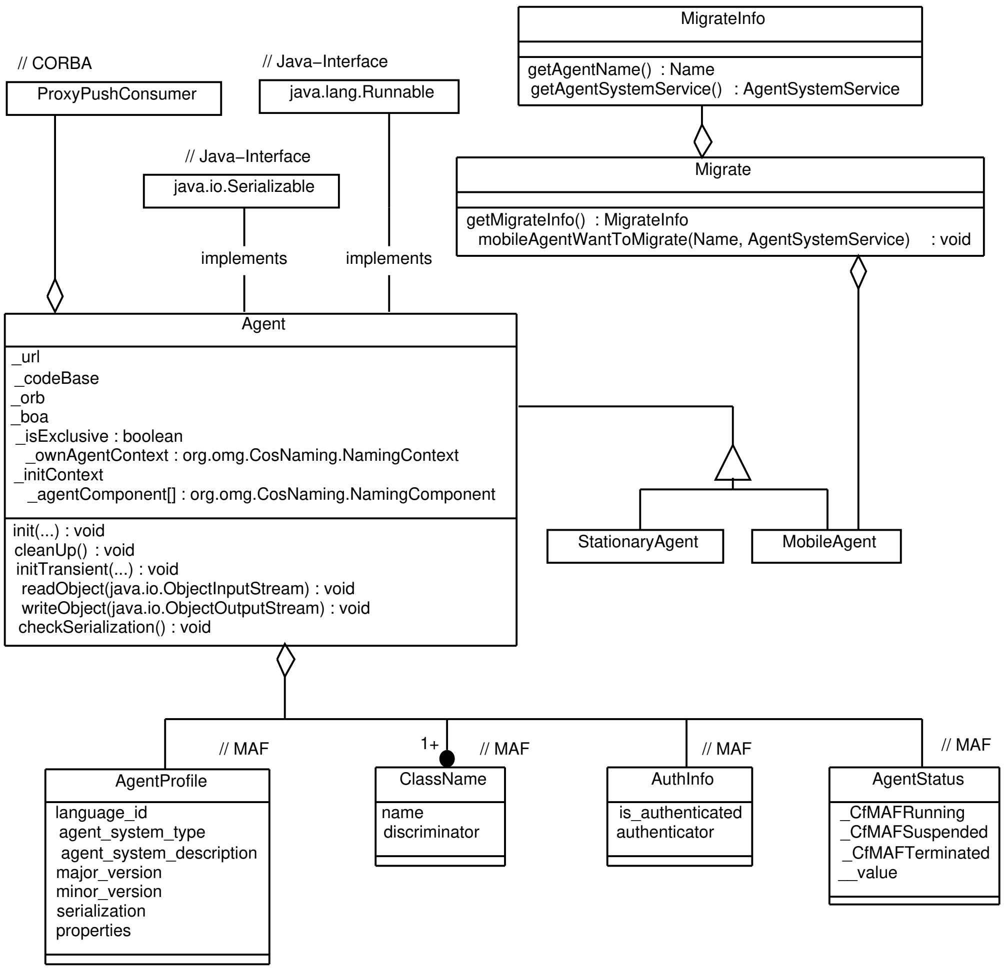

Abbildung 5.5: Teilmodell: Basisagent

Folgende Methoden sind als 'abstract' deklariert und müssen somit von allen Agenten implementiert werden:

- `cleanUp()`: Es sollen in dieser Methode, die vom Agenten belegten Ressourcen freigegeben werden. Diese Methode wird vom `AgentManager` aufgerufen, wenn der Agent terminiert wird.
- `checkSerialization()`: Diese Methode wird vom `AgentManager` aufgerufen, wenn der Agent serialisiert werden soll. Der Agent kann sich mit Hilfe dieser Methode in einen Zustand bringen, in dem eine Serialisierung sinnvoll ist. Will der Agent die Serialisierung ablehnen, so kann er die Ausnahmebehandlung `CouldNotMigrate` einleiten.

5.3.4 Klasse MobileAgent

Die Klasse `MobileAgent` erbt von der Basisklasse `Agent` und ist ebenfalls abstrakt. Nur Agenten, die von dieser Klasse erben, können zur Laufzeit auch migrieren, da diese Klasse die IDL-Schnittstelle ***Migration*** implementiert.

Die Methode `initTransient(...)` überlädt die Methode `initTransient(...)` aus der Basisklasse `Agent`.

Der `AgentManager` ruft bei der Instanziierung des Agenten die Methode `initMigrate(...)` auf, falls der Agent von der Klasse `MobileAgent` erbt.

5.3.5 Klasse StationaryAgent

Soll ein Agent nicht migriert werden können, weil er z. B. nur Informationen über den Host, auf dem er gestartet wurde, sammeln soll, so erbt er von der Klasse `StationaryAgent`. Da diese Klasse keine Attribute oder Methoden enthält, ist sie nicht notwendig. Sie dient aber als Pendant zur Klasse `MobileAgent`.

5.4 Schnittstelle zwischen Agentensystem und Agenten

Die Schnittstelle zwischen Agentensystem und Agenten bilden die Klassen `AgentManager`, `Agent` und `Migrate`. Die Klasse `AgentManager` verwaltet die Agenten und besitzt alle Rechte, Agenten zu erzeugen, zu terminieren, etc. Der Agent wiederum besitzt als Attribut eine Referenz auf seinen `AgentManager` und kann somit alle Methoden aufrufen, die im `AgentManager` als 'public' deklariert sind.

Über die `AgentTable` hat der `AgentManager` indirekten Zugriff auf die Agenten.

Die Klassen `Migrate` und `MigrateInfo` (siehe Abbildung 5.1) werden zur Migration des Agenten benötigt und detailliert in Kapitel 6 vorgestellt.

5.5 Realisierte Agenten

In den folgenden Unterabschnitten werden die Agenten, die im Rahmen dieser Diplomarbeit implementiert bzw. von der FMA-Architektur portiert wurden, vorgestellt. Die Einbindung dieser Agenten in das Basisagenten-Modell wird in Abbildung 5.6 dargestellt. Alle IDL-Schnittstellen der Agenten müssen entweder von der IDL-Schnittstelle *AgentService*, oder *Migration* erben, damit von der IDL-Schnittstelle der Agenten z. B. die Operation `getURL()` aufgerufen werden kann.

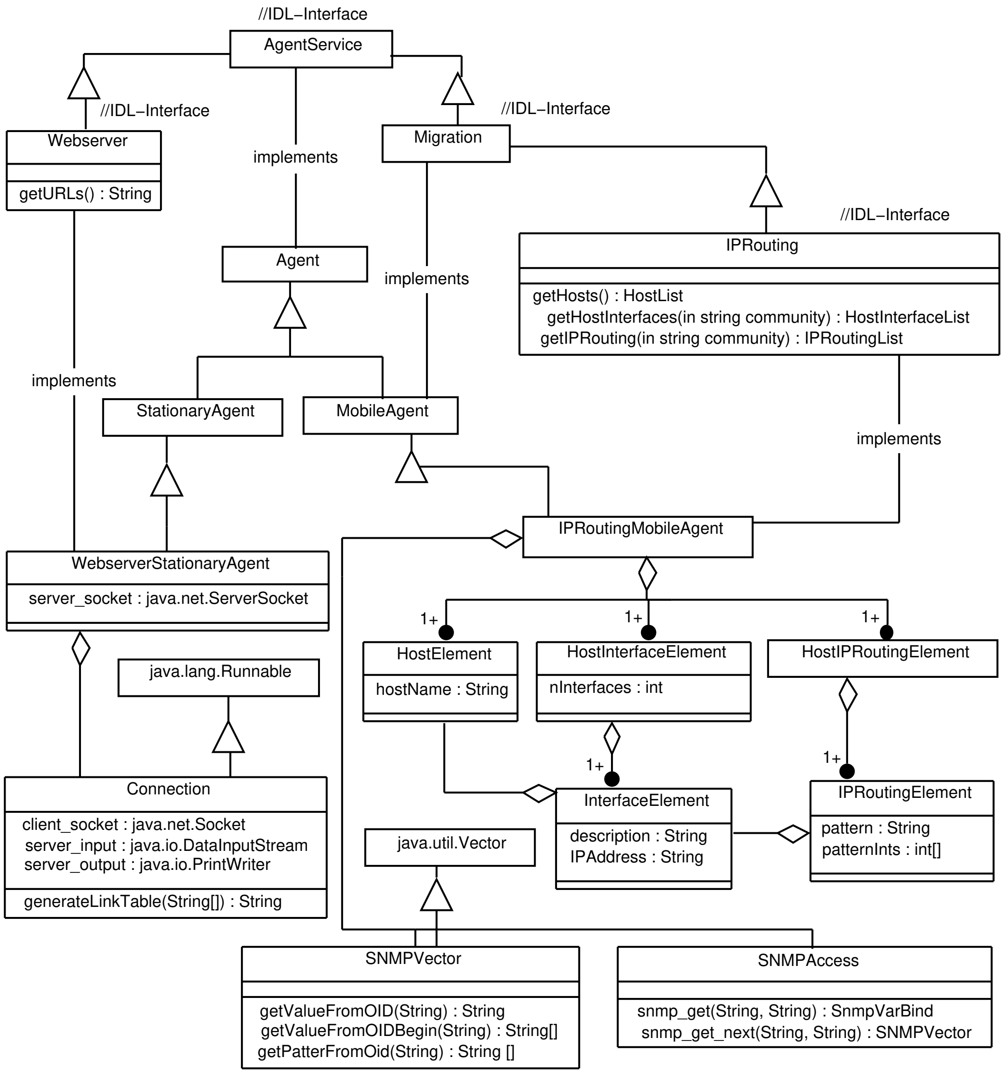

Abbildung 5.6: Einbindung des Webserver- und IPRouting-Agenten in das Basisagenten-Modell

5.5.1 Webserver-Agent

Der Webserver-Agent ist ein stationärer Agent, weil er für **ein** Agentensystem als Webserver fungiert. Deshalb erbt der Webserver-Agent von der Klasse `StationaryAgent`. Der Webserver-Agent bietet als IDL-Schnittstelle alle URLs (*`getURLs()`*) der auf dem Agentensystem ablaufenden Agenten an.

Die HTTP-Kommunikation wird von der Klasse `Connection` abgewickelt.

5.5.2 IPRouting-Agent

Der IPRouting-Agent kann von jedem Agentensystem aus seine Aufgabe erfüllen und ist deshalb ein Mobiler Agent. Dieser Agent ist die Zusammenfassung und Portierung von drei Interpretern der FMA-Architektur (vgl. [Kem97]). Die drei Interpreter sind:

- Agent Host Server (AHS) Interpreter: erzeugt eine Liste von Host, die aus dem Network Information Service (NIS) ausgelesen werden
- Agent Interface Server (AIS) Interpreter: erzeugt, aufbauend auf der Liste des AHS, u. a. zu jeden Host die Liste der Java-Interfaces und deren IP-Adresse
- Agent IP Routing Server (AIPRS) Interpreter: erzeugt, abhängig von der Liste des AIS, die zugehörige Routingtabelle

AIS und AIPRS holen sich die benötigte Information aus der MIB der auf den Hosts laufenden SNMP-Agenten.

Die Portierung der Interpreter gestaltete sich relativ einfach, da bei diesen Interpretern bereits eine RMI-Schnittstelle[3] vorhanden war. Lediglich die Datentypen der Rückgabewerte konnten nicht direkt in die IDL-Schnittstelle übernommen werden. Java-Klassen werden in IDL zu *structs* konvertiert und es existiert kein Vererbungsmechanismus auf *structs*. Die Java-Klassen `HostList`, `HostInterfaceList` und `IPRoutingList` erben voneinander und somit mußten die Datentypen für die IDL-Schnittstelle abgeändert werden. Die Methoden, welche diese Java-Klassen implementierten, gingen natürlich ebenfalls verloren. Die IDL-Schnittstelle sieht nun wie folgt aus:

```
#ifndef _IPRouting_idl_
#define _IPRouting_idl_
#include "Migration.idl"

module iprouting {
  exception ResourceException {string reason; };
  struct HostElement {
    string hostName;
  };
  typedef sequence<HostElement> HostList;
```

[3]Remote Method Invocation (RMI): ein RPC-Mechanismus, der ab JDK 1.1 vorhanden ist.

```
    struct InterfaceElement {
      string description;
      string IPAddress;
      HostElement host;
    };
    typedef sequence<InterfaceElement> InterfaceList;
    struct HostInterfaceElement {
      long nInterfaces;
      InterfaceList interfaces;
    };
    typedef sequence<HostInterfaceElement> HostInterfaceList;
    struct IPRoutingElement {
      string pattern;
      long patternInts[4];
      string IPAddress;
      InterfaceElement interfaceElement;
    };
    typedef sequence<IPRoutingElement> IPRoutingList;
    struct HostIPRoutingElement {
      IPRoutingList ipRoutingList;
    };
    typedef sequence<HostIPRoutingElement> HostIPRoutingList;

    interface IPRouting : agent::Migration
    {
      HostList          getHosts()
                          raises (ResourceException);
      HostInterfaceList getHostInterfaces(in string community)
                          raises (ResourceException);
      HostIPRoutingList getIPRouting(in string community)
                          raises (ResourceException);
      void              update();
    };
  };
  #endif
```

Die Funktionalität der einzelnen Operationen ist gleich geblieben. Neu hinzugekommen ist die Operation ***update()***, die bewirkt, daß bei einem Aufruf der `get*()`-Methoden, die jeweiligen Rückgabesequenzen neu erstellt werden und nicht die zwischengespeicherte Sequenz zurückgeliefert wird.

5.6 Lebenszyklus eines Agenten

Das (statische) Objektmodell ist in den letzten Abschnitten vorgestellt worden. Das 'Zusammenspiel' der Klassen, Schnittstellen und Methoden wird nun anhand des Lebenszyklus eines Agenten näher betrachtet (vgl. Unterabschnitt 2.4.3).

5.6.1 Bootstrapping

Der *Bootstrap*-Mechanismus ist vom Hersteller des ORBs, hier Visibroker for Java 3.0, abhängig, da CORBA dazu keine Vorgaben macht. Bevor ein Agent erzeugt werden kann, müssen eine Reihe von Applikationen gestartet werden. Dabei muß auf dem Rechner, auf dem die Applikationen ausgeführt werden, eine JVM und die Visibroker-Klassen vorhanden sein, außerdem muß die Umgebungsvariable '`OSAGENT_PORT`' für alle Applikationen gleich gesetzt sein.

- Start des *osagent*, einem *Deamon*, der einen Lokalisierungsdienst von CORBA-Objekten bereitstellt. Dieser muß nur einmal im Netz vorhanden sein, muß aber über *User Datagram Protocol (UDP)* für alle CORBA-Objekte und Clients erreichbar sein. Der *osagent* kann auf einem beliebigen Host gestartet werden.
- alle weiteren CORBA-Objekte können auf unterschiedlichen Hosts ablaufen und melden sich über einen UDP Broadcast beim *osagent* an:
  - der Naming Service
  - ein Event Channel
  - das Agentensystem

Der ORB wird nicht als separater Prozeß gestartet, sondern die Klassen, die den ORB bilden, werden zur Laufzeit von der JVM zu den Clients und CORBA-Objekten hinzugebunden.

5.6.2 Erzeugung eines Agenten

Es gibt drei mögliche Wege, wie die Operation *`create_agent(...)`* aufgerufen wird:

- ein beliebiger Client von außen, der über die CORBA-Objektreferenz des *`AgentSystemService`* auf die Operation zugreift, insbesondere auch ein Agent
- ein Agent innerhalb des Agentensystem über seinen `AgentManager`
- das Agentensystem selbst, z. B. den Webserver-Agent, welcher auf jedem Agentensystem ausgeführt werden muß

Die wichtigsten Parameter von *`create_agent(...)`* sind der Name und die Codebase. Das `AgentSystem` leitet den Aufruf unbearbeitet an den `AgentManager` weiter. Dieser prüft, ob der Agent, festgelegt durch den Namen, bereits in der `AgentTable`, oder im Naming Service vorhanden ist. Nun wird über den *Classloader* die Klasse abhängig von der Codebase geladen und instanziiert. Durch die `init(...)`-Methode der Klasse `Agent` werden die Attribute der Klasse gesetzt.

Der Agent wird mit Hilfe des Tie-Mechanismus (vgl. Abschnitt 6.1) an den BOA gebunden und damit über den ORB erreichbar. Nun wird der Agent als Thread gestartet und kann somit autonom handeln.

Eine Instanz der Klasse `AgentReference` wird erzeugt und diese Instanz als Wert und der Namen des Agenten als Schlüssel in die `AgentTable` eingetragen. Der `AgentManager` trägt den neuen Agenten in den Naming Service ein und sendet abschließend ein 'AgentUP' in den Standart Event Channel.

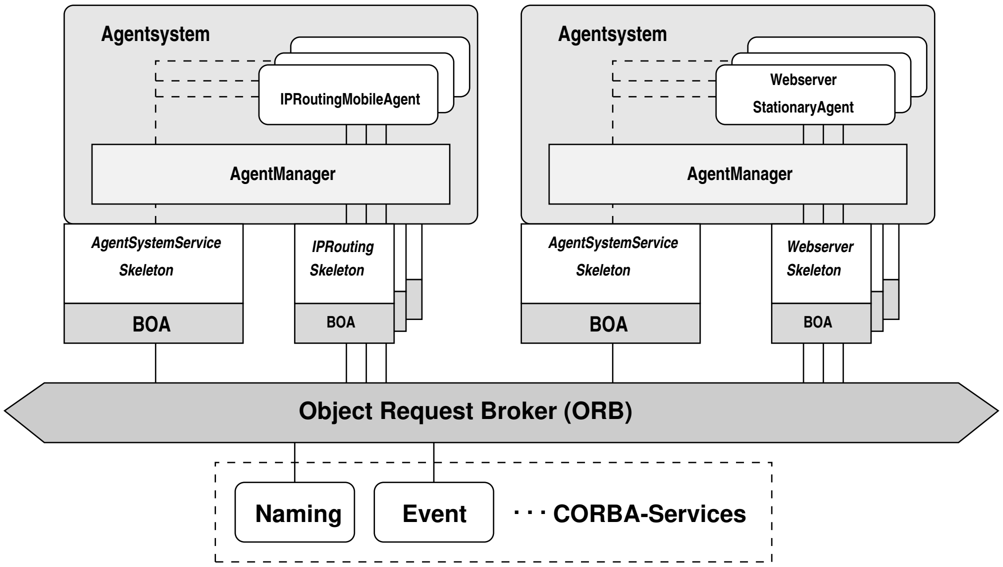

Abbildung 5.7: Beispielszenario der CORBA-Zugriffssicht

Abbildung 5.7 zeigt ein mögliches Szenario eines Managementumfeldes:
Auf jedem `AgentSystem` werden mehrere Agenten ausgeführt, die vom `AgentManager` kontrolliert werden. Das `AgentSystem` ist durch seine IDL-Schnittstelle *`AgentSystemService`* über CORBA erreichbar. Genauso ist jeder Agent, z. B. `IPRoutingMobileAgent` über die IDL-Schnittstelle *`IPRouting`* erreichbar.

5.6.3 Transfer eines laufenden Agenten

Ähnlich wie bei der Erzeugung eines Agent gibt es beim Transfer eines Agenten mehrere mögliche Wege, die Aktion auszulösen:

- ein beliebiger Client von außen, der über die CORBA-Objektreferenz von *`Migration`* die Operation *`migrateTo(...)`* aufruft
- ein anderer Agent innerhalb des Agentensystems, der Agent selbst, oder das Agentensystem über die Klasse `Migrate` mit der Methode `mobileAgentWantToMigrate(...)`

Nur Agenten die von der Klasse `MobileAgent` erben, können migriert werden. Es soll damit verhindert werden, daß Agenten, die z. B. von ihrer Programmierung her, nicht für einen Transfer geeignet sind, auch nicht migriert werden können. Das wird vom `AgentManager` durchgesetzt. Zusätzlich ruft der `AgentManager` die `checkSerialization()`-Methode des Agenten auf, damit dieser aktiv seine Serialierung beeinflussen kann. Der Agent wird suspendiert (`suspend_agent()`) und anschließend mit Hilfe von `writeObject(...)` der Klasse `Agent` zu einem Bytearray serialisiert, wobei eine Kopie des Agenten angelegt wird. Der Agent wird auf dem Quellagentensystem mittels `terminate_agent()` terminiert. Nun wird die Methode `receive_agent(...)` mit der CORBA-Objektreferenz des Zielagentensystems aufgerufen. Das Quellagentensystem entfernt nun den Agenten aus dem `AgentTable` und aus dem Naming Service.

Der `AgentManager` des Zielagentensystem stellt aus dem Bytearray mit der Methode `readObject(...)` der Klasse `Agent` den Agenten wieder her. Anschließend werden die transienten Attribute des Agenten mit `initTransient(...)` neu belegt. Die Tie-Klasse des Agenten wird über die Codebase nachgeladen. Nun wird wie bei `create_agent(...)` der transferierte Agent als Thread gestartet, falls der `AgentStatus` den Wert `Running` hat, eine `AgentReference` erzeugt, die in der `AgentTable` eingetragen wird. Schließlich wird der Agent noch in den Naming Service eingetragen.

Der migrierte Agent besitzt nun eine neue IOR. Damit wird bei Clients, die noch die alte CORBA-Objektreferenz besitzen und auf den Agenten zugreifen wollen, eine Ausnahmebehandlung eingeleitet, da der ORB keine Verbindung zum Agenten herstellen kann. Der Client muß nun über den Naming Service die neue IOR des Agenten herausfinden.

5.6.4 Terminierung eines Agenten

Es gibt vier mögliche Wege einen Agenten zu terminieren, indem die Operation *`terminate_agent()`* aufgerufen wird:

- ein beliebiger Client von außen, der über die CORBA-Objektreferenz des *`AgentSystemService`* auf die Operation zugreift
- ein Agent innerhalb des Agentensystem über seinen `AgentManager`
- das Agentensystem auf dem der Agent läuft
- der Agent selbst

Als erstes wird die Methode `cleanUp()` des Agenten aufgerufen. Dann wird der Thread, in dem der Agent läuft, vom `AgentManager` gestoppt. Der Agent wird aus dem Naming Service entfernt, der BOA deaktiviert den Agenten, damit keine Aufrufe von außen an den Agenten gehen können und abschließend wird die `AgentReference` des Agenten aus dem `AgentTable` gelöscht. In den Event Channel wird der Event 'AgentDown' geschickt.

Kapitel 6

Implementierung

Zu dem im letzten Kapitel beschriebenen Objektmodell, wird nun die Implementierung vorgestellt.

Zu Beginn wird der Tie-Mechanismus, zur Anbindung von CORBA-Objekten an den BOA, erklärt und die exakte Vererbungshierarchie am Beispiel des IPRouting-Agenten nachgeholt.

Anschließend wird eine Übersicht über die Strukturierung des Quellcodes gegeben. Wie bei der Realisierung werden die einzelnen Klassen getrennt nach den Teilmodellen vorgestellt, dabei werden exemplarisch wichtige Programmteile bzw. Algorithmen betrachtet.

Dann werden nacheinander die implementierten Agenten, Hilfsklassen und Applets beschrieben. Anschließend wird die Entwicklung eines neuen Agenten detailliert beschrieben.

Abschließend wird die Installation der Implementierung vorgestellt.

6.1 Der Tie-Mechanismus

Der Tie-Mechanismus wird benötigt, da Java nur die Einfachvererbung von Klassen unterstützt.

Der 'normale' Weg, ein IDL-Schnittstelle durch eine Klasse zu implementieren ist, daß diese Klasse von einer Skeleton-Klasse, die von einem idl2java-Compiler generiert wird, erbt. Damit scheidet in Java zum einen die Möglichkeit aus, noch eine weitere IDL-Schnittstelle zu implementieren, und zum anderen kann auch von keiner anderen Klasse geerbt werden.

Dieses Dilemma wird durch den Tie-Mechanismus gelöst. Der größte Unterschied zum 'normalen' Weg der Implementierung ist, daß nun die Instanz einer sogenannten Tie-Klasse an den BOA gebunden wird. Als Parameter des Konstruktors der Tie-Klasse wird eine Instanz der Implementierungsklasse übergeben. Die Tie-Klasse leitet die Aufrufe nur an die Implementierungsklasse weiter.

Damit kann die eigentliche Implementierungsklasse von einer beliebigen Klasse erben. Der Preis für den Tie-Mechanismus ist die zusätzliche Instanziierung eines Tie-Objekts und ein zusätzlicher Methodenaufruf pro Aufruf einer IDL-Schnittstellenoperation.

Das Agentensystem und alle Agenten werden über den Tie-Mechanismus an den BOA gebunden.

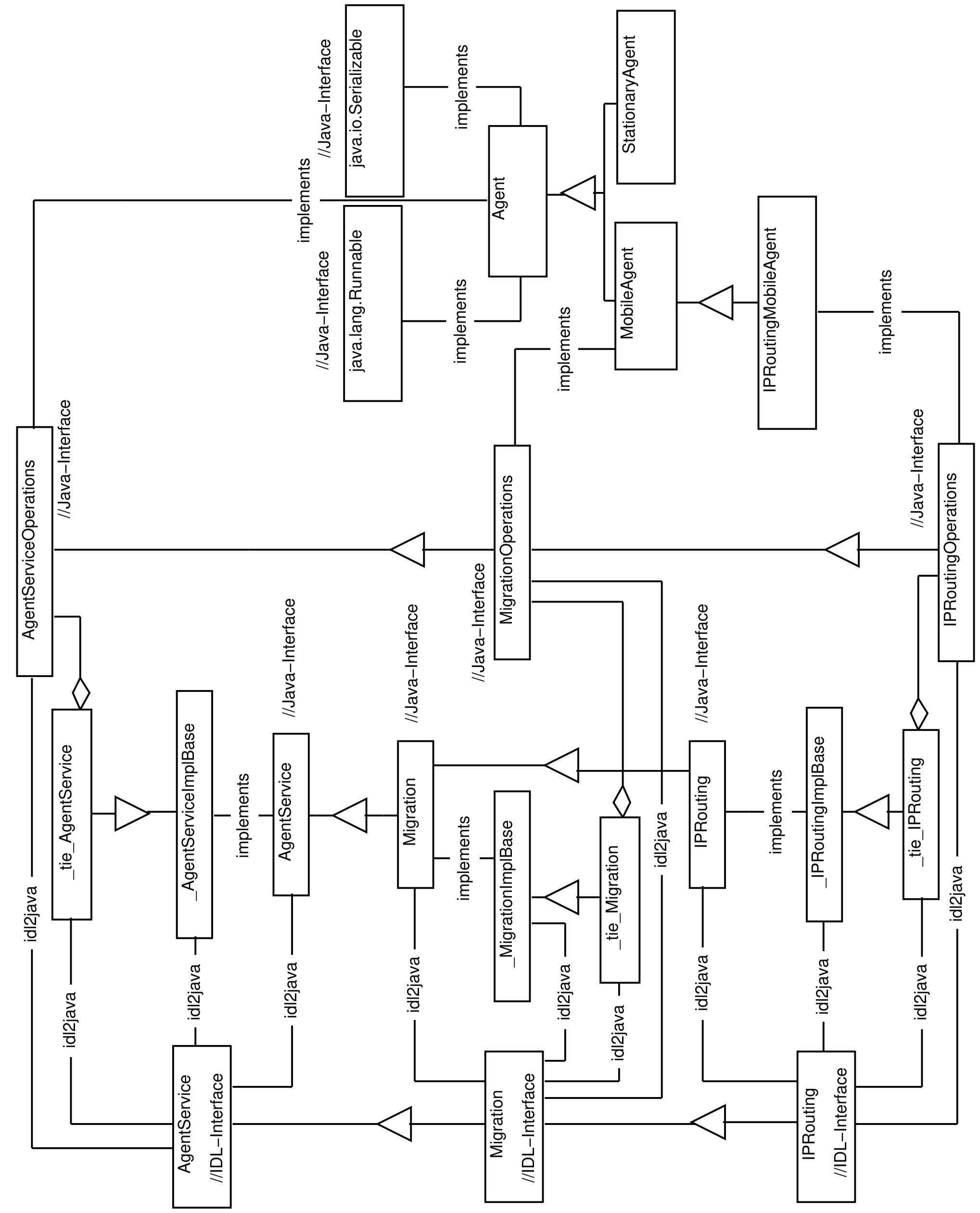

Abbildung 6.1: Tie-Hierarchie

Abbildung 6.1 zeigt nun das exakte Objektmodell bezüglich der Vererbungshierarchie im Agenten-Modell. Das Agentensystem besitzt eine äquivalente Vererbungshierarchie, wird aber hier nicht weiter erörtert.

Am Beispiel des IPRouting-Agenten wird nun die Einbindung erklärt. Auf der linken Seite der Abbildung sind die IDL-Schnittstellen ***AgentService***, ***Migration*** und ***IPRouting*** abgebildet. Aus diesen IDL-Schnittstellen generiert das Programm `idl2java` u. a. die folgenden Objekte:

- Java-Schnittstellen (z. B. `IPRouting` und `IPRoutingOperations`)
- Server Skeletons (z. B. `_IPRoutingImplBase`)
- Tie-Klassen (z. B. `_tie_IPRouting`)
- Client Stubs (z. B. `IPRoutingHelper`), die nicht abgebildet sind, da sie für die Erklärung des Tie-Mechanismus nicht relevant sind.

Diese generierten Objekte wurden im Objektmodell des Kapitel 5 aus Gründen der Übersichtlichkeit vernachlässigt. Die im rechten Teil des Objektmodells angesiedelten Klassen sind aus Kapitel 5 bekannt und stellen den Basis, sowie den IPRouting-Agenten dar. Wie zu sehen ist, implementiert die Klasse `Agent` die Java-Schnittstelle `AgentServiceOperations`, die Klasse `MobileAgent` die Java-Schnittstelle `MigrationOperations` und die Klasse `IPRoutingMobileAgent` implementiert die Java-Schnittstelle `IPRoutingOperations`.

Die Instanziierungen der einzelnen Klassen wird im Folgenden beschrieben:
Der `AgentManager` erzeugt eine Instanz der Klasse `_tie_IPRouting`, welche an den BOA als Referenz übergeben wird. Als Parameter wird dem Konstruktor eine Instanz der Schnittstelle `IPRoutingOperation` übergeben, die von der Klasse `IPRoutingMobileAgent` implementiert wird:

```
1  IPRoutingOperations ipInterface= new IPRoutingMobileAgent();
2  _tie_IPRouting tieObj= new _tie_IPRouting(ipInterface);
3  boa.obj_is_ready(tieObj);
```

Es wird eine Instanz der Java-Schnittstelle `IPRoutingOperations` erzeugt (Zeile 1). Das Tie-Objekt wird kreiert, indem es mit der Java-Schnittstelle initialisiert wird (Zeile 2). Dieses Tie-Objekt wird zuletzt an den BOA gebunden (Zeile 3).

Die Aufrufhierarchie einer CORBA-Operation ist nun wie folgt:
Wird nun der Operationsaufruf ***update()*** über CORBA an den BOA übermittelt, so werden der Reihe nach folgende Klassen aufgerufen:

```
{\tt \_IPRoutingImplBase.invoke(update)}
{\tt \_tie\_IPRouting.update()}
{\tt IPRoutingOperations.update()}
{\tt IPRoutingMobileAgent.update()}
```

6.2 Übersicht über den Quellcode

Um den Quellcode strukturieren zu können, unterstützt Java die Gruppierung von Java-Klassen. Ein Java-*Package* ist die Zusammenfassung von Java-Klassen. Das Gruppierungskonzept ist per Konvention an der *Domain Name*-Hierarchie des Internets angelehnt. Es wird eine weltweit eindeutige Hierarchie von *Packages* aufgebaut, indem der *Package*-Name mit dem umgedrehten *Domain Name* der Organisation oder Firma, die diese Klassen entwickeln, beginnt. Anschließend folgt der Projektname um eine Unterscheidung zwischen verschiedenen Projekten zu erreichen. Der komplette *Package*-Name für dieses Projekt lautet somit:
`de.unimuenchen.informatik.mnm.masa`

wobei `masa` für *Mobile Agent Systems Architecture* steht. Ausgehend von diesem *Package*-Name erfolgt eine weitere Unterteilung in

- `agentSystem`: alle Klassen, die das Agentensystem implementieren
- `agent`: alle Klassen, die den Basisagenten implementieren
- `event`: alle Klassen zum Aufbau eines Event Channel
- `tools`: verschiedene Hilfklassen

Ein konkreter Agent, z. B. der IPRouting-Agent wird unter dem *Package* `agent` mit dem *Package*-Namen `iprouting` eingehängt. Der vollständige *Package*-Name, unter dem sich die Klassen des IPRouting-Agenten befinden, ist:
`de.unimuenchen.informatik.mnm.masa.agent.iprouting`

Die aus den IDL-Schnittstellen der MAF-Spezifikation generierten Java-Klassen und Java-Schnittstellen werden in dem *Package* `CfMAF` abgelegt.

Aus dem Objektmodell wurden Code-Skelette mit Hilfe des StP-Codegenerators erzeugt. Da StP keine 'implements' als Assoziation unterstützt, mußten diese Angaben bei den entsprechenden Klassen von Hand nachgetragen werden.

6.3 Implementierung des Agentensystems

In diesem Abschnitt werden die zentralen Klassen des Agentensystems `AgentSystem`, `AgentManager` und `Migrate` vorgestellt.

6.3.1 Klasse AgentSystem

Die Klasse `AgentSystem` implementiert als wichtigste Methode, die `main(...)`-Methode.

Methode main(...)

Diese Methode wird beim Start des Agentensystems abgearbeitet und kommt ihr deshalb besondere Bedeutung zu. Es werden nun der Reihe nach die wichtigsten Verarbeitungsschritte beschrieben:
Zu Beginn wird die *Property*-Datei eingelesen und für alle Klassen, die innerhalb dieses Prozesses instanziiert sind, zur Verfügung gestellt.

Das erste Agentensystem, welches sich an den Naming Service bindet, generiert den gesamten Namensraum, sowohl für die Agentensysteme als auch für die Agenten. Dabei wird der Naming Service folgendermaßen kontaktiert:

```
org.omg.CORBA.ORB orb= org.omg.CORBA.ORB.init();
org.omg.CORBA.Object namingService=
  orb.resolve_initial_references("NameService");
```

Zuerst wird der ORB initialisiert, anschließend eine CORBA-Objektreferenz des Naming Service in Form des Root Context (`namingService`) zurückgeliefert. Die Typkonvertierung von CORBA-Objektreferenzen wird immer mit der `narrow()`-Methode der entsprechenden *Helper*-Klasse durchgeführt:

```
org.omg.CosNaming.NamingContext rootContext =
  org.omg.CosNaming.NamingContextHelper.narrow(namingService);
```

Die Typkonvertierung ist notwendig, denn `namingService` ist vom Typ `org.omg.CORBA.Object`, der Basisschnittstelle aller CORBA-Objekte und kann so erst einmal nicht auf die Methoden, welche die Schnittstelle `org.omg.CosNaming.NamingContext` bereitstellt, zugreifen.

Die Klasse `AgentSystem` und die Klasse `AgentManager` werden instanziiert. Nun bindet sich das Agentensystem selbst mit dem oben vorgestellten Tie-Mechanismus an den ORB:

```
AgentsystemService agentSystemService=
  new _tie_AgentSystemService(agentSystem);
boa.obj_is_ready(agentSystemService);
```

wobei `agentSystem` eine Instanz der Klasse `AgentSystem` ist.

Das Agentensystem trägt sich in den Naming Service ein, schickt in den Standard Event Channel ein '`AgentSystemUp`' und wird schließlich durch den Aufruf

```
boa.impl_is_ready();
```

in eine Endlosschleife gebracht, um die Terminierung des Programms zu verhindern.

6.3.2 Klasse AgentManager

Eine Instanz der Klasse `AgentManager` wird von der Klasse `AgentSystem` erzeugt.

Es folgt nun die Motivation für die Implementierung des `AgentManager` als Thread:
Ein Agent der seinem `AgentManager` anzeigen will, daß er migrieren will, könnte das abwickeln, indem er eine Methode des `AgentManager` aufruft. Innerhalb dieser Methode würde der `AgentManager` u. a. den Agenten suspendieren. Das Problem dabei ist, daß diese Methode innerhalb des Thread des **Agenten** ablaufen würde. Kommt man nun an den Ausführungspunkt der Suspendierung, so wird der Agent **und** der `AgentManager` angehalten. Die Lösung dieses Problems ist, daß auch der `AgentManager` ein Thread ist und über eine Klasse `Migrate` (vgl. 6.3.3) der synchronisierte Informationsaustausch mit den Agenten geregelt wird.

Die Klasse `AgentManager` implementiert alle Operationen der IDL-Schnittstellen *`MAFAgentSystem`* und *`AgentSystemService`*, welche die Handhabung der Agenten betreffen. Nun werden die wichtigsten Methoden vorgestellt.

Methode create_agent(...)

Die Erzeugung eines Agenten ist eine der wichtigsten Methoden der Klasse `AgentManager`. Diese Methode hat u. a. den Parameter 'agent', einem Bytearray, das der `AgentManager` benötigt um den Agent zu instanziieren. In diesem Parameter sind mehrere Objekte in serialisierter Form enthalten, die ausgelesen werden müssen:

```
 1  java.util.Vector agentArgs= null;
 2  int numberOfObjects= 4;
 3  if ( agent.length > 0 ) { // is array greater than one
 4    agentArgs= new java.util.Vector(numberOfObjects);
 5    try { // read from stream
 6      java.io.ObjectInputStream input=
 7        new java.io.ObjectInputStream(
 8          new java.io.ByteArrayInputStream(agent));
 9      while( true ){ // read till end of stream
10        agentArgs.addElement(input.readObject());
11      }
12    }
13    catch(java.io.EOFException eofe){
14      //Ok, stream full read
15    }
16    catch(java.lang.Exception e){//Error during input.readOject()
17      e.printStackTrace();
18      throw new CfMAF.ArgumentInvalid();
19    }
20  }
```

Die zu lesenden Objekte sollen zur Weiterverarbeitung in einen Vektor namens `agentArgs` gespeichert werden (Zeile 1). Eine Instanz der Klasse `java.io.ObjectInputStream` wird erzeugt, indem es mit dem Bytearray `agent` instanziiert wird (Zeile 6-8). Dieser Stream `input` wird solange gelesen (Zeile 10) und in den Vektor `agentArgs` eingetragen, bis das Ende des Streams erreicht ist. Dann wird die Ausnahmebehandlung `java.io.EOFException` eingeleitet und im zugehörigen `catch`-Block abgefangen (Zeile 13). Sollte ein unerwarteter Fehler beim Lesen dieses Parameters auftreten, so wird die Exception auf der Standardausgabe angezeigt (Zeile 17) und die Abarbeitung dieser Methode beendet, indem die Ausnahmebehandlung `CfMAF.ArgumentInvalid()` angestoßen wird.

Das Laden der Agenten-Klasse wird mit der Klasse `java.rmi.server.RMIClassloader` vorgenommen. Dazu muß aus dem übergebenen *Package*-Namen und dem Namen des Agenten die zu ladende Klasse zusammengesetzt werden. Mit Hilfe des Parameter `code_base`, welches eine URL in Stringformat sein muß, wird die Klasse und die zugehörige Tie-Klasse geladen. Nun werden der Reihe nach folgende Schritte durchgeführt:

1. der Konstruktor der Agenten-Klasse wird aufgerufen
2. die `init(...)`-Methode des Basisagenten wird aufgerufen
3. der Agent wird durch seine Tie-Klasse an den BOA gehängt
4. die Referenzen des Agenten werden in den `AgentTable` gespeichert
5. der Agent wird in den Naming Service eingehängt
6. es wird in den Standard Event Channel ein '`AgentUp`' geschickt
7. der Agent wird als Thread gestartet

Methode migrateAgent(...)

Diese Methode nimmt die tatsächliche Migration des Agenten vor. Es gibt zwei prinzipiell Ansätze, wie bei der Migration vorgegangen werden kann:

- **ein** Agent muß weiterlaufen: der Agent wird suspendiert, kopiert, serialisiert, zum Zielagentensystem gesendet, dort deserialisiert und gestartet. Ist dieser Vorgang fehlerfrei durchgeführt worden, so wird der ursprüngliche Agent auf dem Quellagentensystem terminiert, ansonsten wird er wieder gestartet. Problematisch bei diesem Ansatz ist, daß zur gleichen Zeit zwei Agenten derselben Klasse laufen. Es kann damit leicht zu Konfliktsituationen, z.B beim Naming Service, kommen.
- jede mögliche Konfliktsituation wird vermieden: d. h., der Agent wird suspendiert, kopiert, serialisiert und schon jetzt auf dem Quellagentensystem terminiert. Nun wird der serialisierte Agent zum Zielagentensystem gesandt, dort deserialisiert und gestartet. Das Problem bei diesem Ansatz ist, daß falls bei der Migration ein Fehler auftritt, weder auf dem Quell- noch auf dem Zielagentensystem ein Agent dieses Typs läuft.

Der Autor hat sich für den zweiten Ansatz entschieden, da die möglichen Konfliktsituationen des ersten Ansatzes eventuell das gesamte Managementsystem in Mitleidenschaft ziehen.

Es soll auch die Möglichkeit bestehen, bereits suspendierte Agenten zu migrieren. Damit ein Agent auf dem Zielagentensystem nicht gestartet wird, muß dem Zielagentensystem der aktuelle Zustand des Agenten übermittelt werden. Das hat zur Folge, daß bevor der Agent auf dem Quellsystem terminiert wird, sein aktueller Zustand in den Bytestrom geschrieben wird, da sonst das Zielagentensystem annimmt, daß der Agent terminiert ist und nicht mehr gestartet wird. Erst dann kann der Agent serialisiert werden.

Nun ruft der `AgentManager` die Methode `receive_agent(...)` auf dem Zielagentensystem auf.

Methode receive_agent(...)

Die Methode `receive_agent(...)` hat vom logischen Aufbau her die gleiche Struktur wie `create_agent()`. Nur mit dem Unterschied, daß der Agent nicht mit dem *Classloader* geladen, sondern aus einen *Bytearray* deserialisiert wird:

```
 1  try{// read state of the agent and then the agent
 2    java.io.ByteArrayInputStream in =
 3    new java.io.ByteArrayInputStream(agent);
 4    java.io.ObjectInputStream inputStream =
 5    new java.io.ObjectInputStream(in);
 6
 7    state= (CfMAF.AgentStatus)inputStream.readObject();
 9    newAgent= (Agent)inputStream.readObject();
10    inputStream.close();
11  }
12  catch(java.IOException ioe){//Error
13    ioe.printStackTrace();
15    throw new CfMAF.DeserializationFailed();
16  }
```

Zu Beginn wird der `java.io.ObjectInputStream` mit dem Agenten `agent` instanziiert (Zeile 5). Aus dem Bytestream wird der Zustand `state` des Agenten ausgelesen (Zeile 7). Anschließend wird der eigentliche Agent `newAgent` deserialisiert (Zeile 9). Falls ein Fehler bei der Deserialisierung aufgetreten ist, wird die Exception abgefangen und dem aufrufenden Client dies mit der Exception `CfMAF.DeserializationFailed()` angezeigt.

Da als *Codebase* der Agenten stets eine URL angegeben werden muß, kann man davon ausgehen, daß es prinzipiell von jedem Agentensystem aus möglich ist, die benötigte Tie-Klasse zu laden.

Die als '`transient`' gekennzeichneten Attribute des Agenten werden mit Hilfe der `initTransient(...)`-Methode initialisiert. Anschließend ist der Ablauf wie bei `create_agent(...)`, mit der Ausnahme, daß der Agent nur gestartet wird, wenn der Zustand `state` gleich `CfMAF.Running` ist.

Es bleibt anzumerken, daß beim Start des Thread, die Methode `run()` wieder **von Beginn an** abgearbeitet wird. Das liegt daran, daß ein Thread selbst nicht serialisiert werden kann und deshalb auf dem Zielagentensystem ein neuer

Thread für den Agenten instanziiert werden muß. Das bedeutet aber nicht, daß es sich hierbei um eine neue Instanziierung des Agenten handelt, da seine Attributewerte transferiert worden sind.

6.3.3 Klasse Migrate

Die Klasse `Migrate` dient zur synchronisierten Kommunikation zwischen dem `AgentManager` und den `MobileAgents`. Die Klasse funktioniert nach dem klassischen Erzeuger-Verbraucher-Problem.

Es gibt nur eine Instanz der Klasse `Migrate` im gesamten Agentensystem. Alle `MobileAgents` und der `AgentManager` besitzen somit eine Referenz desselben Objekts.

Java setzt das Konzept des Monitors zur Synchronisation von Threads ein. Der exklusiven Zugriff auf eine Klasse wird durch Methoden geregelt, die mit dem zusätzliche Schlüsselwort '`synchronized`' gekennzeichnet sind.

Alle Methoden der Klasse `Migrate` sind als '`synchronized`' deklariert. Ruft ein Thread eine Methode der Klasse `Migrate` auf, so wird der Zugriff auf die Klasse für alle anderen Threads gesperrt.

`AgentManager` und `MobileAgents` müssen die Möglichkeit haben, sich gegenseitig zu benachrichtigen, daß die Nachricht gelesen bzw. geschrieben wurde. Das geschieht in Java mit den Methoden `wait()` und `notify()` der Klasse `java.lang.Object`.

Methode mobileAgentWantToMigrate(...)

Ein `MobileAgent` ist der Erzeuger und zeigt mit dieser Methode an, daß er migrieren will:

```
public synchronized void
mobileAgentWantToMigrate(CfMAF.Name agentName,
                         AgentSystemService destination)
  throws de.unimuenchen.informatik.mnm.masa.agent.CouldNotMigrate{
  while (_migrateAgent == true) {
    try { // wait till agentManager read MigrateInfo
      wait();
    }
    catch (InterruptedException e) {}
  }
  // _migrateInfo has type MigrateInfo
  _migrateInfo.setInfo(agentName,destination);
  _migrateAgent = true;
  notifyAll();
  try{
    wait();
  }
  catch(java.lang.InterruptedException e){}
```

```
19   if ( ! _couldMigrate)
20     throw new CouldNotMigrate();
21 }
```

Als Parameter muß der Name des Agenten und das Zielagentensystem in Form einer CORBA-Objektreferenz angegeben werden. Eventuell muß der Agent warten, bis der `AgentManager` eine weitere Migration abgearbeitet hat (Zeile 5-10). Anschließend wird der eigentliche Migrationswunsch in das Attribut `_migrateInfo` geschrieben (Zeile 12). Der `AgentManager` wird durch den Aufruf von Methode `notifyAll()` (Zeile 14) aus seinem blockierenden Aufruf `wait()` in der Methode `getMigrateInfo()` unterbrochen (Zeile 4). Der Agent wird wiederum mit der Methode `wait()` solange blockiert (Zeile 16), bis das Quellagentensystem die Migration erfolgreich durchgeführt hat, oder ein Fehler aufgetreten ist und eine Ausnahmebehandlung eingeleitet wird (Zeile 20).

Methode getMigrateInfo()

Der `AgentManager`, hat entweder in der Zeile 4 gewartet, oder er ruft die Methode `getMigrateInfo()` gerade auf und da _migrateAgent den Wert '`true`' hat, wird in beiden Fällen in Zeile 8 mit der Abarbeitung fortgefahren:

```
1 protected synchronized MigrateInfo getMigrateInfo() {
2   while (_migrateAgent == false) {
3     try {
4       wait();// wait till a MobileAgent set MigrateInfo
5     }
6     catch (InterruptedException e) {}
7   }
8   return _migrateInfo;
9 }
```

Nun ruft der `AgentManager` die oben beschriebene Methode `migrateAgent(...)` auf (vgl. Unterabschnitt 6.3.2) und migriert den Agenten. Mit den Methode `setCouldMigrate(...)`:

```
protected synchronized static void
setCouldMigrate(boolean couldMigrate){
  _couldMigrate= couldMigrate;
  _migrateAgent = false;
}
```

wird angezeigt, ob die Migration erfolgreich war und mit der Methode `notifyAgent()`:

```
protected synchronized void notifyAgent(){
  notify();
}
```

wird schließlich der Monitor in der Methode `mobileAgentWantToMigrate(...)` wieder freigeben (Zeile 16), so daß ein neuer `MobileAgent` in die Methode eintreten kann.

`MobileAgent` in der Methode `mobileAgentWantToMigrate(...)` aus dem `wait()` in Zeile 16 unterbrochen.

Die Migration ist vollständig abgeschlossen.

6.4 Implementierung des Basisagenten

Der Basisagent wird von den Klassen `Agent`, `MobileAgent` und `StationaryAgent` gebildet. Da der `StationaryAgent` keine Methoden implementiert, entfällt hier die Beschreibung.

Das Basisapplet aller Agenten wird am Ende dieses Abschnitts vorgestellt.

6.4.1 Klasse Agent

Aus der Klasse `Agent` werden die beiden Methoden `writeObject(...)` und `readObject(...)` vorgestellt.

Methoden readObject(...) und writeObject(...)

Diese Methoden übernehmen die Serialisierung der Objekte und sind bereits implementiert. Nur bei den Klassen, die nicht serialisierbar sind, muß man diese Methoden überladen und neu implementieren.

Das Attribut vom Typ `AgentProfile` enthält einen Typ `org.omg.CORBA.Any` welcher nicht serialisierbar ist. Deshalb werden die Methoden `writeObject()` und `readObject()` neu implementiert:

```
 1  private void writeObject(java.io.ObjectOutputStream out)
 2    throws java.io.IOException {
 3    // write AgentProfile in the stream
 4    out.writeShort(_agentProfile.language_id);
 5    out.writeShort(_agentProfile.agent_system_type);
 6    out.writeObject(_agentProfile.agent_system_description);
 7    out.writeShort(_agentProfile.major_version);
 8    out.writeShort(_agentProfile.minor_version);
 9    out.writeShort(_agentProfile.serialization);
10    out.writeInt(_agentProfile.properties.length);
11    for(int i=0; i < _agentProfile.properties.length; i++)
12      out.writeObject(_agentProfile.properties[i].extract_wstring());
13    // go on with the rest of the class
14    out.defaultWriteObject();
15  }
```

In den Zeilen 4-12 werden die Attribute der Klasse `AgentProfile` einzeln serialisiert und in den Stream geschrieben. Anschließend wird mit der Serialisierung der Agenten-Klasse fortgefahren.

Die Methode `readObject(...)` liest die Objekte in derselben Reihenfolge wieder aus:

```
private void readObject(java.io.ObjectInputStream in)
  throws java.io.IOException, java.lang.ClassNotFoundException{
   _agentProfile= new CfMAF.AgentProfile();
  // read AgentProfile out of stream
   _agentProfile.language_id= in.readShort();
   _agentProfile.agent_system_type= in.readShort();
   _agentProfile.agent_system_description= (String)in.readObject();
   _agentProfile.major_version= in.readShort();
   _agentProfile.minor_version= in.readShort();
   _agentProfile.serialization=  in.readShort();
  int nProperties= in.readInt();
  String props[]= new String[nProperties];
  org.omg.CORBA.Any anies[]= new org.omg.CORBA.Any[nProperties];
  for(int i=0; i < nProperties; i++)
    props[i]= (String) in.readObject();
  org.omg.CORBA.ORB orb= org.omg.CORBA.ORB.init();
  for(int i=0; i < nProperties; i++) {
    org.omg.CORBA.Any any= orb.create_any();
    any.insert_wstring(props[i]);
    anies[i]= any;
  }
   _agentProfile.properties= anies;
 // read the rest of the class
 in.defaultReadObject();
}
```

6.4.2 Klasse MobileAgent

Die Klasse `MobileAgent` implementiert u. a. die Methode `migrateTo()` aus der IDL-Schnittstelle ***Migration***.

Methode migrateTo(...)

In der Methode `migrateTo(...)` wird zuerst dem Agenten durch den Aufruf der Methode `checkSerilization()` (Zeile 4), die Möglichkeit gegeben, sich in einen Zustand zu bringen, in dem ein Transfer sinnvoll ist. Dann erst wird die eigentlich Migration eingeleitet (Zeile 5).

```
1 public void migrateTo( AgentSystemService agentSystem) throws
3   CouldNotMigrate {
4   checkSerialization();
5   _migrate.mobileAgentWantToMigrate(_name,agentSystem);
6 }
```

6.4.3 Klasse AgentApplet

Die Klasse `AgentApplet` ist die abstrakte Basisklasse aller Agenten-Applets. Sie stellt als einzige Methode `initCORBA(...)` zur Verfügung. Diese Methode stellt die Verbindung vom Applet zum Agenten her. Als Beispiel dient der Verbindungsaufbau zwischen dem Applet des IPRouting-Agenten (vgl. 6.5.4) und dem IPRouting-Agenten selbst. Der Ablauf von `initCORBA(..)` wird nachfolgend beschrieben und in Abbildung 6.2 in den Schritten (1) bis (4) verdeutlicht:

```
 1  public org.omg.CORBA.Object initCORBA(String identity)
 2    throws java.net.MalformedURLException, java.io.IOException {
 3    java.net.URL url= this.getDocumentBase(); //get URL of webserver
 4    // create a request to the webserver
 5    String urlname= new String(url.getProtocol()+"://"+
 6                               url.getHost()+":"+
 7                               String.valueOf(url.getPort())+"/"+
 8                               identity+".ior");
 9    java.net.URL urlIOR= new java.net.URL(urlname);
10    java.net.URLConnection urlCon= urlIOR.openConnection();
11    // get the answer from the webserver
12    java.io.BufferedReader in = new java.io.BufferedReader(
13           new java.io.InputStreamReader(urlCon.getInputStream()));
14    String inputLine;
15    String ior=new String();
16    while ((inputLine = in.readLine()) != null) //read IOR
17      ior=ior.concat(inputLine);
18    in.close();
19    // initialize the orb and prevent
20    // connecting visigenic's gatekeeper
21    java.util.Properties prop= new java.util.Properties();
22    prop.put("ORBdisableLocator","true");
23    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this,prop);
24    org.omg.CORBA.Object obj= orb.string_to_object(ior);
25    return obj;
26  }
```

In den Zeilen 3 bis 10 wird eine URL erzeugt und vom Webserver-Agenten angefordert (Abb. Schritt (1)), die folgendes, festgelegte Format hat:
`http://Webserver-Host:Webserver-Port/Identity-Agenten.ior`
ein Beispiel wäre:
`http://sunhegering2:4300/IPRouting.ior`

Das Applet darf diesen Webserver kontaktieren, da es von diesem Webserver auch geladen wurde. Der Webserver sendet daraufhin die IOR des IPRouting-Agenten im Stringformat (Abb. Schritt (4),vgl. 6.5.1 und 6.5.4). Die IOR wird in Zeile 16 und 17 ausgelesen. Der ORB wird mit dem Applet selbst, angezeigt durch `this` und einem `Properties`-Objekt initialisiert (Zeile 23). Dabei muß sichergestellt werden, daß der ORB von Visigenic nicht versucht den *Gatekeeper* (vgl. [Vis97]) zu kontaktieren. Abschließend wird aus der IOR im Stringformat in eine CORBA-Objektreferenz umgewandelt (Zeile 24).

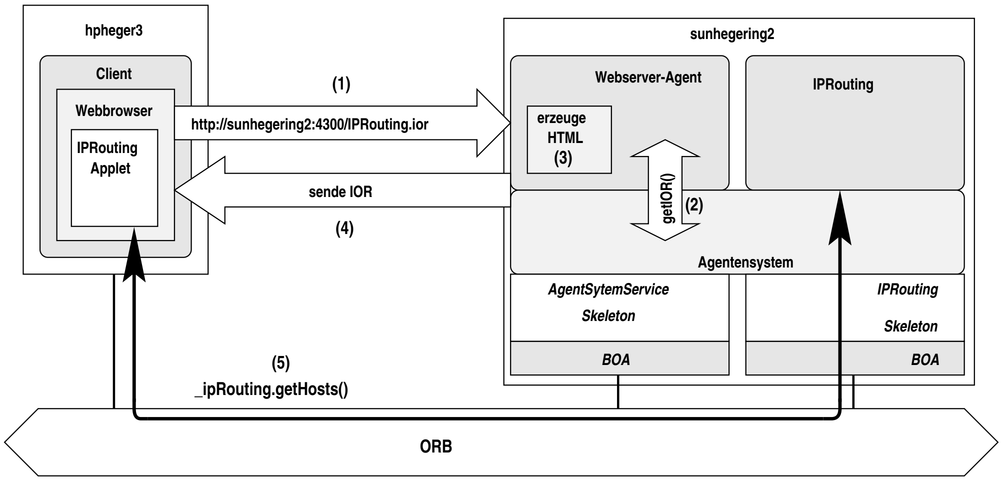

Abbildung 6.2: Verbindungsaufbau von einem Applet zum Agenten

6.5 Implementierung der Agenten

Es wurden zwei Agenten, der Webserver- und der IPRouting-Agent, von der FMA-Architektur auf die MASA-Architektur portiert und erweitert. Beide werden in diesem Abschnitt vorgestellt.

6.5.1 Klasse WebserverStationaryAgent

Der Webserver-Agent erzeugt bei seiner Instanziierung einen Socket auf dem Port, der in der *Property*-Datei angegeben ist. Sollte der Port bereits belegt sein, wird versucht, auf der nächst höheren Portnummer einen Socket zu öffnen usw. Der `WebserverStationaryAgent` wartet nun in seiner `run()`-Methode auf ankommende HTTP-Anfragen und erzeugt pro Anfrage eine Instanz der Klasse `Connection`.

6.5.2 Klasse Connection

Die Klasse `Connection` ist als Thread realisiert und so können mehrere HTTP-Anfragen gleichzeitig abgearbeitet werden.

Methode run()

Es können folgende HTTP-Anfragen bearbeitet werden:

- Homepage des Webservers: es wird die Homepage des Webservers mit Hilfe der `generateLinkTable(...)`-Methode erzeugt.
- Anfrage nach Dateien, z. B. HTML-Files, Bilder (GIF) und Applet-Klassen. Diese werden aus der Datenbasis des Webservers geladen.

- Anfrage nach IOR eines Agenten: wird beim `AgentManager` nachgefordert.

Es ist eine gemeinsame Datenbasis der Webserver-Agenten notwendig, denn die Webseiten der Agenten werden bei ihrer Erzeugung mit angegeben. Soll jetzt ein Agent auf verschiedenen Agentensystemen ablaufen, die keine gemeinsame Datenbasis besitzen, so müßte sich der Pfad der Webseite ständig ändern.

6.5.3 Klasse IPRoutingMobileAgent

Der IPRouting-Agent ist die Zusammenfassung von drei Agenten der FMA-Architektur, AHSInterpreter, AISInterpreter und AIPRSInterpreter (siehe Unterabschnitt 5.5.2). Der IPRouting-Agent erzeugt drei voneinander abhängige Felder, `HostElement[]`, `HostInterfaceElement[]` und `HostIPRoutingElement[]`, die zwischengespeichert werden. Die einzelnen Methoden mußten dahingehend modifiziert werden, daß die Felder aus Vektoren (java.util.Vector) konvertiert werden.

Bei der Beschreibung der Methoden wird vor allem auf die benötigten Ressourcen und Besonderheiten eingegangen.

Methode getHosts()

Bei der Implementierung dieser Methode muß der Aufruf einer `exec()`-Methode vom `AgentSecurityManager` erlaubt werden, da aus dem NIS mit dem Befehl '`/usr/bin/ypcat hosts`' die Hosts ausgelesen werden.

Methode getHostInterfaces() und getIPRouting()

Bei diesen Methoden muß auf jedem Rechner, der in den `HostElement[]` eingetragen ist, ein SNMP-Agent laufen der entweder die HP-Unix-MIB oder die RFC1213-MIB unterstützt. Die Kommunikation mit den SNMP-Agenten wird über die Klassen `SNMPAccess` und `SNMPVector` abgewickelt, die in [Kot97] implementiert wurden. Dabei war der Community String fest im Programm `SNMPAccess` kodiert. Da der IPRouting-Agent ein Applet hat, kann der Community String nun über ein Textfeld einegegeben werden und den Methoden `getHostInterfaces(...)` und `getIPRouiting(...)` als Parameter übergeben werden.

6.5.4 Klasse IPRoutingApplet

Das IPRouting-Applet erbt vom oben beschriebenen `AgentApplet` und wird in den Webbrowser des Client geladen, wie in Abschnitt 4.4 beschrieben.

```
public void init(){
  ...
  try {
```

```
 4      org.omg.CORBA.Object obj= initCORBA("IPRouting");
 5      _ipRouting= IPRoutingHelper.narrow(obj);
 6    }
 7    catch(java.lang.Exception e) {
 8     // error occured during initCORBA()
 9    }
10    ...
11  }
```

In der `init()`-Methode wird die Verbindung zum IPRouting-Agenten durch den Aufruf der Methode `initCORBA(...)` hergestellt (Zeile 4), anschließend folgt das Casting durch die `narrow(...)`-Methode der Helper-Klasse. Bei `_ipRouting` handelt es sich um ein Attribut der Klasse und ist vom Typ `IPRouting`.

Das folgende Programmstück steht innerhalb des *Event Handlers* des getHosts()-*Button* und wird ausgeführt, wenn dieser *Button* gedrückt wird:

```
1  HostElement[] hostElement= null;
2  try {
3    hostElement= _ipRouting.getHosts();
4    for (int i= 0;i < he.length; i++)
5      _resultArea.append(hostElement[i].hostName.toString()+"\n");
6  }
7  catch(ResourceException re){
8     _resultArea.append("ResourceException\n");
9  }
```

Über die CORBA-Objektreferenz `_ipRouting` kann jetzt die Methode `getHosts()` auf dem IPRouting-Agenten ausgeführt werden (Zeile 3 und Abb. 6.2 Schritt (5)). Das Ergebnis wird in der Variable `hostElement` gespeichert und in der *Textarea* `_resultArea` zur Anzeige gebracht (Zeile 5). Sollte ein Fehler beim Erstellen des `HostElement`-Feldes aufgetreten sein, wird die Exception im catch-Block (Zeile 7) abgefangen und in der *Textarea* angezeigt.

Eingebunden wird das IPRouting-Applet in eine entsprechende HTML-Seite:

```
 1  <html>
 2  <head>
 3  <title>HOMEPAGE des IPRouting-Agenten</title>
 4  </head>
 5  <h1 align=center>
 6  <applet codebase=/proj/fagent/masa_0.2/classes/
 7  code=de.unimuenchen.informatik.mnm.masa.agent.\
         iprouting.IPRoutingApplet.class
 8  width=640 height=250
 9  >
10  </applet>
11  </h1>
12  </html>
```

Dabei wird in den Zeilen 6 bis 9 das Applet aufgerufen. Die Codebase Der Webserver muß auf die *Codebase* Zugriff haben.

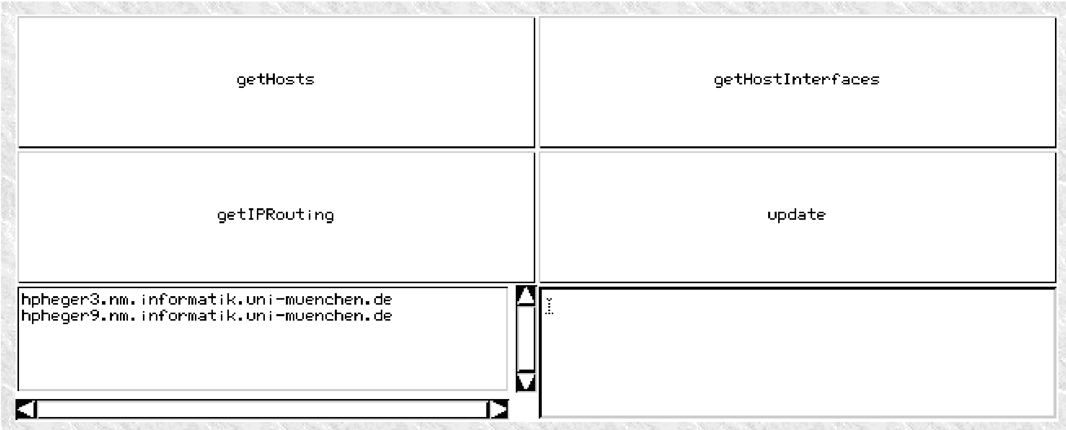

Abbildung 6.3: Das Applet des IPRouting-Agenten

6.6 Klassen des tools Package

Die Klassen des tools Package sind Hilfsklassen und lassen sich in die anderen Packages nicht sinnvoll eingliedern.

6.6.1 Klasse NameWrapper

Diese Klasse wird aus folgendem Grund benötigt:
Der '`struct Name`' aus der MAF-Spezifikation wird durch Java-Mapping zu '`public final class Name`'. Dadurch kann keine Unterklasse von der Klasse `Name` gebildet werden. Da der `Name` als Schlüssel für die Hashtabelle des `AgentManager` dienen soll, müssen zwei Methoden, `equals()` und `hashCode()`, implementiert werden. Da das nicht in der Klasse `Name` geschehen kann, werden die Methoden in der Klasse `NameWrapper` implementiert. Die Klasse `NameWrapper` wird mit einer Instanz der Klasse `Name` erzeugt.

6.6.2 Klasse Debug

Diese Klasse dient zum Protokollieren von Ereignismeldungen. Die Klasse besitzt keinen Konstruktor, sondern die Attribute werden beim Laden der Klasse durch ein `static`-Konstrukt initialisiert.

Methode log(...)

Folgende Ereignismeldungen werden je nach *Property*-Einstellungen (siehe Unterabschnitt 6.8.2) in eine Datei und auf die Standardausgabe geschrieben. Dieselben Ereignismeldungen werden auch in den Standard Event Channel gesendet:

- Start eines Agentensystems
- Terminierung eines Agentensystems
- Erzeugung eines Agenten
- bei der Erzeugung eines Agenten trat ein Fehler auf
- Transfer eines Agenten
- beim Transfer eines Agenten trat ein Fehler auf
- Terminierung eines Agenten

Jede Ereignismeldung hat folgendes Format:

`Datum Uhrzeit Hostname Agentensystemname: Meldung`

und entspricht damit dem Format, wie in UNIX Logdateien aufgebaut sind, z. B. syslog.

6.7 Einbindung neuer Agenten

Die Einbindung eines neuen Agenten wird anhand des IPRouting-Agenten detailliert erklärt.

1. Es muß die Entscheidung getroffen werden, ob der Agent ein `MobileAgent` oder ein `StationaryAgent` sein soll. Es sollte, falls es keinen zwingenden Grund für einen stationären Agenten gibt, sich für einen Mobilen Agenten entschieden werden. Das Agentensystem hat dann die Möglichkeit, falls der Rechner z. B. abgeschaltet wird, den Agenten zu transferieren. Im Beispiel des IPRouting-Agenten ist die Entscheidung auf `MobileAgent` gefallen.
2. Es muß eine IDL-Schnittstelle (*IPRouting*) spezifiziert werden, die die IDL-Schnittstelle *Migration* erweitert:
   ```
   #include "Migration"
   interface IPRouting : agent::Migration {
     ...
   };
   ```
3. Mit dem Programm `idl2java` müssen die Java-Schnittstellen, -Stubs und -Skeletons generiert werden.
4. Es wird die Klasse `IPRoutingMobileAgent` angelegt, die folgenden Kopf besitzt:
   ```
   public class IPRoutingMobileAgent
      extends MobileAgent
      implements IPRoutingOperations
   ```
5. Bei jedem Attribut der Klasse ist festzulegen, ob bei einer Serialisierung des Agenten, diese Attribut neu belegt werden muß, oder nicht. Attribute,

die nicht transferiert werden müssen, werden als `transient` gekennzeichnet. Gibt es solche Attribute, muß man nach dem Transfer selbst die Neubelegung der Attribute vornehmen.

Bei den anderen Attributen muß dafür gesorgt werden, daß sie serialisierbar sind, indem bei der Klasse des Attributs
```
implements java.io.Serializable
```
angefügt wird.

6. Die Methoden der Schnittstelle `IPRoutingOperations` müssen implementiert werden.

7. Die Methoden `cleanUp()` und `checkSerialization()` aus der Basisklasse `Agent` müssen implementiert werden.

Bei der Implementierung ist darauf zu achten, daß keine propietären Funktionen wie sie z. B. Visigenic anbietet, (`extensible structs`, `Helper.bind()` usw.) verwendet werden. Denn die Agenten sollen mit ORBs verschiedener Hersteller, ohne vorherige Portierung, funktionieren.

6.8 Installation

In diesem Abschnitt wird die Übersetzung des Quellcodes, die Konfiguration des Agentensystems, sowie der Start des Agentensystems beschrieben.

6.8.1 Übersetzung des Quellcodes

Zur Übersetzung des Quellcodes wird ein Makefile (siehe Anhang C) verwendet. Es dient sowohl zur Übersetzung der IDL-Dateien, als auch der Java-Dateien. Im ersten Teil des Makefiles werden die Variablen gesetzt. Die wichtigsten Variablen sind:

- `JDKROOT`: Pfad zum JDK
- `VISIBROKER_ROOT`: Pfad zum Visibroker
- `ROOT`: Projektverzeichnis
- `OWN_PACKAGE`: Packagename des eigenen Projekts

Mit dem Aufruf `gmake ALL` wird das gesamte Projekt übersetzt.

6.8.2 Konfiguration des Agentensystems

Um die Konfiguration für den Benutzer, aber auch für den Implementierer so einfach wie möglich zu gestalten, wird eine *Property*-Datei angelegt, in der die Konfigurationsdaten eingetragen werden. Java unterstützt das Auslesen von Werten aus einer *Property*-Datei. Die einzelnen *Properties* werden in Anhang B detailliert erklärt. Beim Start des Agentensystems wird die *Property*-Datei als Kommandozeilen-Argument angegeben.

6.8.3 Start des Agentensystems

Es existiert von Visigenic ein Startskript `vbj`, welches die Verbindung zum *Smart Agent* mit einem UDP-Broadcast herstellt. Anschließend wird die Java Virtual Machine aufgerufen.

Folgenden Voraussetzungen müssen erfüllt sein, damit das Agentensystem gestartet werden kann, dabei muß die Umgebungsvariable `OSAGENT_PORT` überall gleich gesetzt sein:

1. der *Smart Agent* wird mit dem Aufruf `osagent` gestartet
2. der Naming Service muß gestartet werden:
```
/usr/local/mnmcommon/vbroker-3.0/bin/vbj              \
  -DORBservices=CosNaming                              \
  com.visigenic.vbroker.services.CosNaming.ExtFactory \
  myFactory  /tmp/naming.log
```
3. die Komponente, auf der das Agentensystem ausgeführt werden soll muß eine IP-Adresse haben (vgl. 4.5.1)
4. eine *Property*-Datei, wie sie in Anhang B angegeben ist, muß lesbar sein. Damit das Agentensystem auf die Datei zugreifen kann wird der Dateiname als Argument im Kommandozeilenaufruf mit
`-Dmasa.propfile=/myPath/myPropertyFile`
übergeben.
5. Start des Event Channels (ist nicht zwingend erforderlich):
```
/usr/local/mnmcommon/vbroker-3.0/bin/vbj             \
-DORBservices=CosNaming -DSVCnameroot=myFactory      \
-classpath                                           \
/proj/java/jdk1.1-solaris/jdk1.1.6/lib/classes.zip:  \
/usr/local/mnmcommon/vbroker-3.0/lib/vbjcosnm.jar:   \
/usr/local/mnmcommon/vbroker-3.0/lib/vbjcosev.jar:   \
/usr/local/mnmcommon/vbroker-3.0/lib/vbj30.jar:      \
/proj/fagent/masa_0.2/classes                        \
de.unimuenchen.informatik.mnm.masa.event.AgentChannel\
-Dmasa.propfile=/proj/fagent/masa_0.2/masa.properties
```
6. Start des Agentensystems:
```
/usr/local/mnmcommon/vbroker-3.0/bin/vbj                         \
-DORBservices=CosNaming -DSVCnameroot=myFactory                  \
-classpath                                                       \
/proj/java/jdk1.1-solaris/jdk1.1.6/lib/classes.zip:              \
/usr/local/mnmcommon/vbroker-3.0/lib/vbjcosnm.jar:               \
/usr/local/mnmcommon/vbroker-3.0/lib/vbjcosev.jar:               \
/usr/local/mnmcommon/vbroker-3.0/lib/vbj30.jar:                  \
/proj/fagent/masa_0.2/classes:/usr/local/mnmcommon/lib/advent:\
```

```
/proj/evcorr/public-htdocs/prototype-0.3/classes:              \
/users/stud/radisic/Diplom/radi98/Sourcen/classes/:            \
/users/stud/coehn/proto/classes/                               \
de.unimuenchen.informatik.mnm.masa.agentSystem.AgentSystem     \
-Dmasa.propfile=/proj/fagent/masa_0.2/masa.properties
```

Es existiert ein Skript zum Starten und Stoppen des Agentensystems mit dem Namen `masaScript`.

Kapitel 7

Zusammenfassung und Ausblick

Es wurde eine Agentenarchitektur entworfen und implementiert, die folgende wesentliche Komponenten besitzt:

- Der **Agent**, welcher mobil, autonom und kooperativ seine Managementaufgabe erfüllt.
- Die **Laufzeitumgebung** des Agenten, die als wesentliche Aufgaben das Management und die Migration von Agenten hat.
- Jede Laufzeitumgebung hat einen **Webserver**, der die GUIs der Agenten über HTTP bereitstellt.
- Die **Kommunikationsinfrastruktur**, welche die Kommunikation zwischen allen kommunizierenden Komponenten bereitstellt, sowie den Agententransfer unterstützt. Es wurde CORBA als Kommunikationsinfrastruktur verwendet.
- Der **Verzeichnisdienst**, welcher die Identifizierung und Lokalisierung von Agenten und Laufzeitumgebung unterstützt, wurde durch den Naming Service von CORBA realisiert.
- Der **Ereignisdienst**, der die asynchrone Kommunikation ermöglicht, wurde durch den CORBA Event Service realisiert.

Die Agentenarchitektur implementiert im wesentlichen die *Mobile Agent Facility (MAF)* Spezifikation, einem Standard, der von der *Object Management Group (OMG)* herausgegeben wurde. Die MAF-Spezifikation ist ein sehr allgemein gehaltener Standard, der vor allem mit dem Ziel entwickelt wurde, die Interoperabilität von Agentensystemen zu erzielen. Es mußten deswegen zusätzlich noch Schnittstellen definiert werden, z. B. zwischen Agent und Laufzeitumgebung. Eine Standardisierung auch dieser Schnittstelle wäre wünschenswert, damit Agenten nicht nur zwischen Agentensystemen verschiedener Hersteller transferiert werden können, sondern auch die Dienste des Agentensystems voll nutzen können.

CORBA wurde als Kommunikationsinfrastruktur verwendet, da es die Verteilung und die Kommunikation der Agenten unterstützt, sowie durch die IDL-Schnittstelle der Agent in seiner Funktionalität festgelegt ist.

Als Implementierungssprache wurde Java, sowohl für die Agenten, als auch für die Laufzeitumgebung verwendet. Java ist eine objektorientierte, interpretierte Programmiersprache die für viele Plattformen erhältlich ist. Die Serialisierung wird durch Java sehr gut unterstützt.

Die Haupteigenschaften der Agentenarchitektur sind:

- Bereitstellung eines Basisagenten sowie Basis-Applets für Programmierer
- Unterstützung der Migration von Agenten
- Eine vom Hersteller des ORBs unabhängige Implementierung der Architektur

Agenten sind durch ihre Modularität, Verteilbarkeit und Kommunikationsfähigkeit für das Netz- und Systemmanagement geeignet. Mit der Eigenschaft der Mobilität können einige Managementaufgaben jetzt effizient gelöst werden. Es ergibt sich aber aus der Mobilität der Agenten ein gravierendes Problem:
Clients, die von einem Mobilen Agenten eine CORBA-Objektreferenz halten, können nie sicher sein, daß diese Referenz noch gültig ist, da der Mobile Agent inzwischen migriert sein könnte.

Erfahrungsbericht

Als Software-Entwicklungsmethode wurde OMT verwendet. OMT unterscheidet im Objektmodell nicht zwischen Klassen und Schnittstellen. Es fehlt damit auch die entsprechende Assoziation, die ausdrückt, daß eine Klasse eine Schnittstelle implementiert. Das wirkt sich negativ auf das Objektdiagramm aus, da Schnittstellen durch einen Kommentar gesondert gekennzeichnet werden müssen, sowie bei der Codegenerierung, da nur Klassen und keine Schnittstellen erzeugt werden.

Java eignet sich gut als Implementierungssprache für Agenten, da vor allem die Serialisierung optimal unterstützt wird. Dadurch, daß Java keine Adreßarithmetik zuläßt, werden im allgemeinen schnellere Entwicklungszeiten von Programmen erreicht. Java unterstützt keine Mehrfachvererbung, was man beim Design des Objektmodells von Anfang an beachten muß. Der Tie-Mechanismus wird nur wegen der Einfachvererbung von Java benötigt.

Ausblick

Es müssen noch folgende Interpreter der FMA-Architektur portiert werden:

- Event Monitoring Agent (EMAMobileAgent): zeigt alle auftretenden Events an
- Machine Monitoring Agent (MMAMobileAgent): überwacht Hosts über ICMP
- Process Monitoring Agent (PMAMobileAgent): überwacht Prozesse über SNMP-MIB
- Web Monitoring Agent (WMAMobileAgent): überwacht Webserver mittels einer HTTP-Anfrage
- State Monitoring Agent (SMAMobileAgent): überwacht Zustandsübergänge von Events

Hauptaufgabe ist die Definition einer IDL-Schnittstelle für jeden Agenten und die Implementierung der angebotenen Operationen, wobei Teile der Klassen der FMA-Architektur wiederverwendet werden können. Die zugehörigen Applets können direkt übernommen werden, nur die CORBA-Anbindung muß ergänzt werden.

Um die Gruppierung von Agenten zu unterstützten, müßte ein Gruppierungsdienst entwickelt werden, welcher eventuell auf dem CORBA Naming Service oder Topology Service aufbaut.

Um eine Gesamtübersicht des Managementsystems zu bekommen, muß der Naming Service herangezogen werden. Der Naming Service könnte in einen Agenten gekapselt werden, der dann eine graphische Übersicht bereitstellt.

Der exklusive Zugriff auf Ressourcen ist für einige Agenten von Bedeutung. Zur Durchsetzung dieser Anforderung könnten Policies eingesetzt werden, die bei der Erzeugung und Migration des Agenten angegeben werden. Das Agentensystem muß die Einhaltung der Policies überwachen.

Wie in der Arbeit dargestellt, wird, nachdem ein Agent migriert ist, ein neuer Thread gestartet und die `run()`-Methode von Beginn an abgearbeitet. Um einen Agenten an der Stelle auf dem Zielagentensystem weiter auszuführen, bei der er auf dem Quellagentensystem angehalten wurde, müßten zusätzlich noch die Register, der Programmzähler und der Executionstack mit transferiert werden. Die JVM verbietet den Zugriff auf diese Informationen. Es müßte über die JVM eine Ausführungsumgebung gelegt werden, die diese Informationen simuliert und bei der Migration des Agenten mitsendet.

Anhang A

Abkürzungen

ACL	Access Control List
AHS	Agent Host Server
AIPRS	Agent IP-Routing Server
AIS	Agent Interface Server
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
FMA	Flexible Management Agent
GUI	Graphical User Interface
GIOP	General Inter-ORB Protocol
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
IP	Internet Protocol
JAT	Java Agent Template
JDK	Java Development Kit
JVM	Java Virtual Machine
KQML	Knowledge Query and Manipulation Language
M2M	Manager-to-Manager MIB
MAF	Mobile Agents Facility
MASA	Mobile Agent System Architecture
MbD	Management by Delegation
MI	Management Information
MIB	Management Information Base
MO	Managed Object
NIS	Network Information Service

OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modelling Techinque
ORB	Object Request Broker
POA	Portable Object Adapter
QoS	Quality of Service
RFC	Request for Comment
RMI	Remote Method Invocation
RMON	Remote Networking Monitoring MIB
RPC	Remote Procedure Call
StP	Software through Pictures
SNMP	Simple Network Management Protocol
SNMPv2	Simple Network Management Protocol Version 2
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifiers
URL	Uniform Resource Locators
WWW	World Wide Web

Anhang B

Properties

Es folgt eine Beschreibung aller Properties, wobei alle Properties den Präfix 'de.unimuenchen.informatik.mnm.masa.' besitzen. In Klammern ist der Datentyp der Property angegeben.

- log.display (boolean): Anzeige der Logmeldungen auf der Standardausgabe
- log.hasFile (boolean): Ausgabe der Logmeldungen in eine Datei
- log.filename (String): Name der Logdatei mit vollständigem Pfad
- tieprefix (String): Präfix der der aus IDL generierten Tie-Klassen
- interface (String): Postfix der aus IDL generierten Java-Schnittstellen
- agentSystem.type (short): Typ des Agentensystems
- agentSystem.authority (short): Authority des Agentensystems
- webserverUp (boolean): Webserver wird automatisch beim Start des Agentensystems hochgefahren
- webserverPort (int): Nummer des Webserver-Ports
- channel (String): Name des Standard Event Channel

Anhang C

Makefile

```
#
# Java-Stuff
#
JDKROOT       = /proj/java/jdk1.1-solaris/jdk1.1.6
JAVAC_OPTIONS = -depend -deprecation
JAVAC         = ${JDKROOT}/bin/javac
JAVACC        = ${JAVAC} ${JAVAC_OPTIONS} -d ${LOCALCLASSES} \
                -classpath ${SRC}:${JAVA_CLASSPATH}
JAVAVM        = ${JDKROOT}/bin/java

#
# Visibroker-Stuff
#
VISIBROKER_ROOT  =  /usr/local/mnmcommon/vbroker-3.0
VISIBROKER_PATH  =  ${VISIBROKER_ROOT}/bin
VISIBROKER_VM    =  ${VISIBROKER_PATH}/vbj

#
# IDL-Stuff
#
IDL2JAVA_OPTIONS =  -no_examples -strict
IDL2JAVA         =  ${VISIBROKER_PATH}/idl2java
PACKAGE          =  de.unimuenchen.informatik.mnm.masa
PACKAGE_SOURCE   =  de/unimuenchen/informatik/mnm/masa
OWN_PACKAGE      =  de.unimuenchen.informatik.mnm.masa

#
# Project-Stuff
#
ROOT          = /proj/fagent/masa_0.2
ADVENTCLASSES = /usr/local/mnmcommon/lib/advent
EVCORRCLASSES = /proj/evcorr/public-htdocs/prototype-0.3/classes
WORKDIR       = /tmp
ARGV          = -Dmasa.propfile=${ROOT}/masa.properties

#
```

```
# Do _NOT_ edit this section
#
SRC          = ${ROOT}/src
LOCALCLASSES =  ${ROOT}/classes
IDL_ROOT     = ${SRC}/idl

JAVA_CLASSPATH = \
${JDKROOT}/lib/classes.zip:${VISIBROKER_ROOT}/lib/vbjcosnm.jar:      \
${VISIBROKER_ROOT}/lib/vbjcosev.jar:${VISIBROKER_ROOT}/lib/vbj30.jar:\
${LOCALCLASSES}:${ADVENTCLASSES}:${EVCORRCLASSES}:                   \
/users/stud/radisic/Diplom/radi98/Sourcen/classes/:                  \
/users/stud/coehn/proto/classes/

#
# Compile Java-Source
#
ALL: idl_all all
all: src

src: omg agentSystem agent tools event test
omg: cfmaf notify trade
AGENTS = pma foo iprouting webserver
agent: shared ${AGENTS}

cfmaf:
        ${JAVACC} ${SRC}/CfMAF/MAFAgentSystem.java
notify:
        ${JAVACC} ${SRC}/CosNotification/*.java
trade:
        ${JAVACC} ${SRC}/CosTrading/*.java
agentSystem:
        ${JAVACC} ${PACKAGE_SOURCE}/agentSystem/AgentSystem.java
tools:
        ${JAVACC} ${PACKAGE_SOURCE}/tools/*.java
event:
        ${JAVACC} ${PACKAGE_SOURCE}/event/AgentChannel.java
shared:
        ${JAVACC} ${PACKAGE_SOURCE}/agent/*.java
test:
        ${JAVACC} ${SRC}/client/*.java
${AGENTS}:
        ${JAVACC} ${PACKAGE_SOURCE}/agent/$@/*.java

#
# Compile IDL-Stuff
#

idl_all: idl idl_omg
idl: idl_agentSystem idl_migration idl_agent idl_agents
idl_omg: idl_mafAgentSystem idl_notify idl_trade

idl_mafAgentSystem:
        ${IDL2JAVA} ${IDL2JAVA_OPTIONS} \
```

```
        ${IDL_ROOT}/MAFAgentSystem.idl
idl_notify:
        ${IDL2JAVA} ${IDL2JAVA_OPTIONS} \
        ${IDL_ROOT}/CosNotification.idl
idl_trade:
        ${IDL2JAVA} ${IDL2JAVA_OPTIONS} \
        ${IDL_ROOT}/CosTrading.idl
idl_agentSystem:
        ${IDL2JAVA} ${IDL2JAVA_OPTIONS} -package ${OWN_PACKAGE}\
        -idl2package ::CfMAF CfMAF                             \
        -idl2package ::agent  ${PACKAGE}.agent                 \
        ${IDL_ROOT}/AgentSystemService.idl
idl_agent: idl_agentService idl_migration
idl_agentService:
        ${IDL2JAVA} ${IDL2JAVA_OPTIONS} -package ${OWN_PACKAGE}\
        -idl2package ::agentSystem ${PACKAGE}.agentSystem      \
        -idl2package ::CfMAF CfMAF                             \
        ${IDL_ROOT}/AgentService.idl
idl_migration:
        ${IDL2JAVA} ${IDL2JAVA_OPTIONS} -package ${OWN_PACKAGE}\
        -idl2package ::agentSystem ${PACKAGE}.agentSystem      \
        -idl2package ::CfMAF CfMAF                             \
        ${IDL_ROOT}/Migration.idl

idl_agents = FOO PMA IPRouting Webserver
${idl_agents}:
  ${IDL2JAVA} ${IDL2JAVA_OPTIONS} -package ${OWN_PACKAGE}.agent\
  -idl2package ::agentSystem ${PACKAGE}.agentSystem            \
  -idl2package ::agent  ${PACKAGE}.agent                       \
  -idl2package ::CfMAF CfMAF                                   \
  ${IDL_ROOT}/$@.idl
```

Literaturverzeichnis

[Agl] AGLETS: http://www.trl.ibm.co.jp/aglets.

[Aon96] AONIX: *StP Core - Fundamentals of StP Release 2.4.* Technischer Bericht 1996.

[Che96] CHEONG, F.: *Spiders, Wanderers, Brokers, and 'Bots.* New Riders Publishing, 1996.

[COR97] *CORBAservices: Common Object Service Specification.* Technischer Bericht, Object Management Group, 1997.

[COR98] *The Common Object Request Broker: Architecture and Specification.* Technischer Bericht, Object Management Group, 1998.

[DCE] DCE. http://www.dstc.edu.au/AU/research_news/dce/dce.html.

[DCO] DCOM. http://www.microsoft.com/com/dcom-f.htm?RLD=59.

[Fla97] FLANAGAN, D.: *Java in a Nutshell.* O'Reilly, zweite Auflage, 1997.

[Fro97] FROST, R.: *The Java Agent Template v0.3.* http://cdr.stanford.edu/ABE/JavaAgent.html, 1997.

[GMD97] GMD FOKUS: *Mobile Agents System Facility.* Technischer Bericht, Object Management Group, 1997.

[Gra] GRASSHOPPER. http://www.ikv.de/products/grasshopper/grasshopper.html.

[HA93] HEGERING, H.-G. und S. ABECK: *Integriertes Netz- und Systemmanagement.* Addison-Wesley, 1993.

[Hol97] HOLLERITH, A.: *Entwurf einer Architektur für flexible Agenten auf der Basis des Konzepts von Management by Delegation.* Diplomarbeit, Technische Universität München, Februar 1997.

[Kem97] KEMPTER, B.: *Realisierung eines Managementwerkzeugs in Java für das Management von IP-Netzen.* Technischer Bericht, Technische Universität München, 1997.

[Kot97] KOTSELIDIS, T.: *Erweiterung eines Java-Agenten um Managmentfunktionenen.* Technischer Bericht, Technische Universität München, 1997.

[MB97] Mountzia, M.-A. und D. Benech: *Communication Requirements and Technologies for Multi-Agent Management Systems.* In: *Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, Sydney, Australia, Oktober 1997.

[MOA] MOA. http://www.camb.opengroup.org/RI/java/moa/index.htm.

[Mou97a] Mountzia, M.-A.: *Flexible Agents in Integrated Network and Systems Management.* Doktorarbeit, Technische Universität München, Dezember 1997.

[Mou97b] Mountzia, M.-A.: *Verteiltes und flexibles Management mit erweiterbaren, kooperierenden Agenten.* In: *Proceedings of the 27th GI-Jahrestagung*, Aachen, September 1997.

[OHE97] Orfali, R., D. Harkey und J. Edwards: *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, 1997.

[Orf98] Orfali, R.: *Client/Server Programming with Java and CORBA.* John Wiley & Sons, zweite Auflage, 1998.

[OSF93] *Osf Dce Administration Guide-Core Components.* Open Software Foundation, 1993.

[Ros94] Rose, M.: *The Simple Book.* Prentice Hall, zweite Auflage, 1994.

[Rum93] Rumbaugh, J. et al.: *Objektorientiertes Modellieren und Entwerfen.* Hanser, 1993.

[SCP96] Schwartz, R., T. Christiansen und S. Potter: *Perl.* O'Reilly, 1996.

[Ses97] Sessions, R.: *COM and DCOM: Microsoft's Vision for Distributed Objects.* John Wiley & Sons, 1997.

[Tan97] Tanenbaum, A. S.: *Computernetzwerke.* Prentice Hall, dritte Auflage, 1997.

[tel96] *Telescript Technology: the foundation for the electronic marketplace, General Magic Inc.* http://www.genmagic.com/agents/Whitepaper/whitepaper.html, 1996.

[Vis97] Visigenic: *Visibroker for Java 3.0: Reference Manual*, 1997.

[Wen97] Wennrich, M.: *Erstellung eines Kommunikationsmodells für kooperierende Agenten für das Management von verteilten Systemen.* Diplomarbeit, Technische Universität München, Februar 1997.

[ZP98] Zeltserman, D. und G. Puoplo: *Building Network Management Tools With Tcl/Tk.* Prentice Hall, 1998.