Architektur eines sicheren Mobile-Agenten-Systems für das Netzmanagement

Frank Joachim Leitner

Lehrstuhl für Datenverarbeitung

Architektur eines sicheren Mobile-Agenten-Systems für das Netzmanagement

Frank Joachim Leitner

Vollständiger Abdruck der von der Fakultät für Elektrotechnik
und Informationstechnik der Technischen Universität München
zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr.-Ing. U. Wagner
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. techn. J. Swoboda (i.R.)
	2.	Univ.-Prof. Dr.-Ing. G. Färber

Die Dissertation wurde am 22.01.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 03.06.2003 angenommen.

Kurzfassung

Die Komplexität und Größe heutiger Netzwerke verlangt nach neuen Architekturen für das technische Netzmanagement. Das klassische Client-Server-Modell skaliert nicht ausreichend und ist wenig flexibel. Ein vielversprechender neuer Ansatz ist das Mobile-Agenten-Paradigma: Mobile Agenten sind Programme, die sich in einem Computernetz autonom bewegen und die dabei gesammelten Daten mitführen. In vielen Fällen erfüllen sie ihre Aufgaben besser als herkömmliche Architekturen, sie erzeugen weniger Netzlast und benötigen weniger Systemressourcen. Durch ihre Autonomie sind sie unabhängig von der Netzverbindung zum Manager.

Neben den Vorteilen birgt das neue Modell allerdings hohe potenzielle Gefahren. Die beteiligten Geräte müssen zur Laufzeit unbekannten Code akzeptieren, der auf das System zugreift. Schädliche Agenten könnten dies nutzen, um die Plattform anzugreifen; auch andere Agenten und das gesamte Netz sind gefährdet. Selbst Agenten aus vertrauenswürdiger Quelle können durch Modifikation des Codes oder ihrer Daten schädlich werden. Ein Blick auf das Szenario lässt erkennen, dass keine zentrale Instanz existiert, die eine Sicherheits-Policy vorgeben und durchsetzen könnte. Diese Arbeit verfolgt daher das Modell des aktiven Selbstschutzes der Agenten und der Plattform, die die Agenten ausführt.

Eine Analyse der relevanten Bedrohungen im Netzmanagement-Umfeld zeigt, welche Dienste die Sicherheitsarchitektur bereitstellen muss, um einen effektiven Selbstschutz zu ermöglichen. Wichtige Aspekte sind die Kapselung der Agenten, die Identifizierung und Authentifizierung der Agenten und ihrer Benutzer während der Kommunikation sowie die sichere Migration. Als besondere Probleme sind die sichere Identifizierung des Aufrufers einer Methode zu lösen sowie die zuverlässige Trennung der Klassen bei gleichzeitiger Minimierung des Overheads durch unnötig mehrfach vorhandene, identische Klassen. Ein eigener Security Manager sorgt für den Grundschutz, zudem wird eine plattformzentrale Rechteverwaltung implementiert.

Sicherheitsarchitekturen sind immer abhängig von dem System, das sie schützen. Ausgangspunkt war der Prototyp eines Mobile-Agenten-Systems, der jedoch keinerlei Sicherheitsfunktionen enthielt. Dieser Prototyp wurde ergänzt und an zentralen Stellen stark modifiziert. Die Arbeit stellt das resultierende System vor und bewertet relevante Aspekte bezüglich ihrer Sicherheitsfolgen. Als Maßstab dienen das Anforderungsprofil des Netzmanagements und die daraus abgeleiteten Designkriterien. Durch Modularisierung und konsequente Anwendung des Agentenparadigmas bleiben die zentralen Komponenten der Sicherheitsarchitektur trotz ihrer Integration flexibel und austauschbar.

Die erreichte Flexibilität zeigt sich unter anderem bei der Authentifizierung: Sie ist durch Agenten-Plugins realisiert, aufgeteilt in Authentifizierungs-Client und -Server. Ohne Modifikation am Agenten können damit neue Mechanismen implementiert und genutzt werden. Die Plugins funktionieren lokal und netzweit über Plattformgrenzen hinweg. Dies wird durch den Entwurf der Kommunikationsprotokolle als Java-Objektstrom möglich. Die Protokolle sind ebenfalls modular, zudem mehrschichtig und kompatibel zu HTTP, die Plattform muss nur einen Server-Port anbieten. Die Protokolle können mit SSL/TLS kryptografisch gesichert werden.

Die komplette Architektur wurde im Rahmen des MobMan-Projekts (Mobile Manager) in Java implementiert. Durch eine Reihe von Anwendungen im Netzmanagement sowie im Informationsmanagement konnte die Praxistauglichkeit bestätigt werden.

Inhaltsverzeichnis

1 Einführung und Motivation 1

1.1 Aufgaben 2
1.2 Ansätze 3
1.2.1 Der klassische Ansatz 3
1.2.2 Dezentrales Management 5
1.2.3 Webbasiertes Management 6
1.2.4 Web Services 6
1.3 Der neue Ansatz 7
1.3.1 Management mit mobilen Agenten 7
1.3.2 Vorteile im Netzmanagement 8
1.3.3 Integration 9
1.3.4 Probleme 9
1.4 Überblick 10

2 Mobile Agenten 11

2.1 Grundlagen mobiler Agenten 11
2.1.1 Mobiler Code 13
2.1.2 Fähigkeiten von Agenten 15
2.1.3 Anforderungen an die Infrastruktur 15
2.1.4 Potenzielle Vorteile 17
2.2 Sicherheit 18
2.2.1 Schutz der Plattformen vor den Agenten 20
2.2.2 Schutz der Agenten voreinander 20
2.2.3 Schutz der Agenten vor den Plattformen 20
2.2.4 Schutz einer Gruppe von Plattformen vor einem Agenten 21
2.2.5 Weitere 21
2.3 Bestehende Agentensysteme 22
2.3.1 Standards für mobile Agenten 22
2.3.2 Kommerzielle Agentensysteme 23
2.3.3 Forschungsprojekte 23

3 Bedrohungen 25
3.1 Alte Annahmen werden ungültig 25
3.1.1 Zuordnung von Personen zu Programmen (Identitätsannahme) 26
3.1.2 Trojanische Pferde sind selten 26
3.1.3 Quelle der Angriffe 26
3.1.4 Programme bleiben, wo sie sind 27
3.1.5 Weitere geänderte Rahmenbedingungen 27
3.2 Beteiligte Instanzen 27
3.3 Angreifer 29
3.4 Relevanz für Anwendungen im Netzmanagement 30
3.5 Schutzmechanismen 32
3.5.1 Hintergrundfragen 32
3.5.2 Konfigurationssprache 33
3.5.3 Selbstschutz 33
3.6 Erforderliche Barrieren 34

4 Architektur 37
4.1 Designkriterien 38
4.2 Sprache 39
4.3 Klassenmodell 40
4.4 Plattform 42
4.5 Agenten und ihre Komponenten 44
4.5.1 Agenten 46
4.5.2 Plugins 47
4.5.3 Properties 49
4.5.4 Ergebnisse 50
4.5.5 Wichtige Systemagenten 51
4.6 Migration 55
4.6.1 Migration zwischen Plattformen 55
4.6.2 Migration auf die erste Plattform 57
4.7 Kommunikation 57
4.7.1 Aufgabenstellung 58
4.7.2 Events 58
4.7.3 Klassifikation von Nachrichten 60
4.7.4 Zustellungsmechanismen für Nachrichten 60
4.7.5 Realisierung der Nachrichten in MobMan 62
4.8 Protokolle 64
4.8.1 MAP: MobMan Application Protocol 66
4.8.2 WAMP: WILMA Agent Migration Protocol 66
4.8.3 WRAP: WILMA Remote Administration Protocol 69
4.8.4 RAMP: Remote Agent Messaging Protocol 71
4.8.5 Beurteilung der Protokolle 72

5 Java-Spezifika 73
5.1 Java-Sicherheitsmodell 73
5.1.1 Überblick 74
5.1.2 Die Sandbox 75
5.1.3 Schritte der Sicherheitsprüfung 75

5.2 Beschränkungen von Java ... 77
5.3 Trennung der Klassen ... 78
5.3.1 Getrennte Laufzeitumgebungen ... 78
5.3.2 Verschiedene Classloader ... 79
5.3.3 Verschiedene Classloader bei Erkennung identischer Klassen ... 79
5.3.4 Sonderbehandlung von Archiven ... 80
5.3.5 Sonderbehandlung spezieller Klassen ... 80
5.3.6 Zusammenfassung ... 81
5.4 Trennung der Agenten: keine Referenzen ... 81
5.5 Agenten und ihre IDs/Agent Properties ... 82
5.6 Caller-ID ... 83
5.6.1 Problem der Identifikation des Aufrufers einer Methode ... 83
5.6.2 Lösungsansätze ... 83
5.6.3 Nachrichten an andere Plattformen ... 90
5.7 Security Manager ... 90
5.7.1 Überblick ... 90
5.7.2 Realisierung ... 90
5.7.3 Resource Limits ... 91

6 Sicherheitsmechanismen ... 93

6.1 Überprüfung neuer Agenten ... 93
6.2 Authentifizierung ... 95
6.2.1 Grundidee ... 95
6.2.2 Authentifizierungs-Client und -Server ... 96
6.2.3 Plattforminterne Kommunikation ... 96
6.2.4 Externe Kommunikation ... 100
6.2.5 Konfiguration ... 102
6.2.6 Authentifizierung beim Web Agent ... 104
6.3 Rechte ... 105
6.4 Gesicherte Kommunikation mit SSL/TLS ... 106
6.4.1 Zertifikate ... 106
6.4.2 Verifikation ... 108
6.4.3 Mini-CA ... 108

7 Anwendungen ... 109

7.1 Einsatzbeispiele ... 109
7.1.1 Netz- und Systemmanagement ... 109
7.1.2 Informationsmanagement ... 110
7.2 Mögliche Anwendungen ... 110
7.3 Einsatz für Sicherheitsanwendungen ... 111
7.3.1 Host Security ... 111
7.3.2 Netzwerk-Scanner ... 112
7.3.3 Intrusion Detection ... 113

8 Zusammenfassung, Bewertung und Ausblick ... 115

8.1 Zusammenfassung ... 115
8.2 Bewertung ... 116
8.3 Ausblick ... 117

Abkürzungsverzeichnis 119

Literaturverzeichnis 123

Anhang 141

A Konfigurationsdateien von MobMan 141
A.1 Format 141
A.2 Beispiel 142
B Das Launch-Programm 152
C TLS: Transport Layer Security 152
D Schlüssel und Zertifikate in Browser importieren 154
E Weitere Agentensysteme 155
E.1 Kommerzielle Agentensysteme 155
E.2 Forschungsprojekte 158
F Weitere Lösungen für Caller-ID 163
F.1 Variante „Cookie“ 163
F.2 Variante „Private ID“ 164
F.3 Variante „geschützte Methode und Forwarder-Referenz“ 164
F.4 Variante „geschützte Methode und Shared Secret“ 165
G Zuordnungstabellen 166
G.1 Plattform 166
G.2 Security Agent 167
G.3 Authentifizierungs-Server 167
G.4 Pia 167
G.5 Migration Agent 168
H UML-Diagramme 168
H.1 Klassendiagramme 168
H.2 Sequenzendiagramme 172

Abbildungsverzeichnis

Abbildung 1.1: Dimensionen des Netzmanagements ... 2
Abbildung 1.2: Netzmanagement – prinzipielle Lösung ... 3
Abbildung 1.3: SNMP-Architektur ... 4
Abbildung 1.4: SNMP mit Vorverarbeitungsagenten ... 5
Abbildung 1.5: Asynchrone Aufgaben: Client-Server und mobile Agenten ... 8
Abbildung 1.6: Aufbau der Arbeit ... 10
Abbildung 2.1: Einteilung intelligenter Agenten ... 12
Abbildung 2.2: Bestimmung des Aufenthaltsortes ... 16
Abbildung 3.1: Überblick über die beteiligten Instanzen ... 28
Abbildung 4.1: Vernetzte MobMan-Plattformen ... 40
Abbildung 4.2: Klassenmodell des MobMan-Systems – Übersicht ... 41
Abbildung 4.3: Klassendiagramm MobManPlatform ... 42
Abbildung 4.4: Klassendiagramm Agent ... 45
Abbildung 4.5: MobMan-Protokolle ... 65
Abbildung 4.6: MAP-Protokoll ... 66
Abbildung 4.7: WAMP-Protokoll ... 67
Abbildung 4.8: WRAP: Beteiligte Instanzen ... 69
Abbildung 4.9: WRAP-Protokoll ... 70
Abbildung 4.10: RAMP-Protokoll ... 71
Abbildung 5.1: Schritte bei der Ausführung von Java-Applets ... 74
Abbildung 5.2: Die Java-Sandbox ... 75
Abbildung 5.3: Kapselung der Agenten ... 81
Abbildung 5.4: Agenten und IDs ... 82
Abbildung 5.5: Caller-ID-Variante „Vertrauen“ ... 84
Abbildung 5.6: Caller-ID-Variante „Thread Group“ ... 85
Abbildung 5.7: Caller-ID-Variante „Rückgriff“ ... 86
Abbildung 5.8: Caller-ID-Variante „Message Stub“ ... 87
Abbildung 5.9: Caller-ID-Variante „geschützte Methode und Package Visibility“ ... 89
Abbildung 6.1: Lokale Authentifizierung ... 97
Abbildung 6.2: Lokale Authentifizierung: User-Passwort-Verfahren ... 98
Abbildung 6.3: Lokale Authentifizierung: Challenge-Response mit Zertifikat ... 99

Abbildung 6.4: Externe Authentifizierung 100
Abbildung 6.5: Authentifizierung von Applets 104
Abbildung 6.6: Zertifikate und CAs 107
Abbildung 7.1: Portscanner bei vorhandener Firewall 112
Abbildung 7.2: Firewall-Test mit mobilen Agenten 113
Abbildung C.1: Aufbau des TLS-Protokolls 153
Abbildung C.2: Verbindungsaufbau bei TLS 154
Abbildung F.1: Caller-ID-Variante „Cookie“ 163
Abbildung F.2: Caller-ID-Variante „Private ID“ 164
Abbildung F.3: Caller-ID-Variante „geschützte Methode und Forwarder-Referenz“ 165
Abbildung F.4: Caller-ID-Variante „geschützte Methode und Shared Secret“ 166

Tabellenverzeichnis

Tabelle 1.1: Probleme des klassischen Netzmanagements 5
Tabelle 2.1: Eigenschaften von Agenten 12
Tabelle 2.2: Ansätze für mobilen Code 13
Tabelle 2.3: Arten der Migration 14
Tabelle 2.4: Potenzielle Vorteile mobiler Agenten 18
Tabelle 3.1: Relevanz möglicher Angreifer 31
Tabelle 4.1: Pia-Module 53
Tabelle 5.1: Kapselung 77
Tabelle 6.1: Aktuell genutzte Rechte in MobMan 105
Tabelle H.1: UML: Klassen und Pakete 169
Tabelle H.2: UML: Beziehungen der Klassen 170
Tabelle H.3: UML: Felder und Methoden 171
Tabelle H.4: UML: Sichtbarkeit von Feldern und Methoden 171
Tabelle H.5: UML: Sequenzendiagramme 173

Kapitel 1

Einführung und Motivation

Es grenzt an eine Binsenweisheit, ist aber nicht von der Hand zu weisen: Computernetze wachsen ständig und ihre Komplexität steigt. Nebenbei nimmt auch die Heterogenität zu. Die Aufgaben des Managements dieser Netze wachsen mit und verlangen nach neuen Ansätzen, die den neuen Anforderungen gerecht werden.

Trotz aller Konsolidierungsbemühungen finden immer neue Geräte Einzug in die Netzwerke und erhöhen die Heterogenität. Neben der klassischen Workstation, dem Server und dem PC führen Appliances spezielle Funktionen aus. Reine Netzinfrastrukturkomponenten wie Router oder Switches erhalten zusätzliche Aufgaben als Paketfilter oder Intrusion-Detection-System. All dies soll in das bestehende Netzmanagement integriert werden.

Zu dieser an sich bekannten Entwicklung gesellt sich ein neuer Trend, der zusätzliche Fragen aufwirft: Mobile Computing. Nicht nur Laptops werden in Unternehmensnetzwerke integriert, auch vom Mobiltelefon aus ist der Zugang über WAP (Wireless Application Protocol) und WML (Wireless Markup Language) möglich. Dazu kommen noch die mobilen persönlichen Assistenten (PDAs, Personal Digital Assistant), die Anschluss an die Unternehmens-DV finden. Sie benutzen Verfahren wie IrDA-Verbindungen (Infrared Data Association), Bluetooth oder, wie beim Mobiltelefon, GSM (Global System for Mobile communication) und in naher Zukunft auch UMTS (Universal Mobile Telecommunications System).

Im Gegensatz zum klassischen Netzmanagement sind diese mobilen Geräte in aller Regel nicht ständig erreichbar. Während im herkömmlichen Netz die Erreichbarkeit den Normalfall darstellt und die Nichterreichbarkeit ein Fehler ist, sind mobile Geräte nicht ständig online. Dies liegt nicht nur an den Kosten einer permanenten Verbindung, sondern auch an technischen Randbedingungen. Beispielsweise überbrückt Bluetooth nur zehn Meter, und ein PDA ist die meiste Zeit sogar ausgeschaltet, um Strom zu sparen. In absehbarer Zeit ist also nicht damit zu rechnen, dass alle Geräte, die es zu administrieren gilt, auch ständig erreichbar sind. Dem muss das Netzmanagement geeignet Rechnung tragen.

1.1 Aufgaben

Vor dem Blick auf neue Ansätze sollte ein Blick zurück auf die etablierten Lösungen stehen. Eine verbreitete Definition des Netzmanagements lautet [Hegering 99]:

> Das Management vernetzter Systeme umfasst [...] alle Maßnahmen, die einen effektiven und effizienten, an den Zielen des Unternehmens ausgerichteten Betrieb der Systeme und ihrer Ressourcen sicherstellt.

Netzmanagement betrifft den Betrieb, die Verwaltung und die Planung eines Rechnernetzes. Es umfasst Verfahren, Methoden und Programme, sowie menschliche Ressourcen, Werkzeuge und technische Systeme. Die Aufgabe lässt sich in fünf Gebiete einteilen: Das Konfigurationsmanagement (*configuration*) kümmert sich um die Konfiguration aller betroffenen Komponenten. Beispiele sind Routing-Tabellen oder Netzmasken. Die Fehlererkennung und -behebung (*fault*) sorgt dafür, dass der Administrator nicht erst von seinen Kollegen erfährt, wenn ein Bereich ausgefallen ist. Das Leistungsmanagement (*performance*) kümmert sich um eine sinnvolle Auslastung der Netzsegmente. In kommerziellen Netzen ist außerdem die Abrechnung (*accounting*) der erbrachten Leitungen eine zentrale Aufgabe. Das Sicherheitsmanagement (*security*) umfasst alle bisher genannten Bereiche und soll die Sicherheit des Netzes erhalten.

Neben der funktionellen Dimension steht die zeitliche Komponente. Das Management greift bereits bei der Planung neuer Netze, setzt sich über die Installation fort und hat seinen Schwerpunkt beim Betrieb. Auch Änderungen bestehender Netze sind eine Management-Aufgabe. Als dritte Dimension gilt das Einsatzszenarium. Das Netzmanagement kann einzelne Komponenten oder Systeme betreffen, aber auch Anwendungen oder die komplette Unternehmens-DV.

Aus diesen drei Dimensionen ergibt sich die Darstellung in Abbildung 1.1 *Dimensionen des Netzmanagements*, Seite 2. Die drei Achsen spannen einen Raum auf, in dem sich die Management-Aufgaben verteilen. Daraus lässt sich unmittelbar ablesen, dass es sich nicht um singuläre Aufgaben handelt, sondern ein umfassender Ansatz gefordert ist.

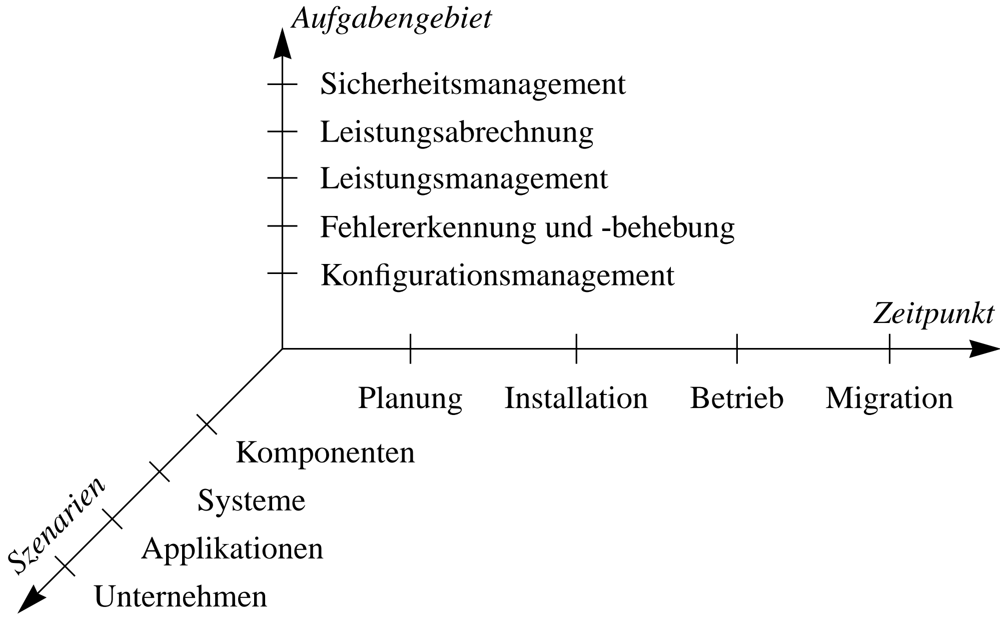

Abbildung 1.1 *Dimensionen des Netzmanagements*

1.2 Ansätze

1.2.1 Der klassische Ansatz

In den Netzen, die im Rahmen dieser Arbeit im Vordergrund stehen, hat sich das von der IETF (Internet Engineering Task Force) standardisierte SNMP (Simple Network Management Protocol, [RFC 1157]) als De-facto-Standard etabliert, während in OSI-Netzen (Open Systems Interconnection) meist CMIP (Common Management Information Protocol, [ISO IS9596]) zum Einsatz kommt. Beide Verfahren gehen von einer zentralen Managementstation aus, die über das jeweilige Managementprotokoll die einzelnen zu verwaltenden Geräte administriert. Stark an das OSI-Management angelehnt ist TMN, das Telecommunications Management Network [Sidor 98]. Es ist an die Anforderungen öffentlicher Telekommunikationsnetze angepasst.

Abbildung 1.2 *Netzmanagement – prinzipielle Lösung*, Seite 3 verdeutlicht die Architektur des klassischen Internet-Netzmanagements. Über ein lokales Netz sind die verschiedenen zu verwaltenden Geräte verbunden, hier als PC, Workstation, Server und Host bezeichnet. Eines der Hauptziele des Managements sind dabei die Netzinfrastrukturkomponenten, in der Grafik sind ein Gateway und eine Bridge als Beispiele enthalten.

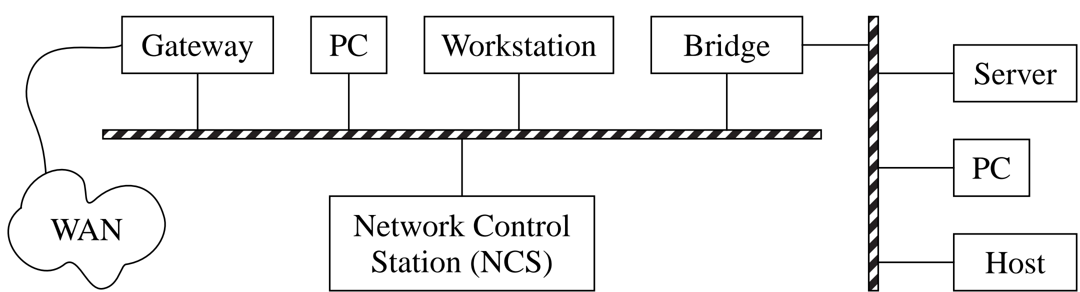

Abbildung 1.2 *Netzmanagement – prinzipielle Lösung*

Das Management findet zentral an der Network Control Station (NCS) statt. Über das Netzmanagementprotokoll (beispielsweise SNMP) nimmt sie Verbindung zu den von ihr administrierten Geräten (Managed Objects) auf und fragt dort nach dem aktuellen Zustand oder sie greift aktiv ein und ändert beispielsweise Parameter. Nur im Ausnahmefall meldet sich ein Managed Object von sich aus beim Manager.

Das klassische Netzmanagement basiert also auf einer Client-Server-Architektur. Abbildung 1.3 *SNMP-Architektur*, Seite 4 zeigt, wie diese Architektur bei SNMP umgesetzt ist. Die Manager-Applikation ruft über `GET`- und `GET_NEXT`-Anfragen Daten von den Managed Objects ab. In beiden Fällen antwortet das Managed Object mit den gewünschten Informationen. Die Struktur und das Format dieser Daten ist in der Management Information Base (MIB) beschrieben.

Über `SET`-Kommandos kann die Manager-Applikation auch Daten auf den einzelnen Geräten ändern und damit aktiv in die Konfiguration eingreifen. Lediglich in vorher festgelegten Ausnahmefällen versendet das Managed Object ohne vorherige Anfrage ein `TRAP`-Paket an die NCS.

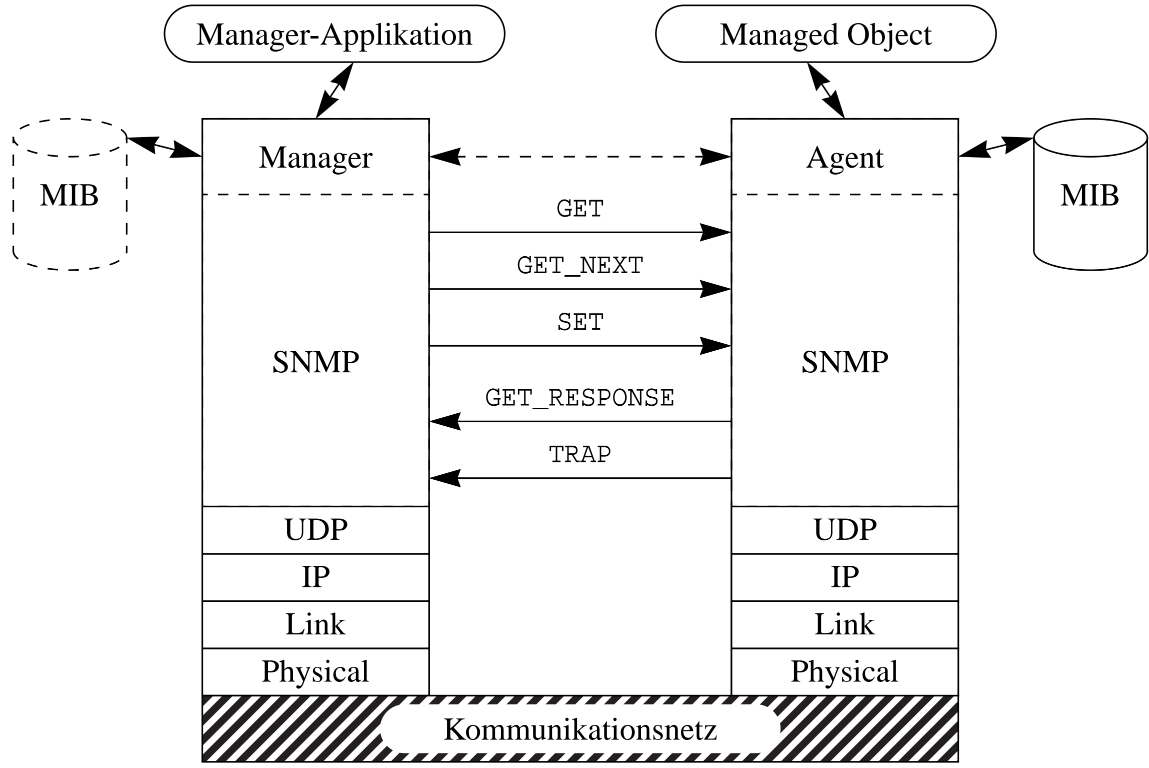

Abbildung 1.3 *SNMP-Architektur*

Diese knappe Beschreibung zeigt bereits, dass jede Anfrage des Managers und jeder Änderungswunsch zu einer Belastung des Netzes führen. Dies lässt sich kaum vermeiden, die Information muss schließlich vom Manager zum Managed Object und umgekehrt gelangen. Problematisch ist daran unter anderem, dass eine direkte Verbindung zwischen beiden bestehen muss, und zwar zu dem Zeitpunkt, zu dem ein Zustand abgefragt werden soll oder eine Änderung vorzunehmen ist.

An der Network Control Station laufen alle Verbindungen zusammen. Ihre Anbindung stellt ein Nadelöhr dar, über sie laufen alle Informationen. Je größer das Netz wird, je mehr Managed Objects es zu verwalten gilt, um so größer wird die Netzlast, die das Netzmanagement selbst verursacht. Dadurch stößt die Skalierbarkeit an ihre Grenzen. Wenn das Netz in Teilbereichen mehr mit dem Transport der Managementdaten beschäftigt ist als mit Nutzdaten, dann erfüllt das Netzmanagement nur noch einen Selbstzweck. Tabelle 1.1 *Probleme des klassischen Netzmanagements*, Seite 5 fasst die wichtigsten Grenzen zusammen, an die das Netzmanagement in größeren Netzen stößt [Sugauchi 99].

Problem	Begründung
Höhere Last auf der Management-Station	Je größer das Netz, desto mehr Netzkomponenten muss die Management-Station verwalten. Jede Änderung muss einzeln an alle Komponenten übertragen werden.
Mehr Netzverkehr bei den Netzkomponenten	Je mehr Aufgaben zu managen sind, um so mehr Netzlast entsteht bei jeder einzelnen Netzkomponente.
Mehr Netzverkehr bei der Management-Station	Hier steigt die Netzlast mit der Anzahl der Netzkomponenten und mit der Komplexität der Aufgaben.
Schwierigkeit, neue Management-Funktionen bereitzustellen	Neue Funktionen müssen in der Regel nicht nur in der Management-Station implementiert werden, sondern jede Netzkomponente muss die neuen Funktionen bereitstellen.

Tabelle 1.1 *Probleme des klassischen Netzmanagements*

1.2.2 Dezentrales Management

Einen Ausweg aus diesem Dilemma hat das WILMA-Projekt am Lehrstuhl für Datenverarbeitung aufgezeigt [Konopka 95], [Trommer 96]. Durch mehrstufiges Management, die Einteilung in Domänen und durch intelligente Vorverarbeitung (Abbildung 1.4 *SNMP mit Vorverarbeitungsagenten*, Seite 5) lassen sich viele der beschriebenen Probleme erfolgreich bewältigen.

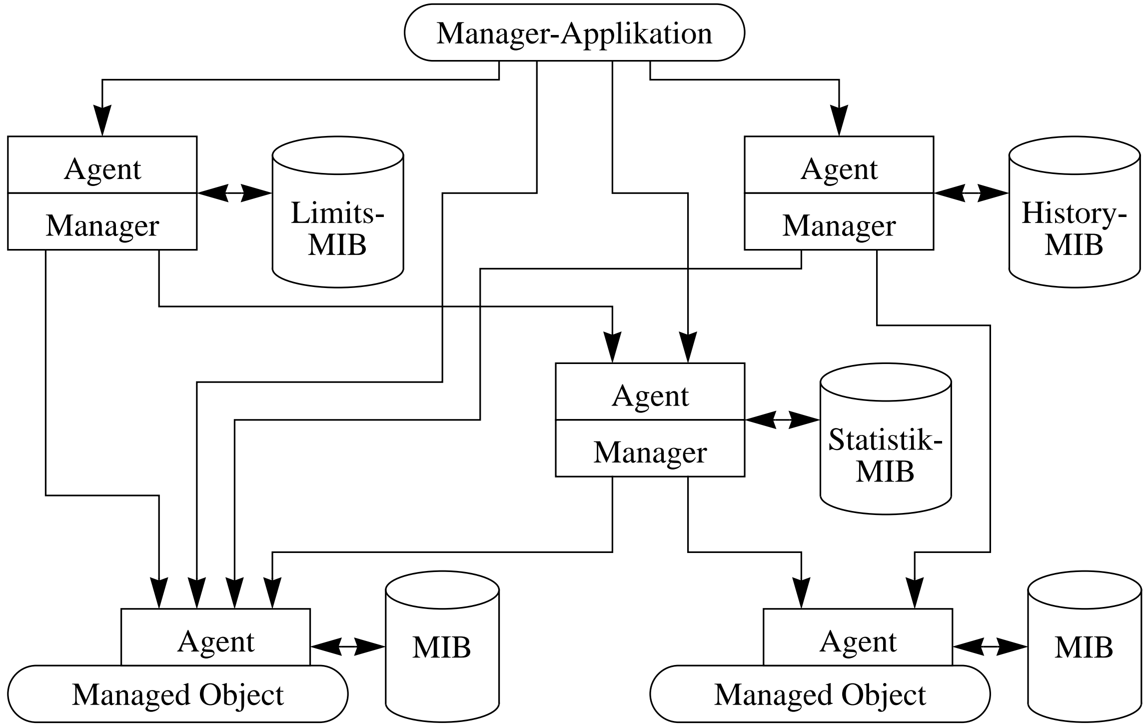

Abbildung 1.4 *SNMP mit Vorverarbeitungsagenten*

Die intelligenten Vorverarbeitungsagenten bringen Dezentralität in das SNMP-Konzept ein: Sie vereinen die Funktionalität von Agenten und Managern in sich. Sie können direkt auf die Agenten der Managed Objects zugreifen, weitere Vorverarbeitungsagenten benutzen und von herkömmlichen Management-Plattformen über SNMP verwaltet werden. Besonders geeignet sind sie für Statistik, Prognose, Logging (History) und Alerting (Limits und deren Überwachung).

Die Traps in SNMP könnten als einfacher Ansatz für dezentrales Management verstanden werden. Auch die Proxy-Agenten aus SNMPv2 sowie die RMON-MIB (Remote Network Monitoring Management Information Base, [RFC 1757]) dienen dazu, Dezentralität in SNMP einzuführen [Puliafito 99]. RMON spielt in der Praxis eine herausragende Rolle. Als standardisierte Network Probe (Messsonde) sammelt sie Daten über den Netzverkehr und wertet ihn unter anderem statistisch aus. Die Auswertung geschieht auch ohne Verbindung zur Manager-Applikation. Allerdings ist der Einsatzbereich von RMON durchaus eingeschränkt [Zapf 99a]:

- Die RMON-MIB zielt ursprünglich auf die Messung von Ethernet-LAN-Segmenten auf MAC-Ebene. Für andere Medien oder höhere Schichten sind Erweiterungen nötig.
- RMON-Probes sind, wie andere SNMP-Agenten auch, plattformabhängig.
- Die Konfiguration von RMON ist statisch.

Einen interessanten Ansatz stellt auch DISMAN dar (Distributed Management), es realisiert das Prinzip „Management by Delegation“. DISMAN definiert Module, die sich – ähnlich wie die Vorverarbeitungsagenten des WILMA-Projekts – gleichzeitig als Agent und als Manager verhalten. Die Managementaufgabe wird als Skript in einer beliebigen, nicht näher festgelegten Programmiersprache formuliert und über die Script-MIB [RFC 3165] verteilt. Diese überträgt die Management-Skripts, kontrolliert und überwacht ihre Ausführung und übermittelt Parameter und Ergebnisse. [Hegering 99] stellt allerdings in Frage, ob sich das DISMAN-Konzept wegen seiner hohen Implementierungskosten durchsetzen kann.

1.2.3 Webbasiertes Management

Eine relativ junge alternative Technik ist das webbasierte Management. Dieses Verfahren befreit den Manager von der Abhängigkeit von einer speziellen Managementkonsole. Auf den Managed Objects läuft dabei ein einfacher Webserver mit einer Webapplikation, die das Management der jeweiligen Komponente anbietet.

Für einzelne Geräte bietet dieser Ansatz entscheidende Vorteile, jeder gewöhnliche PC mit Webbrowser kann als NCS dienen. Andererseits ist auch hier eine Online-Verbindung nötig, außerdem steigt die Menge der übertragenen Daten. Neben den reinen Managementdaten muss auch die Benutzeroberfläche, hier in Form von HTML-Seiten, übertragen werden. Dazu kommt ein weiterer Nachteil: Jedes Gerät hat seine eigene Management-Applikation, der Administrator muss sich also jedes mal in eine neue Anwendung einarbeiten. Zudem skaliert diese Lösung schlecht, jedes Gerät wird einzeln verwaltet. So praktisch dieser Ansatz für kleine Netze mit sehr wenigen Geräten ist, so ungeeignet ist er in großen, heterogenen Umgebungen.

1.2.4 Web Services

In jüngerer Zeit ist unter webbasiertem Management eine neue Variante zu verstehen. Sie nutzt zwar Web-Techniken wie HTTP (Hypertext Transport Protocol) und XML (Extensible Markup Language), allerdings nicht im Rahmen des Client-Server-Modells mit Browser und Webserver, sondern als Kommunikationsmechanismus innerhalb aufwändigerer Infrastrukturen. Mit Web-Based Enterprise Management WBEM [DMTF-02] steht hierfür ein Standard bereit, herausgegeben von der DMTF (Distributed Management Task Force).

Trotz ihres Namens handelt es sich bei Web Services nicht um Webseiten, sondern im Grunde um eine Alternative zu RPC (Remote Procedure Call) oder CORBA (Common Object Request Broker Architecture). Auch CORBA und RPC dienen als Basis für eine Reihe von Netzmanagement-Applikationen. Prominenter Vertreter für CORBA ist Tivoli TME, auf RPC basiert das Desktop Management Interface DMI [DMTF-98] der DMTF. Web Services benutzen eine XML-Sprache für die PDUs (Protocol Data Units), die über HTTP transportiert werden, während CORBA [OMG 02] und RPC eigene Standards darstellen.

Bei diesen Techniken handelt es sich um verteilte Applikationen, deren Bestandteile auch auf den Managed Objects installiert sein müssen. Auch hier ist wieder eine eigene Management-Konsole nötig, die Benutzeroberfläche kann allerdings in Form von HTML-Seiten realisiert sein und damit über jeden Browser bedient werden.

1.3 Der neue Ansatz

Bei den klassischen Lösungen bleibt die Abhängigkeit des Managements vom Netzwerk – von dem Netz, das es zu managen gilt, oder von einem zusätzlichen Management-Netz. Die Managed Objects müssen zudem das jeweilige Protokoll unterstützen. Die strenge Client-Server-Architektur erweist sich dabei als starr und unflexibel [Baldi 97]. Selbst bei intelligenten Vorverarbeitungsagenten ist eine Netzverbindung zum Agenten zwingende Voraussetzung.

Bei der Suche nach Alternativen zum Client-Server-Modell finden sich verschiedene Paradigmen für den Entwurf verteilter Anwendungen. Ansätze mit mobilem Code können hier ihre spezifischen Vorteile ausspielen [Carzaniga 97]. Eine recht junge Variante sind mobile Agenten.

1.3.1 Management mit mobilen Agenten

Die Kernidee des Managements mit mobilen Agenten ist, nicht nur Daten mit einem Managed Object auszutauschen, sondern auch Code, also vollständige Programme [Bic 98]. Diese Programme werden dann remote auf dem Managed Object ausgeführt und können auch ohne Verbindung zur Management-Applikation eigene Entscheidungen fällen, Statistiken führen oder von sich aus weitere Komponenten kontaktieren.

Ein klassisches Programm, das manuell auf dem Managed Object installiert wird, kann den wechselnden Anforderungen nicht immer gewachsen sein. Wann immer neue Probleme zu lösen sind, eine neue Statistik geführt werden muss oder eine bestimmte Änderung vorzunehmen ist, wäre ein neues Programm oder eine neue Version zu installieren. Dadurch verlagert sich unter Umständen ein Teil der Aufgaben des Managers auf den Anwender, der das Programm installieren müsste. Dies kann nicht im Sinne des Netzmanagements sein. Die naheliegende Lösung ist, dass der Manager selbst die neuen Programme installiert, etwa durch remote Login mit Rlogin oder Telnet. Aber dafür ist wieder die direkte Online-Verbindung nötig, die es ja gerade zu umgehen gilt. Dazu kommt, dass in einer heterogenen Umgebung viele verschiedene Versionen für die jeweiligen Architekturen vorzuhalten und zu installieren sind.

Bei mobilen Agenten stellt sich die Situation anders dar. Hier wandert (migriert) der Code mitsamt seinem aktuellen Zustand (den Daten) selbständig von Plattform zu Plattform. Ein mobiler Agent verhält sich, einmal auf einer Plattform angekommen, wie jedes gewöhnliche Programm. Er kann also die Konfiguration ändern [Berghoff 96], um beispielsweise Fehler beim Netzzugriff zu beheben, er kann Statistikdaten sammeln oder auch Netzverbindungen zu anderen Diensten aufnehmen. Damit könnte er mittels SNMP, CMIP oder Web-Zugriff auch weitere Managed Objects verwalten. [Puliafito 99] [Bellavista 99] [Zapf 99a] [Zapf 99b]

Agenten und Plattformen können auch selbst Web-Schnittstellen anbieten, über die der Administrator mithilfe eines Browsers Kontakt zu seinen Agenten und Plattformen aufnehmen kann. Verallgemeinert ergibt sich hier das Management des Agentensystems selbst [Breugst 99] als Meta-Management-Aufgabe.

1.3.2 Vorteile im Netzmanagement

Bereits frühe Überlegungen zu mobilen Agenten [Harrison 95] kamen zu dem Schluss, dass diese Technik zwar keinen singulären entscheidenden Vorteil besitzt, aber durch die Summe vieler einzelner Pluspunkte doch überzeugende Vorzüge aufweist. Im Unterschied zu herkömmlichen Programmen kann sich ein Agent zu jedem Zeitpunkt auf eine andere Plattform übertragen, mitsamt seinen bereits gesammelten Statistikdaten und den Konfigurationsänderungen, die der Administrator für die weiteren Plattformen vorgesehen hat. Der Aufwand für die Integration neuer Module in das Netzmanagement ist minimal, da nur ein Agent gestartet werden muss anstatt der erwähnten Installation auf sämtlichen Managed Objects. Diese schnelle Realisierung und Integration neuer Funktionalität ist eine der herausragensten Eigenschaften.

Da ein Agent auf dem Managed Object lokal ausgeführt wird, sind seinen Einflussmöglichkeiten zunächst keine Schranken gesetzt. Auch durch den wechselnden Ort innerhalb des Netzes ergeben sich Vorteile: Wenn eine Firewall den Zugriff auf einen Rechner sperrt, dann kann ein mobiler Agent, der sich hinter der Firewall befindet, doch noch die dringend notwendigen Konfigurationsänderungen vornehmen. Oder die Netzauslastung in diesem Segment messen, oder den erzielbaren Durchsatz. Von der anderen Seite der Firewall aus wäre all dies nicht möglich.

In vielen Fällen können mobile Agenten die durch das Management selbst verursachte Netzlast vermindern, da eben nicht mehr alle Daten vom Client zum Server übertragen werden müssen. Dieses Einsparpotenzial wurde früh erkannt [Chess 95] und später in genaueren Untersuchungen nachgewiesen [Fuggetta 98], [Sahai 98], [Puliafito 00], [Sugauchi 99]. Realitätsnahe Experimente zeigen, dass vor allem die Verteilung der Agenten Vorteile bringt [Simões 02]. Die Untersuchungen von [Gray 00] weisen für einige Anwendungsbeispiele aus anderen Bereichen neben der geringeren Netzlast auch eine höhere Gesamtperformance nach (Aufgabe schneller erfüllt, niedrigere Latenzzeiten) sowie eine günstige Verteilung der Rechenlast. Mithilfe einer Bewertung der einzelnen Verbindungen ergibt sich durch geschickte Wegewahl zusätzliches Einsparpotenzial [Theilmann 99].

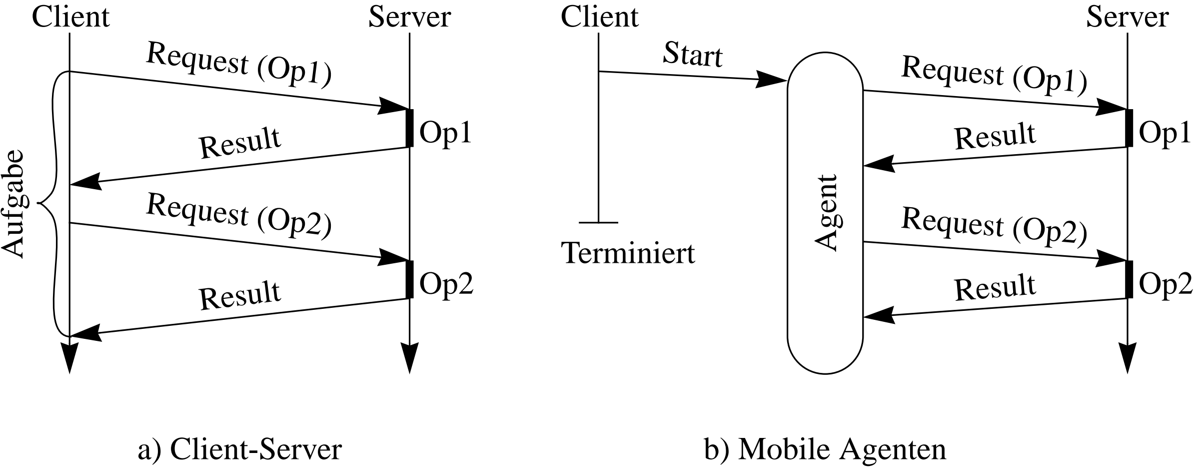

Abbildung 1.5 *Asynchrone Aufgaben: Client-Server und mobile Agenten*

Ein entscheidender Vorteil ergibt sich bei mobilen Geräten: Sobald der Agent übertragen ist, kann er ohne Netzanbindung lokal weiterarbeiten. Diese Eigenschaft hilft auch bei Netzen, die eigentlich immer online sein sollten: Bricht die Verbindung zusammen, dann ist das Netzmanagement besonders gefordert. Genau dann hilft es, wenn Teile der Management-Applikation auf beiden Seiten weiter funktionieren [Baumann 98a], [Gray 97a]. Diese asynchrone Eigenschaft verdeutlicht Abbildung 1.5 *Asynchrone Aufgaben: Client-Server und mobile Agenten*, Seite 8.

Durch den parallelen Einsatz mehrerer Agenten lässt sich Redundanz erzeugen, die für Fehlertoleranz notwendig ist. Redundante Agenten können unabhängig voneinander oder in Gruppen organisiert sein [Baumann 97b].

[Lange 98a] und [Lange 99] fassen die Vorteile in sieben Punkten zusammen:

1. Geringere Netzlast
2. Keine Latenzzeiten durch das Netzwerk
3. Kapseln Protokolle
4. Laufen asynchron und autonom
5. Passen sich dynamisch an
6. Sind von sich aus heterogen
7. Robust und fehlertolerant

1.3.3 Integration

Der Einsatz mobiler Agenten erfordert nicht notwendigerweise eine völlige Umstellung: Die Agenten können und sollen auch klassische Managementprotokolle wie SNMP oder HTTP nutzen, um mit existierenden (Legacy)-Systemen zu kommunizieren. [Baldi 97] [Puliafito 99]

Untersuchungen am Lehrstuhl für Datenverarbeitung haben gezeigt, dass mobile Agenten nicht in jedem Fall die ideale Lösung darstellen. Je nach vorliegender Situation sollte der Manager daher zwischen den verschiedenen Mechanismen abwägen. Aus der Kombination der verschiedenen Managementverfahren ergeben sich die größten Vorteile. [Reitzig 99]

1.3.4 Probleme

So vorteilhaft die Möglichkeiten mobiler Agenten sind, so bergen sie durchaus auch Gefahren [Baumann 95], [Dula 97]. Bereits die sehr weitreichenden Möglichkeiten lassen erahnen, dass die Sicherheit des Agentensystems und der verwalteten Geräte gefährdet ist. Auch einem Angreifer, der einen Administrationsagenten einschleusen kann, stehen diese enormen Möglichkeiten zur Verfügung. Die Sicherheit hat folglich zentrale Bedeutung.

Da ein mobiler Agent auf den unterschiedlichsten Architekturen ausführbar sein soll, kommen keine Binärprogramme in Frage. Stattdessen ist eine Skriptsprache erforderlich, oder zumindest ein einheitlicher Zwischencode, der von einem Interpreter ausgeführt wird. Der Migrationsschritt selbst bedingt spezielle Programme und Plattformen, die diese Möglichkeit bereitstellen. Schließlich ist noch dafür Sorge zu tragen, dass die Netzbelastung nicht durch große Mengen zu übertragendem Programmcode weiter steigt. Je nach Szenario ist die Netzbelastung durch Client-Server-Architekturen höher oder niedriger als bei mobilen Agenten [Baldi 98], die Vorteile der Architektur sollte das Protokoll also nicht verspielen.

All diese Probleme zu umgehen und dennoch die potenziellen Vorteile ausschöpfen zu können, ist der Inhalt dieser Arbeit. Durch die Sicherheitsarchitektur des Agentensystems kann auch der Einsatz klassischer, unsicherer Netzmanagementprotokolle wie SNMPv1 besser geschützt werden. Wenn die Protokollpakete nicht mehr durch das ganze Netz zwischen Network Control Station und Managed Object laufen, sondern über eine möglichst geringe Distanz, eventuell sogar nur innerhalb des Rechners, dann sinken auch die Möglichkeiten potenzieller Angreifer, die Managementdaten auszuspionieren oder zu manipulieren. Agenten lassen sich auch für das Sicherheitsmanagement nutzen, etwa um Sicherheitstests vorzunehmen, die bei anderen Techniken bereits an einer Firewall scheitern würden.

1.4 Überblick

Um eine geeignete Sicherheitsarchitektur zu finden, sind zunächst zwei Grundlagen erforderlich: Neben einer Aufstellung der Anforderungen an das System ist eine Analyse der Bedrohungen nötig. Kapitel 2 beginnt mit einer genaueren Beschreibung mobiler Agenten und stellt einige bereits existierende Agentensysteme vor. Darauf folgt in Kapitel 3 eine Analyse der Bedrohungen. Ausgehend von den beteiligten Instanzen und den potenziellen Angreifern wird auf die notwendigen Barrieren geschlossen. Dieses Kapitel endet mit der Beschreibung der benötigten Schutzmechanismen.

Die folgenden drei Kapitel behandeln die Umsetzung innerhalb der MobMan-Architektur (Mobile Manager). Die Gesamtarchitektur wird – ausgehend von den relevanten Designkriterien in Kapitel 4 – vorgestellt, wobei die verwendeten Protokolle und Nachrichtenmechanismen eine zentrale Rolle einnehmen. Bevor Kapitel 6 spezielle Sicherheitsmechanismen in MobMan vorstellt, geht Kapitel 5 auf die Java-spezifischen Probleme ein. Abbildung 1.6 *Aufbau der Arbeit*, Seite 10 zeigt, wie diese Kapitel zusammenhängen.

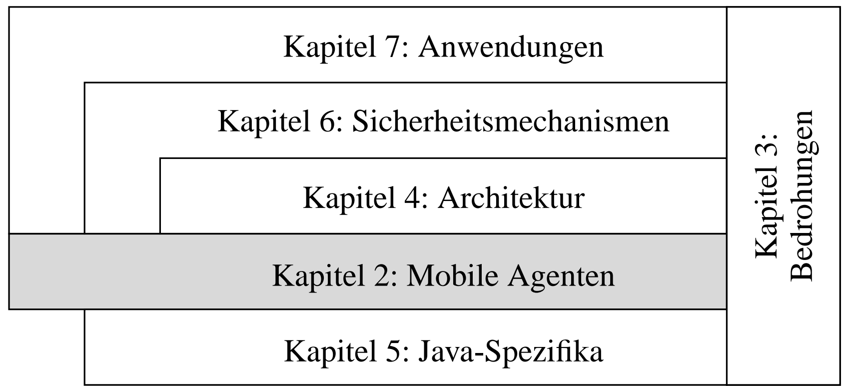

Abbildung 1.6 *Aufbau der Arbeit*

Den Abschluss bilden Kapitel 7 mit einigen Einsatzszenarien von MobMan und Kapitel 8 mit einer Zusammenfassung der Arbeit.

Kapitel 2

Mobile Agenten

Das Einleitungskapitel hat bereits den Begriff „mobiler Agent“ eingeführt. Dieses Kapitel definiert nun mobile Agenten, stellt die Grundlagen dieses neuen Paradigmas für verteilte Systeme dar, führt die wesentlichen Vorteile auf und stellt eine Reihe bestehender Lösungen vor.

2.1 Grundlagen mobiler Agenten

Der Agentenbegriff wird teilweise sehr unterschiedlich definiert [Franklin 96], [Bradshaw 96]. Abbildung 2.1 *Einteilung intelligenter Agenten*, Seite 12 zeigt eine übliche Charakterisierung anhand der Eigenschaften Agententum, Intelligenz und Mobilität:

- *Agententum*
  Der Grad an Autonomie und Autorität, über die ein Agent verfügt. Ein Agent muss zumindest asynchron laufen. Das Agententum steigt, wenn der Agent den Benutzer vertritt oder wenn er mit Daten, Applikationen oder Diensten interagiert.
- *Intelligenz*
  Intelligenz beschreibt, in welchem Umfang ein Agent Schlussfolgerungen treffen, planen und lernen kann.
- *Mobilität*
  Mobilität besagt, inwiefern sich ein Agent durch das Netzwerk bewegen kann. Mobile Skripte werden auf einer Maschine erzeugt und auf einer anderen ausgeführt, während mobile Agenten von Maschine zu Maschine wandern und dabei ihren Zustand mitnehmen.

Im Rahmen dieser Arbeit sind mit Agenten immer mobile Agenten gemeint, wobei der Schwerpunkt auf dem Aspekt Mobilität liegt.

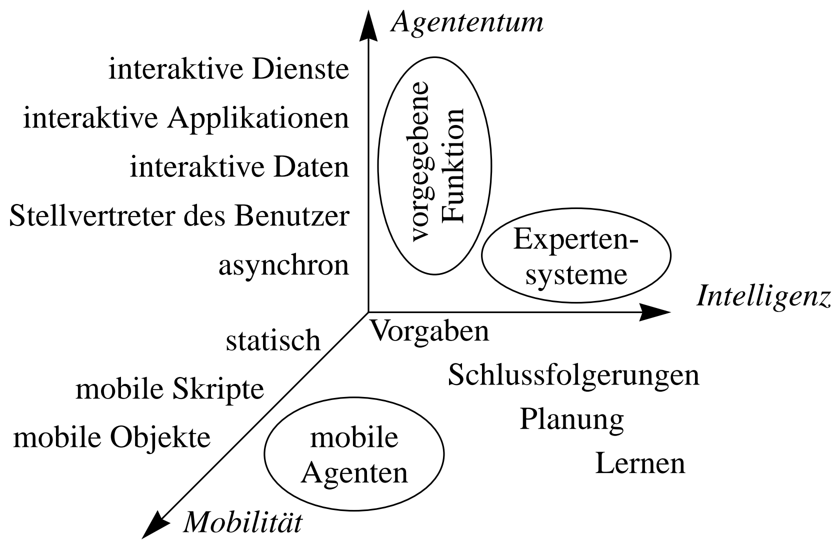

Abbildung 2.1 *Einteilung intelligenter Agenten*

Allgemein herrscht Einigkeit darüber, dass ein Agent mindestens die in Tabelle 2.1 *Eigenschaften von Agenten*, Seite 12 genannten Eigenschaften aufweisen muss [Braun 99].

[Franklin 96] führt ergänzend noch weitere typische Eigenschaften von Agenten auf: Adaptivität (Lernfähigkeit), Mobilität, Flexibilität und Charakter (Persönlichkeit). Agenten sind kommunikativ und werden als ein fortlaufender Prozess ausgeführt (nicht ständig neu gestartet).

Eigenschaft	Erläuterung
Autonomie	Einem Agent ist nicht jeder Teil seiner Handlung vorgegeben, er operiert nach einem selbst erstellten Plan ohne ständigen Einfluss des (menschlichen) Auftraggebers.
Soziales Verhalten	Agenten kommunizieren miteinander oder mit ihrem Auftraggeber. Sie benutzen hierzu häufig eine eigene Sprache: ACL, die Agent Communication Language.
Reaktivität	Agenten nehmen ihre Umwelt wahr und reagieren auf sie.
Proaktivität	Agenten sind zu zielgerichtetem Planen und Handeln fähig, sie ergreifen die Initiative.

Tabelle 2.1 *Eigenschaften von Agenten*

2.1.1 Mobiler Code

Nicht jeder mobile Code ist auch ein mobiler Agent. Bei mobilem Code sind im Wesentlichen drei Ansätze verbreitet [Vigna 98b], die sich wie folgt charakterisieren lassen:

- *Remote Evaluation, REV*
  Ein komplettes Programm wird zu einem Server übertragen und das Ergebnis an den Client übermittelt. Ein Beispiel hierfür ist die Remote Shell unter Unix.
- *Code on Demand, COD*
  Das Programm oder Teile davon werden bei Bedarf von einem Server geladen. Beispiele sind Java-Applets oder ActiveX-Controls.
- *Mobile Agenten, MA*
  Das komplette Programm, mit seinem aktuellen Zustand, kann sich frei von einer Plattform zur nächsten bewegen.

[Ghezzi 97] vergleicht diese drei Ansätze mit dem klassischen Client-Server-Modell und entwickelt dazu eine Systematik. Die Autoren betrachten den Code (dort als *Know-how* bezeichnet), die Daten (*Ressourcen*) und den Prozessor (meint die abstrakte Maschine, auf der das Programm ausgeführt wird). In einer verteilten Anwendung mit zwei Seiten A und B unterscheiden sich die Architekturen dadurch, wie die drei Teile der Anwendung (Code, Daten und Prozessor) auf beiden Seiten verteilt sind. Tabelle 2.2 *Ansätze für mobilen Code*, Seite 13 zeigt, wo bei den Architekturen die drei Teile zunächst liegen, wenn Seite A die Aktionen auslöst. Der Stern (*) kennzeichnet, auf welcher Seite die Verarbeitung stattfindet, also wo die drei Bestandteile im Laufe der Aktion zusammenkommen. Die Tabelle enthält auch das klassische Client-Server-Modell, obwohl hier keinerlei Mobilität von Code vorliegt.

Paradigma	Seite A	Seite B
Client-Server		Code Daten * Prozessor
Remote Evaluation	Code	 Daten * Prozessor
Code on Demand	 Daten * Prozessor	Code
Mobile Agenten	Code Prozessor 1	 Daten * Prozessor 2

Tabelle 2.2 *Ansätze für mobilen Code*

Im Client-Server-Modell liegen Code, Daten und die Verarbeitungsleistung auf Seiten des Servers. Bei Remote Evaluation stammt der Code von A, wird aber auf B ausgeführt. Im Gegensatz dazu holt sich bei Code on Demand A den benötigten Code von B, die Verarbeitung findet beim Initiator A statt.

Bei mobilen Agenten stammt der Code von Seite A und läuft ursprünglich auch dort ab. Der Code migriert dann zu den Daten, die er verarbeiten soll, und wird dort weiter ausgeführt. Die Darstellung in [Ghezzi 97] enthält nur Prozessor 1. Charakteristisch für mobile Agenten ist aber gerade, dass sie auf beiden Seiten ausgeführt werden, daher wurde die Darstellung um einen zweiten Prozessor ergänzt.

[Baumann 98a] betrachtet die übertragenen Informationen und entwickelt daraus eine Systematik verschiedener Arten von Code-Mobilität (siehe Tabelle 2.3 *Arten der Migration*, Seite 14). Remote Execution überträgt Code und Daten an den Zielrechner, der das Programm bis zu dessen Ende ausführt. Remote Evaluation gibt zusätzlich die Ergebnisse an den ursprünglichen Rechner zurück. Bei Code on Demand holt der Client den Code und die darin enthaltenen Daten vom Server ab und führt das Programm dann selbst aus.

Migrationsart	Code	Daten (Konstanten und Parameter)	Zustand (Daten)	Ausführungs-zustand
Remote Execution, Remote Evaluation und Code on Demand	x	x		
Schwache Migration	x	x	x	
Starke Migration	x	x	x	x

Tabelle 2.3 *Arten der Migration*

Die Mobilitätsformen Remote Execution und Code on Demand übertragen nur den Code, nicht den Zustand. Erst die Mobilität von Agenten (Migration) schließt auch den Zustand mit ein, wobei zwei Fälle zu unterscheiden sind:

- *Schwache Migration*
  Bei der schwachen Migration wird der Zustand des Agenten nur in Form seiner aktuellen Daten übertragen. Die Ausführung beginnt an jedem neuen Ort mit dem Aufruf einer vorgegebenen Funktion oder Methode.
- *Starke Migration*
  Die starke Migration überträgt zusätzlich den Ausführungszustand. Der Agent setzt seinen Ablauf exakt an der Stelle fort, an der er auf der vorherigen Plattform die Migration angestoßen hatte.

Zu weiteren Arten von mobilem Code siehe etwa [Bieszczad 98b], zur feineren Untergliederung von Migrationsmechanismen [Braun 99] und zu geeigneten Sprachen [Cugola 97a] und [Cugola 97b]. Ein Framework zur Klassifikation von Konzepten für mobilen Code, das gut zwischen Implementierungstechniken, Applikationen und Paradigmen unterscheidet, findet sich in [Fuggetta 98]. Die Autoren dieser Arbeit weisen besonders auf die unscharfe Definition vieler Begriffe im Umfeld von mobilem Code hin. Unter dem Zustand eines Agenten, der bei der Migration mit übertragen wird, können sich sehr unterschiedliche Dinge verbergen, ebenso hinter dem Begriff des Agenten selbst.

2.1.2 Fähigkeiten von Agenten

[Abdalla 97] erstellt ein Modell eines Agentensystems, in dem die Agenten eine Reihe von charakteristischen Fähigkeiten besitzen.

a) Erzeugen

Agenten können andere Agenten erzeugen, die ihre eigenen Aufgaben erledigen. Diese neuen Agenten können lokal oder remote entstehen; sie sind unabhängig von ihrem Erzeuger.

b) Ausführen

Agenten müssen in irgend einer Form ausgeführt werden. In der Regel geschieht das mithilfe eines Interpreters, der eine Laufzeitumgebung zur Verfügung stellt, innerhalb der der Agent abläuft.

c) Zugang zu Ressourcen

Ein Agent benötigt Zugang zu verschiedenen Ressourcen. Neben den Daten, die er selbst mit sich trägt, sind das vor allem Daten und Dienste, die auf der jeweiligen Plattform lokal zur Verfügung stehen.

d) Migration

Dieser Mechanismus verleiht dem Agenten seine Mobilität. Migration ist die Besonderheit bei mobilen Agenten, mit ihrer Hilfe wandert der Agent von Plattform zu Plattform.

e) Kommunikation

Agenten sollen miteinander kommunizieren, um ihre Aufgaben zu erfüllen. Man kann zwischen lokaler Kommunikation (alle Partner sind auf einer Plattform) und Kommunikation über Plattformgrenzen hinweg unterscheiden. Agenten können auch mit anderen Diensten kommunizieren, die nicht Teil einer Plattform sind.

f) Unterstützung durch die Sprache

Verschiedene Programmiersprachen bieten bereits einige Eigenschaften, die sie für mobile Agenten besonders geeignet erscheinen lassen. Vor allem interpretierte Sprachen haben hier Vorteile, da der Interpreter unbeschränkte Kontrolle über den Agenten ausüben kann.

g) Weitere Dienste

Zusätzliche Dienste können meist von Agenten oder der Plattform selbst bereitgestellt werden. Beispiele sind Authentifizierung oder Namensdienste.

2.1.3 Anforderungen an die Infrastruktur

An die Infrastruktur sind nach [Aridor 98a] einige Anforderungen zu stellen. Die Infrastruktur wird von den Plattformen bereitgestellt, auf denen die Agenten ausgeführt werden.

a) Kommunikation

Agenten können nicht nur mit anderen Agenten kommunizieren. [Baumann 97a] führt zusätzlich die Kommunikation mit Diensten (die auch als Agenten realisiert sein können) und Gruppen von Agenten an, ebenso die Kommunikation mit dem Anwender.

Bei der Kommunikation zwischen und mit den Agenten tritt das Problem der Lokalisierung auf [Roth 01c], also der Frage, wo sich der Agent gerade befindet. Hier sollte die Plattform in geeigneter Weise Unterstützung anbieten. Oft wird ein Namensdienst eingesetzt, der einen Agenten unabhängig vom Aufenthaltsort adressieren kann.

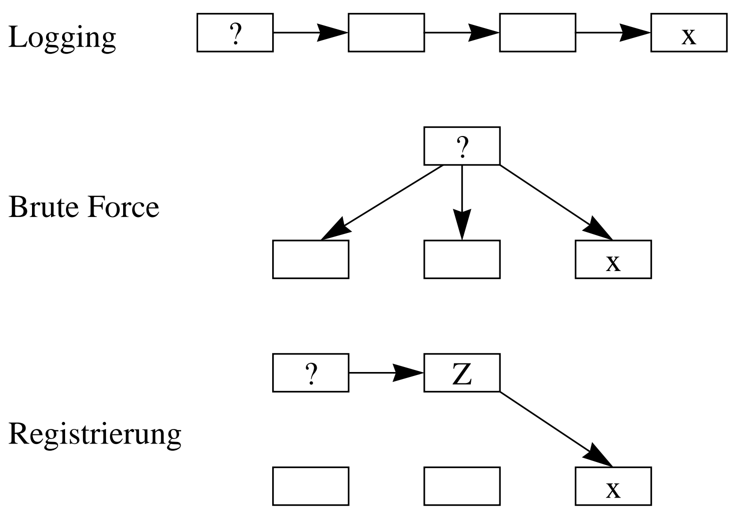

Abbildung 2.2 *Bestimmung des Aufenthaltsortes*

Der Aufenthaltsort eines Agenten lässt sich durch verschiedene Mechanismen ermitteln (siehe Abbildung 2.2 *Bestimmung des Aufenthaltsortes*, Seite 16). [Baumann 99a] untersucht diese Verfahren (und einige Varianten) und bewertet sie bezüglich der Fehlertoleranz, Nachrichtenkomplexität und Verzögerung bei der Migration.

- *Logging*
  Hier hinterlässt der Agent auf jeder Plattform, die er besucht, eine Spur, die zur nächsten Plattform führt. Um den Aufenthaltsort zu ermitteln, muss ein Agent einen der bisherigen Orte (etwa den Ursprungsort) kennen und dann diese Verweise bis zum Agenten weiter verfolgen.
- *Brute Force*
  Der Agent wird auf allen Plattformen gesucht. Diese Suche kann auch parallel auf mehreren Plattformen erfolgen.
- *Registrierung*
  Der Agent teilt einer zentralen Registrierungsstelle seinen jeweils aktuellen Ort mit. Über diese Zentrale ermitteln dann andere Agenten den Aufenthaltsort.

Für die Nachrichtenzustellung gibt es zwei Varianten:

- *Locate and Transfer*
  Erst ermittelt der Absender den Aufenthaltsort, dann sendet er die Nachricht dort hin.
- *Forwarding*
  Die Lokalisierung und das Zustellen der Nachricht finden in einem Schritt statt.

Ähnliche Methoden kommen auch bei der Suche nach Waisen zum Einsatz [Baumann 98b], [Beck 97]. Waisen sind Agenten, die nicht mehr benötigt werden, aber trotzdem unkontrolliert weiter ausgeführt werden. Verallgemeinert stellt sich hier das Problem der Kontrolle mobiler Agenten [Baumann 99b], wobei die Kommunikation und die Waisenerkennung eine zentrale Rolle spielen.

b) Mobilität

Die Mobilität kann durch verschiedene Faktoren eingeschränkt sein: Plattformen können kurzzeitig nicht erreichbar sein (da etwa der Rechner abgeschaltet ist), nur über dynamische Adressen verfügen, oder durch Firewalls abgeschottet sein.

Gerade für mobile Plattformen (etwa auf einem Notebook) kommen häufig Proxy-ähnliche Konstruktionen zum Einsatz, die Agenten entgegennehmen, wenn das eigentliche Ziel gerade nicht erreichbar ist. Beispiele sind die Agent-Box bei Aglets oder ein Docking-System wie bei AgentTcl. Allerdings stellt sich die Frage, ob spezielle Proxies überhaupt nötig sind: Jede gewöhnliche Plattform kann diese Aufgaben übernehmen. Auch ohne Unterstützung durch die Plattform kann ein Agent von sich aus auf eine Ausweichstation migrieren, falls sein Ziel gerade nicht verfügbar ist.

c) Management und Kontrolle

Agenten müssen auch kontrolliert werden können, während sie unterwegs sind. Die Anforderungen an die Agenten-Management-Applikation sind je nach Einsatzzweck sehr unterschiedlich. Einige Arbeiten (unter anderem [Avvenuti 98]) stellen eine Kombination mobiler Agenten mit Web-Techniken vor, in der die Plattform über HTML-Formulare und Java-Applets gesteuert wird. Ein Agent kann dabei sein eigenes GUI mit sich tragen, das dann ebenfalls als Applet im Browser des Administrators abläuft.

2.1.4 Potenzielle Vorteile

Neben den im Kapitel 1.3.2 *Vorteile im Netzmanagement*, Seite 8 beschriebenen Vorteilen lassen sich viele weitere nützliche Eigenschaften des Mobile-Agenten-Paradigmas aufführen. So listet [Bieszczad 98a] eine Reihe von potenziellen Vorteilen für alle geeigneten Einsatzzwecke auf, als Vergleichsmaßstab dient das klassische Client-Server-Modell. Eine Zusammenfassung dieser Vorteile ist in Tabelle 2.4 *Potenzielle Vorteile mobiler Agenten*, Seite 18 enthalten.

Potenzieller Vorteil	Begründung
Weniger CPU-Belastung	Die gesamte CPU-Belastung auf den beteiligten Rechnern ist geringer, da ein mobiler Agent zu einem Zeitpunkt nur auf einer einzelnen Plattform ausgeführt wird.
Geringerer Platzbedarf	Ein mobiler Agent belegt nur Platz auf einem einzelnen Rechner. Statische Server belegen dagegen fortwährend Platz auf allen Maschinen.
Weniger Netzbelastung	In vielen Anwendungen belegt der Code weniger Platz als die Daten, die er verarbeitet. Bei mobilen Agenten müssen die Daten vor ihrer Verarbeitung nicht übertragen werden.
Asynchrone autonome Interaktion	Es ist keine fortwährende Verbindung zum Absender des Agenten nötig.
Bessere Zusammenarbeit mit Realzeit-Systemen	Ein mobiler Agent wird näher am Realzeit-System platziert, dadurch verringern sich die Latenzzeiten. Dies wirkt sich vor allem in überlasteten Netzen positiv aus.
Robustheit und Fehlertoleranz	Wenn verteilte Systeme instabil werden, können mobile Agenten immer noch helfend eingreifen, wenn ein zentraler Ansatz keine Verbindung über das gestörte Netzwerk mehr aufnehmen kann.
Unterstützung heterogener Umgebungen	Die Agentenplattform abstrahiert bereits von der verwendeten Hardware.
Dienste können online ergänzt werden	Mobile Agenten können die Dienste einer Applikation im laufenden Betrieb erweitern und ergänzen oder neue Dienste hinzufügen.
Einfache Entwicklung	Verteilte Systeme in Form mobiler Agenten zu entwickeln ist relativ einfach, der schwierige Teil ist das Framework an sich. Wenn das Framework vorhanden ist und die Plattformen installiert sind, könnten die Agenten auch in RAD (Rapid Application Development)-Umgebungen entwickelt werden.
Einfache Upgrades	Mobile Agenten lassen sich einfach austauschen. Eine neue Server-Version zu installieren ist wesentlich komplizierter.

Tabelle 2.4 *Potenzielle Vorteile mobiler Agenten*

2.2 Sicherheit

Bei der Sicherheit von mobilen Agenten ergeben sich aus der Tatsache, dass Code aus eigener Motivation von einer Plattform zur nächsten migriert, deutlich andere Randbedingungen als bei klassischen Multi-User- und Client-Server-Architekturen. Den Sicherheitsarchitekturen klassischer Systeme liegen viele Annahmen zu Grunde, die bei mobilem Code und besonders bei mobilen Agenten nicht mehr gelten. [Chess 98] analysiert diese Änderungen; in Kapitel 3.1 *Alte Annahmen werden ungültig*, Seite 25 sind die wesentlichen Aussagen zusammengefasst.

Bereits die Authentifizierung stellt bei mobilen Agenten neue Anforderungen [Berkovits 98], [Schelderup 99]. Agenten bestehen nicht nur aus unveränderlichen Teilen wie ihrem Code und den konstanten Daten, die durch eine digitale Signatur bereits beim Absender geschützt werden können [Roth 98]. Ihr Zustand ändert sich von Plattform zu Plattform; durch unbefugte Änderungen am Zustand kann ein „guter“ Agent durchaus zu einem „schlechten“ mutieren. Dem ist durch einen Schutz bei der Übertragung und durch Vertrauensbeziehungen zwischen den Plattformen entgegenzuwirken. Jede bisher besuchte Plattform könnte den Agenten modifiziert haben [Chan 99], selbst wenn jede Übertragung abgesichert stattgefunden hat. Daher muss die aktuelle Plattform allen bislang besuchten Plattformen vertrauen [Schelderup 99]. Im einfachsten Fall gelingt das mit Single-Hop-Agenten, die nur genau einmal migrieren und immer von einer sicheren Basis aus zu ihrem Ziel wandern [Ordille 96], [Korba 99].

Bei mehreren beteiligten Hosts wird es um so schwieriger, den Agenten oder seinen Eigentümer zu authentifizieren. Die Plattform müsste sicherstellen, dass weder der Code noch der Zustand modifiziert wurden und dass die Wegewahl (Routing) des Agenten korrekt war, er also auf keiner Plattform war, zu der er nicht hätte migrieren dürfen [Swarup 97], [Wilhem 98]. Aus dem tatsächlich gewählten Weg lassen sich Rückschlüsse ziehen, wie vertrauenswürdig der Agent noch ist [Knoll 01].

In der Sicherheitsproblematik unterscheidet [Farmer 96] zwischen unlösbaren, trivialen und schwierigen, aber lösbaren Fragen. In der Praxis zeigt sich allerdings, dass selbst die trivial anmutenden Aufgaben eine hohe Komplexität aufweisen.

- *Unlösbare Fragen*
  Wurde die Plattform angegriffen? Wird sie den Agenten korrekt ausführen? Bis zum beabsichtigten Ende? Und ihn dann zu seinem gewünschten Ziel transportieren? Können Code und Daten des Agenten geheim gehalten werden? Speziell auch kryptografische Geheimnisse (Schlüssel)? Kann die Kommunikation geheim bleiben? Lässt sich ein Agent von einem geklonten Duplikat unterscheiden?
- *Triviale Fragen*
  Können Autor und Absender authentifiziert werden? Kann die Integrität des Codes geprüft werden? Können Plattformen den Agenten während der Migration vor dem Ausspähen schützen? Können sich Plattformen vor Agenten schützen?
- *Schwierige, aber lösbare Fragen*
  Kann eine Sprache genutzt werden, in der alle Programme sicher sind? Kann ein Absender die Flexibilität seines Agenten einschränken? Kann eine Plattform sicherstellen, dass ein Agent in einem sicheren Zustand ist? Kann der Absender festlegen, welche Plattformen sein Agent besuchen darf?

Im Rahmen dieser Arbeit werden einige dieser sehr allgemeinen Fragen nicht weiter verfolgt, da sie in dem Netzmanagement-Szenario irrelevant sind. Zu den Gründen gibt Kapitel 3 *Bedrohungen*, Seite 25 näher Auskunft.

[Abdalla 97] teilt die Punkte, an denen Sicherheitsvorkehrungen zu treffen sind, in vier Gruppen ein: Schutz der Plattformen vor den Agenten, Schutz der Agenten voreinander, Schutz der Agenten vor den Plattformen und Schutz einer Gruppe von Plattformen vor einem Agenten.

2.2.1 Schutz der Plattformen vor den Agenten

Dieser Aspekt ähnelt dem Problem klassischer Betriebssysteme ([Greenberg 98]; siehe aber auch 3.1 *Alte Annahmen werden ungültig*, Seite 25). [Ousterhout 98] unterscheidet folgende Angriffe:

- Verletzung der Integrität
- Verletzung der Vertraulichkeit
- Denial of Service

Diese Liste lässt sich weiter aufschlüsseln; [Greenberg 98] nennt zusätzlich folgende Punkte:

- Zerstörung
- Ärgernis (etwa durch wiederholte Angriffe)
- Social Engineering (Manipulation von Personen, Hosts oder Agenten)
- Logische Bomben, Event-gesteuerte Angriffe
- Zusammengesetzte Angriffe

2.2.2 Schutz der Agenten voreinander

Auch dieser Punkt erinnert an die Situation in klassischen Betriebssystemen, in denen die Prozesse voneinander getrennt werden. Bei Agentensystemen spielt vor allem die Kommunikation zwischen den Agenten eine wichtige Rolle. Entsprechend sind hier geeignete Mechanismen nötig, die die Agenten sicher und effizient miteinander kommunizieren lassen, ohne dass sie dadurch angreifbar werden.

2.2.3 Schutz der Agenten vor den Plattformen

Der Schutz der Agenten vor den Plattformen, auf denen sie ausgeführt werden, ist die Achillesverse für viele Einsatzzwecke mobiler Agenten. Es gibt zwar einige Ansätze, die diesen Schutz unter bestimmten Rahmenbedingungen bereitstellen. Dabei ist zwischen der Integrität der Berechnungen (korrekte Ausführung) und der Vertraulichkeit (Plattform weiß nicht, was der Agent berechnet) zu unterscheiden [Yee 97]. Allgemeinen ist dieser Schutz aber kaum zu erreichen [Chess 98] und wird oft als unmöglich bezeichnet [Busse 98].

Ansätze, die den Schutz rein in Software zu realisieren versuchen, sind:

- *Code Mess Up, Blackbox Security*
  Der Programmcode wird so unübersichtlich gestaltet, dass die automatisierte Analyse mehr Zeit in Anspruch nimmt als die Lebenszeit des Agenten beträgt. [Hohl 97a] [Hohl 98a] [Hohl 00] [Wang 00]
- *Registrierungsprotokoll*
  Eine Registierungsinstanz überwacht die Ausführung des Agenten. Dieser kann die Eingabedaten für eine Funktion zusammen mit dem erwarteten Ergebnis ablegen. Nur wenn das Ergebnis bei jeder Berechnung identisch zu dem registrieren Ergebnis ist, wird der Agent weiter ausgeführt. Damit sollen Blackbox-Tests unterbunden werden. [Hohl 99]
- *Nachträgliche Erkennung von Modifikationen*
  Durch zusätzlich eingefügte Dummy-Objekte, die bei regulärer Bearbeitung nicht modifiziert werden, soll eine unerwünschte Veränderung erkennbar werden. [Meadows 97]

- *Snapshots, Traces*
  Ein interessanter Ansatz fügt Assertions in den Agenten-Code ein und speichert Momentaufnahmen des Zustands (Snapshots), um dadurch im Nachhinein ein eventuelles Fehlverhalten der Plattformen zu erkennen und die Vertrauenswürdigkeit der Ergebnisse zu beurteilen. Dieser Ansatz verhindert jedoch keine Manipulation. [Kassab 98] [Vigna 97] [Vigna 98a]
- *Verschlüsselte Funktionen, Function Hiding*
  Eine verschlüsselte Funktion berechnet mit ebenfalls verschlüsselten Daten ein verschlüsseltes Ergebnis. Ein ausführender Host kann zwar den ganzen Vorgang beobachten, ihn aber nicht verstehen, da an keiner Stelle Klartextdaten vorliegen und auch die durchgeführte Berechnung nicht nachvollziehbar ist. Dies ist jedoch nur mit bestimmten Arten von Funktionen möglich. [Sander 97a] [Sander 97b] [Sander 98a] [Sander 98b] [Kotzanikolaou 00]
- *Supervisor-Worker-Pattern*
  Ähnlich dem Master-Slave-Prinzip besteht dieses Modell aus zwei Agenten, dem Supervisor und dem Worker. Der Supervisor migriert ausschließlich zu vertrauenswürdigen Hosts und hält dort alle sicherheitskritischen Daten vor. Der Worker erhält nur ein Minimum an Informationen, migriert damit zu den gefährlichen Hosts und meldet alle Ergebnisse an den Supervisor. [Fischmeister 99]

Vergleiche einiger Schutztechniken finden sich beispielsweise in [Yee 97] und [Hohl 00]. Das Angriffsmodell von [Hohl 98c] könnte dazu dienen, verschiedene Schutzmechanismen auf ihre Wirksamkeit hin zu untersuchen.

Ein praktikabler Schutz ist noch am ehesten von einer vertrauenswürdigen Plattform zu erwarten, die auf einer vertrauenswürdigen Hardware läuft (Trusted Computing Base) [Wilhem 98], [Wilhem 99]. Java-Smartcards stellen hier einen interessanten Ansatz dar [Fünfrocken 99b].

2.2.4 Schutz einer Gruppe von Plattformen vor einem Agenten

Die isolierte Betrachtung einzelner Plattformen ist dann nicht mehr ausreichend, wenn ein Agent innerhalb einer Gruppe von Plattformen insgesamt nur eine bestimmte Menge einer Ressource benutzen darf. Ein weiteres Problem könnte sein, dass ein Agent durch das Zusammenfügen von einzeln nutzlosen Informationen, die er auf mehreren Plattformen sammelt, insgesamt zu mehr Wissen gelangt als ihm zugestanden wird. In diesen Fällen ist eine Kooperation der Plattformen nötig. Digitales Geld könnte zur Abrechnung der Dienste und Ressourcen eingesetzt werden. [Gray 97b] [Gray 98a] [Hohl 00]

2.2.5 Weitere

Verschiedene Autoren erkennen auch weitere Bedrohungsfelder, etwa die zwischen Hosts und unautorisierten Dritten [Rothermel 97a]. Dies verdeutlicht, dass eine detaillierte Bedrohungsanalyse, die alle beteiligten Instanzen mit einbezieht, notwendig ist (siehe Kapitel 3 *Bedrohungen*, Seite 25).

2.3 Bestehende Agentensysteme

Sowohl in der Industrie als auch in der Forschung wurden in den letzten Jahren eine Reihe von Plattformen für mobile Agenten entwickelt. Diese Entwicklungen verfolgen allerdings unterschiedliche Ziele und sind nur selten für das Netzmanagement ausgelegt. Der Aspekt Sicherheit ist unterschiedlich stark ausgeprägt.

Zu den unabhängig entwickelten Plattformen kommen einige Bestrebungen, Standards für mobile Agenten einzuführen [Dickinson 97]. Parallel dazu wird an einigen Stellen versucht, zwischen bestehenden, an sich inkompatiblen Agentensystemen Interaktionen zu ermöglichen [Dömel 97]. [Misikangas 00] beschreibt einen Ansatz, Agenten auch ohne einen gemeinsamen Standard zwischen inkompatiblen Plattformen migrieren zu lassen. In diesem Fall trennen die Autoren einen Agenten in Kopf und Körper, wobei sie den plattformunabhängigen Kopf auf verschiedene plattformabhängige Körper setzen, die auf das jeweilige Plattform-API passen. Der Kopf trägt mehrere Körper mit sich und benutzt den jeweils passenden.

Im Folgenden kann nur ein Teil der bekannten Systeme beschrieben werden. Es wurde versucht, einen Überblick über die wichtigsten Architekturen zu geben. Ein Vergleich verschiedener Agentensystemen und speziell deren Sicherheitsmechanismen ist beispielsweise in [Görl 98], [Karnik 99b] oder [Lugmayr 99] zu finden.

2.3.1 Standards für mobile Agenten

Im Wesentlichen sind zwei Standards in der Entwicklung, die Agenten, speziell auch mobile Agenten, behandeln [Dickinson 97]: MASIF und FIPA.

Die Mobile Agent System Interoperability Facility MASIF ist ein Standard der OMG (Object Management Group), von der unter anderem der CORBA-Standard stammt (Common Object Request Broker Architecture). MASIF ist der Nachfolger von MAF (Mobile Agent Facility) [OMG 97]. MASIF standardisiert die folgenden vier Bereiche [Lange 98a] und behandelt dabei auch Sicherheitsfragen [Milojicic 98a]:

- Agenten-Management
- Agenten-Transfer (Migration)
- Namen für Agenten und Agentensysteme
- Syntax für Abfrage von System-Typ und -Ort

MASIF benutzt CORBA für viele dieser Aufgaben, etwa als Namensdienst oder für Security Services [Milojicic 98b]. MASIF standardisiert aber nicht die Systemtypen, Sprachen, Serialisierungsmechanismen oder Authentifizierungsmethoden. Der Standard soll lediglich die Interoperabilität zwischen den verschiedenen Systemen sicherstellen, etwa zwischen Aglets, MOA und AgentTcl.

Die Foundation for Intelligent Physical Agents FIPA startete im Dezember 1995. Als Organisation aus akademischen und industriellen Mitgliedern versucht sie, Standards für das äußerliche Verhalten von sowie Schnittellen zu Agenten und Agentensystemen zu definieren. Zu diesem Zweck wurden einige Referenzanwendungen definiert. Der Standard umfasst vor allem eine Agent Communication Language (ACL), das Management von Agenten sowie die Integration von Agenten und herkömmlicher Software.

2.3.2 Kommerzielle Agentensysteme

Viele Plattformen wurden vor oder parallel zu den Standards entwickelt. Da MASIF und FIPA vor allem die Kommunikation, Schnittstellen und das äußere Verhalten betreffen, lässt sich die Standardkonformität häufig nachrüsten. So stehen für das kommerzielle Mobile-Agenten-System Grasshopper (Hersteller: IKV++) zwei optionale Softwarekomponente zur Verfügung, die für MASIF- oder FIPA-Kompatibilität sorgen.

Ebenfalls aus dem kommerziellen Umfeld stammt das Aglets-System der IBM. Interessant an Aglets ist vor allem die umfangreiche Konfigurations- und Berechtigungssprache. Das Aglets-Sicherheitsmodell unterscheidet zwischen Prinzipalen (identifizierbare Entitäten) und Identitäten (Name und eventuell weitere Attribute). Aus diesen Prinzipalen leiten sich die jeweiligen Rechte (Privileges) ab. Die dazu nötige Sicherheits-Policy wird in einer eigenen Konfigurationssprache definiert.

Der Klassiker unter den Systemen mobiler Agenten ist Telescript von General Magic. Dieses System bietet starke Migration. Es verwendet dazu eine eigene Programmiersprache, die an Java und C++ angelehnt ist. Sicherheit gilt als wichtiger Designfaktor: Die Rechte der Agenten sind anhand von Permits festgelegt, mit Permits lassen sich auch Resource Limits regeln. General Magic hat die proprietäre Programmiersprache im Nachfolgeprodukt Odyssey durch das verbreitete Java 1.1 ersetzt. Obwohl sich das Design stark an Telescript orientiert, kann das neuere System nur noch schwache Migration bieten.

Größere Bekanntheit haben auch die Concordia-Plattform von Mitsubishi sowie Jumping Beans erlangt. Beide Systeme basieren auf Java. Nähere Details zu den genannten kommerziellen Mobile-Agenten-Systemen sind im Anhang E.1 *Kommerzielle Agentensysteme*, Seite 155 zu finden. Auch in Forschung und Wissenschaft entstand eine ganze Reihe von Agentensystemen, siehe Anhang E.2 *Forschungsprojekte*, Seite 158.

2.3.3 Forschungsprojekte

Eines der ersten frei verfügbaren Mobile-Agenten-Systeme wurde am Dartmouth College in Hanover, USA entwickelt. Die ursprüngliche Version AgentTcl (später D'Agents) benutzt Tcl als Programmiersprache für Agenten. Der Tcl-Interpreter wurde dazu um die Fähigkeit erweitert, den Zustand eines Agenten zu speichern und wieder herzustellen. Die Agenten laufen in eigenen Prozessen und nicht, wie bei den meisten Plattformen, in Threads. Die Architektur kann dadurch auch verschiedene Interpreter-Sprachen für Agenten unterstützen.

Dieses Versprechen setzt die neuere Version (D'Agents) in die Realität um: Neben Tcl sind nun auch Scheme und Java als Sprache möglich. Beide Systeme greifen auf das externe PGP-Programm zurück, um Verschlüsselung und Signaturen zu berechnen und zu verifizieren. Agenten können sich gegenüber der ersten Plattform mit einer PGP-Signatur authentifizieren. Alle folgenden Plattformen müssen ihren Vorgängern vertrauen. Verlässt ein Agent eine Gruppe von Plattformen, die sich gegenseitig vertrauen, wird er zu einem anonymen Agenten. Um auch für Java starke Migration zu ermöglichen, verwendet D'Agents eine modifizierte JVM.

Die relativ selten angebotene starke Migration ist auch bei Ara zu finden. Diese Plattform unterstützt Tcl, Java und sogar C/C++. Letztere wird dazu in eine Bytecode-Zwischensprache übersetzt. Ara-Agenten können aber auch aus nativem Code bestehen, wenn die Sicherheitsanforderungen dies zulassen. Trotzt der verschiedenen Sprachen arbeitet Ara mit Multi-Threading. Ara

unterscheidet bei der Authentifizierung zwischen dem Hersteller (Programmierer), der den Code signiert, und dem Besitzer, der die Parameter und die Allowances signiert. Ein Ausweis (Passport) enthält die Allowances eines Agenten, seine global eindeutige ID sowie die Zertifikate und Signaturen des Herstellers und des Besitzers.

Recht ungewöhnlich sind auch die Architekturen von ffMAIN und TACOMA. Die in Frankfurt am Main entwickelte Agentenplattform ffMAIN benutzt HTTP als Migrationsprotokoll. Die Agenten können in Tcl oder Perl geschrieben sein. Die Spezialität von TACOMA sind die vielen unterstützten Programmiersprachen: C/C++, ML, Perl, Python und weitere. Jeder Agent läuft in diesem System in einem eigenen Unix-Prozess. Die Agenten sind jedoch selbst dafür verantwortlich, ihren Code und ihren Zustand in einem Folder zu speichern. Diesen Folder überträgt das Framework dann bei der Migration zum Zielhost.

Die meisten Agentensysteme beschränken sich auf Java als Implementierungssprache für die Plattform und die Agenten und bieten schwache Migration. Beispiele sind MAP (für das Netzmanagement entwickelt), MASA (kompatibel zu MASIF), MAMAS (MASIF-kompatibel und für das Systemmanagement gedacht), MOA (hält Kommunikationskanäle über die Migration hinweg offen), SAE/WASP (Agentenerweiterung für Webserver) und viele mehr.

Ajanta-Agenten laufen in ihrer eigenen Thread Group und werden über ihren eigenen Classloader geladen. Durch dieses Verfahren identifiziert Ajanta die Agenten und stellt ihnen eine Protection Domain zur Verfügung. Der Zugang zu Systemressourcen ist durch Proxy-Objekte abgesichert. Bei diesem Ansatz kann das Zugriffsrecht einem Agenten auch nachträglich wieder entzogen werden. Ajanta enthält auch einige Vorkehrungen, die Agenten vor der Plattform schützen sollen: Nur lesbarer Zustand (Read-Only), nur anfügbare Logs (Append-Only) und nur für bestimmte Plattformen lesbare Daten.

Gypsy ist ebenfalls Java-basiert, benutzt aber Java Beans für die Agenten und überträgt sie als Jar-Archiv. Es bietet mehrere Verbreitungstechniken (RMI, E-Mail+PGP und SSL-gesicherte Kommunikation) in Form eigener Communicator-Agenten. Gypsy enthält das Konzept von kooperierenden Supervisor- und Worker-Agenten: Der Supervisor migriert und startet bei Bedarf seine Worker-Agenten. Interessant ist die Art, wie Gypsy zusätzliche Programmiersprachen integriert: Die Interpreter sind selbst in Java programmiert und werden in der JVM ausgeführt. Gypsy legt besonderen Wert darauf, dass für die meisten Aufgaben des Systems Agenten eingesetzt werden, also das Agenten-Paradigma möglichst universell genutzt wird.

Jeder Agent ist in Mole durch eine globale, eindeutige ID gekennzeichnet. Die RMI-Kommunikation findet in Sessions statt. Alternativ können die Agenten in einer Session auch asynchrone Nachrichten übertragen. Mole bietet auch anonyme Kommunikation zwischen Gruppen von Agenten. Durch einen eigenen Scheduler für Java-Threads kann Mole ein Accounting von CPU-Zeit, Netzwerk-Kommunikation, Anzahl erzeugter Agenten und der Gesamtzeit im Ort vornehmen.

Kapitel 3

Bedrohungen

Nach den Grundlagen und Beispielen von Mobile-Agenten-Systemen im vorherigen Kapitel geht dieses näher auf die Bedrohungen ein, denen ein Agentensystem ausgesetzt ist. Auf eine Aufstellung von nicht mehr geltenden Annahmen folgt eine Analyse, welche Instanzen an einem typischen Szenario beteiligt sind. Daraus lässt sich ableiten, mit welchen Angreifern zu rechnen ist und an welchen Stellen diese eingreifen können.

Die einzelnen Bedrohungen werden dann bezüglich ihrer Bedeutung für das Netzmanagement bewertet. In diesem Anwendungsfall sind einige Freiheitsgrade nicht relevant, beispielsweise muss eine Netzmanagement-Plattform keine anonymen Agenten ausführen. Am Ende des Kapitels werden verschiedene Mechanismen erarbeitet, die einen wirksamen Schutz vor den Bedrohungen bieten.

3.1 Alte Annahmen werden ungültig

Kapitel 2.1.1 *Mobiler Code*, Seite 13 hat gezeigt, wie sich die wesentlichen Varianten mobilen Codes unterscheiden. Jede dieser Varianten bringt neue Rahmenbedingungen für die Sicherheit eines Systems mit sich und führt neue Instanzen ein, die bei klassischen Software-Paradigmen nicht vorhanden sind. [Chess 98] fasst die aus den Randbedingungen abgeleiteten Annahmen zusammen, die ihre Gültigkeit bei mobilem Code verlieren.

3.1.1 Zuordnung von Personen zu Programmen (Identitätsannahme)

Die Identitätsannahme besagt, dass sich jede Aktion eines Programms einer Person zuordnen lässt und diese Person die Aktion auch beabsichtigt. Daraus wird abgeleitet, dass die Rechte des Programms den Rechten der zugeordneten Person entsprechen. Außerdem folgt, dass nur Personen, die dem System bekannt sind, Programme ausführen dürfen. Jedem Benutzer werden die Rechte zugeteilt, die für alle seine Programme gelten sollen. Als Nebenaspekt lässt sich daraus ableiten, dass auf Single-User-Systemen keinerlei Sicherheitsvorkehrungen nötig sind.

Bei mobilem Code kommen dagegen viele Programme aus unbekannten und eventuell nicht vertrauenswürdigen Quellen. Wenn ein Programm eine Aktion ausführt, kann es unmöglich sein, diese Aktion einer bestimmten Person zuzuordnen. Selbst wenn eine Person für das Programm zuständig ist, ist sie häufig auf der Plattform unbekannt, die den Code ausführt.

3.1.2 Trojanische Pferde sind selten

Die Annahme, dass alle Aktionen eines Programms von seinem Benutzer auch beabsichtigt sind, war im Grunde nie vollständig korrekt. Neben unabsichtlichen Fehlfunktionen verletzen vor allem trojanische Pferde diese Annahme. Ein Programmierer kann in sein Programm beliebige versteckte Funktionen integrieren. Lediglich die Tatsache, dass trojanische Pferde eher selten vorkommen, macht die Identitätsannahme anwendbar. Dazu kommt eine Annahme über den Ursprung der Programme: Im Wesentlichen stammen alle Programme aus leicht zu identifizierenden und vertrauenswürdigen Quellen. Enthält ein Programm versteckte Funktionen, kann sein Urheber ausfindig gemacht und belangt werden. Diese Annahme gilt auch umgekehrt: Der Programmierer vertraut dem Anwender, dass dieser das Programm nur entsprechend den Lizenzbedingungen nutzt. In beiden Fällen unterstützen gesetzliche Regelungen das Vertrauensverhältnis.

Da es bei mobilem Code unmöglich sein kann, eine Person zuzuordnen und für Schaden zu belangen, steigt die Wahrscheinlichkeit, dass die Programme unerwünschte Funktionen aufweisen. Was in herkömmlichen Systemen Viren und Würmer vorführen, ist bei mobilem Code bereits im System immanent: Code verbreitet sich selbständig. Es kann sein, dass keine beteiligte Person eine Aktion gewollt hat. Auch hat der Programmierer keinen Einfluss mehr auf die Anwender.

3.1.3 Quelle der Angriffe

Aus der Identitätsannahme und den Annahmen über den Ursprung der Software lässt sich ableiten, dass die meisten Bedrohungen von Angreifern stammen, die Programme nutzen, um unberechtigten Zugriff zu erlangen. Dieser Zugriff kann durch das Annehmen einer falschen Identität gelingen, oder durch Umgehen der Sicherheitsschranken des Systems. Die am weitesten verbreiteten Sicherheitssysteme konzentrieren sich daher auf sichere Authentifizierung und auf User-basierte Zugriffskontrollen.

Authentifizierung ist bei mobilem Code häufig unmöglich, etwa weil die Person unbekannt ist oder es generell beabsichtigt ist, dass der Code anonym bleibt. Damit versagt auch die User-basierte Zugriffskontrolle. Selbst wenn ein Benutzer zugeordnet werden kann, wird das Programm auch Aktionen ausführen, die der Benutzer nicht beabsichtigt hat, die er nicht kontrollieren kann und von denen er nichts weiß.

3.1.4 Programme bleiben, wo sie sind

Diese Annahme wird meist implizit getroffen: Programme bleiben in der Regel innerhalb einer administrativen Zone, also beispielsweise auf einem Rechner oder in einem Rechner-Pool. Dies ändert sich nur, wenn Anwender oder Administratoren absichtlich ein Programm übertragen, etwa durch Kopieren oder Installieren. Für Prozesse (Programme, die gerade ausgeführt werden) gilt diese Annahme verstärkt. Man geht auch davon aus, dass ein Programm immer auf dem selben Betriebssystem läuft, und dieses Betriebssystem für die Sicherheit zuständig ist.

Würmer verletzen diese Annahme auch in klassischen Systemen. Bei mobilen Agenten ist dies die wesentliche Eigenschaft, die diese Systeme ausmacht: Die Programme (und Prozesse) überschreiten administrative Grenzen, und dies aus eigenem Antrieb ohne Einwirkung eines Benutzers. Dabei wird der Code auf unterschiedlichen Betriebssystemen ausgeführt. Das Betriebssystem fällt somit als Verantwortlicher für die Sicherheit aus.

3.1.5 Weitere geänderte Rahmenbedingungen

[Fong 98] beschreibt weitere Rahmenbedingungen, die im Widerspruch zu bisherigen Sicherheitssystemen stehen. Ein wichtiger Aspekt ist, dass Plattformen für mobile Agenten in der Regel als einzelner Prozess innerhalb eines Host-Betriebssystems ausgeführt werden, die einzelnen Agenten sind dann als Threads innerhalb dieses Prozesses realisiert. Damit tritt die Situation ein, dass sich Teile eines Prozesses gegenseitig nicht vertrauen. Im klassischen Betriebssystem gehört jeder Prozess einem einzelnen Benutzer, die Teile eines Prozesses (und damit auch die Threads) sind kooperativ und benötigen keinen Schutz voreinander. Mobile Agenten verletzen offensichtlich diese Annahme.

Die Autoren führen die Analogie eines Untermieters ein: Aus Sicht des Betriebssystems ist jeder Prozess ein Mieter, mobiler Code und im speziellen mobile Agenten entsprechen einem Untermieter.

3.2 Beteiligte Instanzen

Die neuen Randbedingungen für die Sicherheitsarchitektur, die bei mobilen Agenten gelten, erfordern eine grundlegende Analyse der Bedrohungen und eine darauf abgestimmte Architektur. Da viele Annahmen nicht mehr gelten, ist zunächst zu untersuchen, welche Instanzen an Mobile-Agenten-Systemen beteiligt sind. Eine einfache Auflistung allgemeiner Bedrohungen (etwa [Bryce 99a]) erscheint als nicht ausreichend. Auch Modelle, die nur die Plattformen und die Agenten betrachten [Jansen 99a] [Jansen 00], vereinfachen die Realität zu stark. Die Darstellung in [Vitek 97a] und [Vitek 97b] berücksichtigt neben Agenten und Plattformen immerhin das Netzwerk und die verschiedenen Kommunikationsbeziehungen. Alle diese Ansätze führen zwar zu einer einfachen und übersichtlichen Illustration der Grundbedrohungen, lassen aber einige Wechselwirkungen nicht erkennen. Es gilt daher, die möglichen Angriffspunkte systematisch zu untersuchen [Busse 98] und bezüglich ihrer Relevanz zu beurteilen.

Durch die Beschränkung auf das Anwendungsfeld „Netzmanagement" in dieser Arbeit ist es möglich, einige der Freiheitsgrade wieder einzuschränken. Das Ziel ist dabei, die Sicherheit des Gesamtsystems besser beherrschbar zu gestalten. Je mehr es den etablierten Modellen gleicht, um so verständlicher und leichter administrierbar ist es.

Abbildung 3.1 *Überblick über die beteiligten Instanzen*, Seite 28 zeigt, welche Instanzen an einem System mobiler Agenten beteiligt sind und wie sie zusammenhängen.

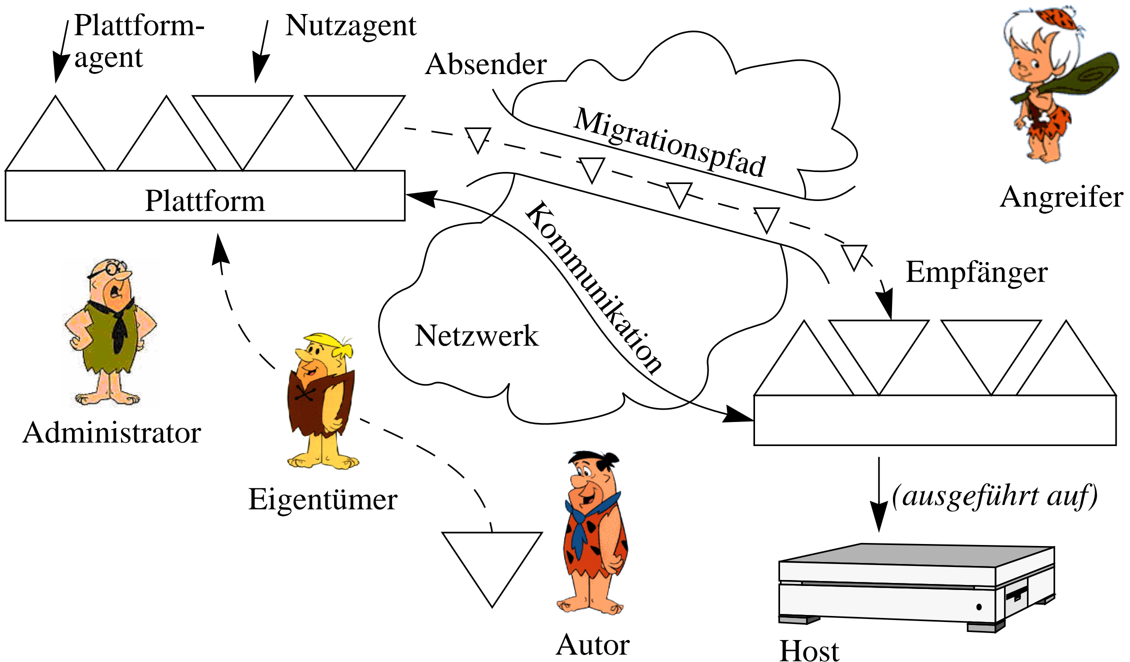

Abbildung 3.1 *Überblick über die beteiligten Instanzen*

Die einzelnen Instanzen erfüllen dabei folgende Aufgaben:

- *Nutzagent:* Mobiler Agent, der im Auftrag seines Eigentümers Aufgaben erledigt und dabei von Plattform zu Plattform migriert.
- *Plattformagent* oder Systemagent: Spezieller Agent, der die Plattform um zusätzliche Funktionen ergänzt.
- *Autor:* Programmiert den Agenten.
- *Eigentümer:* Konfiguriert den Agenten und schickt ihn auf seinen Weg.
- *Plattform:* Ablaufumgebung für die Agenten.
- *Host:* Führt die Plattform aus.
- *Administrator:* Installiert und konfiguriert die Plattform.
- *Netzwerk:* Verbindet die Plattformen und transportiert die Agenten.
- *Migrationspfad:* Weg durch das Netz, auf dem sich der Agent bewegt.
- *Absender:* Sendet den Agenten über das Netz zum Empfänger.
- *Empfänger:* Nimmt den Agenten entgegen und führt ihn aus.
- *Kommunikation:* Agenten und Plattformen kommunizieren miteinander und mit anderen Diensten.
- *Angreifer:* Diese Rolle kann jede Instanz übernehmen.

3.3 Angreifer

Angriffe können im vorliegenden Szenario von allen Instanzen aus erfolgen. Die nachfolgende Aufstellung ist nicht als vollständige Liste aller möglichen Angriffe gedacht. Sie enthält zum Beispiel bewusst keine Hinweise auf Denial-of-Service-Angriffe. Solche Attacken können von jeder der genannten Instanzen aus erfolgen.

a) Autor

Der Autor eines Agenten kann Funktionen implementieren, die dem Eigentümer (ersten Absender) nicht bekannt sind. Dies entspricht dem klassischen trojanischen Pferd. Selbst wenn der Autor eigentlich vertrauenswürdig ist, könnte er selbst Opfer eines anderen Angriffs geworden sein. Der von ihm ausgelieferte Code könnte in der Folge Funktionen enthalten, die dem Autor selbst nicht bekannt sind. Möglich wären solche Angriffe beispielsweise mithilfe eines modifizierten Java-Compilers. Als Schadensfunktionen sind alle Angriffe denkbar, die der Agent selbst auch ausführen kann.

b) Agent (Nutzagent und Plattformagent)

Ein Agent kann in erster Linie andere Agenten angreifen. Bei Java-basierten Agentensystemen ist es üblich, dass alle Agenten einer Plattform in einer gemeinsamen JVM ausgeführt werden. Daher sind Angriffe auf parallel laufende Agenten besonders naheliegend. Die Angriffe können vom Ausspähen und Manipulieren des Zustandes (Daten) bis zum Eingriff in den Programmablauf (etwa durch einen einfachen Methodenaufruf) führen.

Auch die Plattform ist Angriffen durch Agenten ausgesetzt, die auf ihr laufen. Sogar der darunter liegende Host ist gefährdet. Da die Plattform lokal abläuft, greifen Host-basierte Netzwerk-Zugriffskontrollen (etwa Firewalls) nicht. Falls ein Agent auf einer Plattform ausgeführt wird, kann er einerseits die Zugriffskontrollen mit den Rechten des Hosts umgehen, und zusätzlich direkt den Host angreifen. Viele bekannte Angriffe sind nicht über das Netzwerk, sondern nur vom lokalen Rechner aus möglich, so dass das Gefährdungspotenzial im Vergleich zu Angriffen von außen zusätzlich erhöht ist.

Über das Netzwerk sind auch Angriffe auf entfernte Hosts und Plattformen möglich. Die Quelle dieser Angriffe ist der Host, auf dem die Plattform ausgeführt wird. Wichtig ist dies wieder bei Firewalls, da der Host eventuell Zugriff auf Systeme hat, die von außen (über das Internet) nicht zugänglich sind. Das ist auch der wesentliche Unterschied zu beliebigen Angriffen aus anderen Teilen des Netzwerks.

Die Endpunkte des Migrationspfades liegen in der Plattform beziehungsweise einem ihrer Plattformagenten. Diese Endpunkte (sowohl als Socket innerhalb der JVM als auch die Daten auf dem Netzwerk) stellen ebenfalls mögliche Angriffsziele dar.

c) Eigentümer

Der Eigentümer des Agenten kann seine Rechte missbrauchen, um zum Beispiel „böswillige“ Agenten einzuspielen. Er kann auch durch eine gezielt manipulative Parametrisierung einen eigentlich „gutwilligen“ Agenten zu unerwünschten Aktionen bewegen.

d) Plattform

Die Plattform hat naturgemäß vollen Zugriff auf den Code und den Zustand der ausgeführten Agenten. Sie interpretiert jede einzelne Anweisung des Programmcodes und führt sie aus. Ein Schutz der Agenten vor Angriffen der Plattformen scheint daher kaum oder nur sehr eingeschränkt möglich (siehe 2.2.3 *Schutz der Agenten vor den Plattformen*, Seite 20).

Neben direkten Angriffen auf die Agenten können auch Informationen, die die Agenten über ihre Umgebung erhalten, manipuliert sein. Weiterhin kann die Plattform das Migrationsziel modifizieren und den Agenten zum falschen Zeitpunkt ausführen oder ihn klonen.

e) Administrator

Der Administrator der Plattform konfiguriert unter anderem die Sicherheitsparameter. Er könnte sie etwa vollständig abschalten oder die Vertrauensbeziehungen und Rechtezuweisungen manipulieren. Da der Administrator die Plattform auch installiert, kann er sie beliebig modifizieren.

Der Autor der Plattform hat ähnlichen Einfluss wie der Autor des Agenten. Dies entspricht aber dem Problem bei herkömmlicher Software. Bei der Plattform ist noch menschliche Interaktion zur Installation und zum Start nötig, während der Agent autonom agiert, daher ist nur bei Agenten der Autor als eigene Instanz relevant.

f) Host

Die Sicherheit der Plattform (und damit ihrer Agenten) hängt generell von der Betriebssystemsicherheit des Hosts ab. Wurde der Host angegriffen, dann sind auch lokale Angriffe auf die Plattform möglich. Hier muss das Betriebssystem für entsprechende Schutzmechanismen sorgen. Zudem gilt für den Host ähnliches wie für die Plattform: Der Host führt letztlich alle Befehle aus und kann daher die Plattform, ihre Agenten und ihre Sicht auf die Umgebung beliebig manipulieren. Bei Multi-User-Systemen sind auch Angriffe von anderen, gleichzeitig auf dem Host arbeitenden Benutzern möglich.

g) Migrationspfad

Die klassischen Angriffe aus der Netzwerksicherheit (Abhören, Modifizieren von Code und/ oder Zustand, Unterschieben etc.) sind entlang des Migrationspfades denkbar. Bei Shared-Medium-Netzwerken (wie Ethernet) ist der Pfad unter Umständen ziemlich breit, so dass viele Hosts den Datenstrom abhören oder manipulieren können.

h) Netzwerk/Kommunikation

Von beliebigen Stellen im Netzwerk ausgehend, nicht nur entlang des Migrationspfades, können Angriffe auf den Host oder die Plattform durchgeführt werden. Agenten sind unmittelbar gefährdet, wenn sie als Server fungieren; etwa beim Web Agent ist dies der Fall.

3.4 Relevanz für Anwendungen im Netzmanagement

Bei der Menge beteiligter Instanzen und der Anzahl potenzieller Angreifer ist es notwendig, eine Bewertung der einzelnen Risiken bezüglich ihrer Relevanz vorzunehmen. Es stellt sich die Frage, welcher Schutz vor welchen Angreifern wirklich nötig ist. Tabelle 3.1 *Relevanz möglicher Angreifer*, Seite 31 enthält eine solche Bewertung.

Zusammenfassend ergibt sich, dass vor allem der Schutz der Plattform und der Agenten vor anderen Agenten und vor Angriffen von außen, sowie der Schutz der Agenten während der Übertragung relevant sind.

Angreifer	Relevanz	Kommentar
Autor	Hoch	Administratoren und Benutzer werden häufig bestehende Agenten einsetzen, in der Regel aber nicht über die Fähigkeit und Zeit verfügen, die Quelltexte der Agenten zu überprüfen (Auditing).
Agent	Hoch	Die Agenten sollen Aufgaben im System- und Netzmanagement übernehmen, dazu benötigen sie weitreichende Eingriffsmöglichkeiten in die jeweiligen Systeme. Entsprechend hoch ist das Risiko, das von einem nicht autorisierten Agenten ausgeht.
Benutzer	Mittel-Hoch	Je nachdem, ob nur vertrauenswürdige Anwender das System nutzen dürfen oder beliebige User, müssen auch von legitimen Benutzern Angriffe erwartet werden.
Plattform	Niedrig	Kann als sicher modelliert werden. Sie ist nicht anders zu bewerten als herkömmliche Serversoftware. Voraussetzung ist, dass die Agenten nur auf Plattformen migrieren, denen sie auch vertrauen.
Administrator	Niedrig	In der Regel ist der Administrator der Plattform in Personalunion auch Nutzer des Agentensystems. Im Netz- und Systemmanagement wird der Administrator gerne als vertrauenswürdig angenommen, er hat alle Rechte. Daher geht von ihm auch kaum eine Gefahr aus.
Host	Niedrig	Der Host ist das Ziel des Managements und gilt in diesem Umfeld nicht als feindlich. Eine Ausnahme sind Systeme, in die eingebrochen wurde. Genau dies zu verhindern ist eine der Aufgaben des Netz- und Systemmanagements.
Migrationspfad	Hoch	Große verteilte Netze erstrecken sich häufig über unsichere öffentliche Netze, so dass von dort durchaus mit einer Gefahr zu rechnen ist.
Netzwerk	Hoch	Die Agentenplattformen werden in vielen Fällen auch von nicht vertrauenswürdigen Quellen (oder von beliebiger Stelle) aus erreichbar sein. Sie sind daher allen Gefahren des Internets ausgesetzt.

Tabelle 3.1 *Relevanz möglicher Angreifer*

3.5 Schutzmechanismen

Vor der Realisierung der erforderlichen Schutzmechanismen sind eine Reihe von Überlegungen nötig. Es stellt sich nicht nur die Frage, wie die Barrieren ihren Schutz realisieren, sondern auch wie sie konfiguriert werden und anhand welcher Informationen sie ihre Entscheidungen treffen sollen. Bei jeder Aktion mit Wirkung über die Grenzen eines Agenten hinaus (Methodenaufruf, Zugriff auf Ressourcen und ähnliches) ist zu entscheiden, ob diese Aktion zulässig ist oder nicht und bei Bedarf muss entsprechend reagiert werden.

3.5.1 Hintergrundfragen

Die folgende Liste stellt exemplarisch einige Hintergrundfragen zusammen, die im Rahmen dieser Arbeit beim Design der MobMan-Sicherheitsarchitektur (Mobile Manager) zu beantworten waren. In der Liste zeigt sich auch, wie viele Freiheitsgrade in einem System dieser Art vorhanden sind.

a) Welche Aktionen sind zu überwachen?

- Zugriff auf andere Objekte (Agenten, Plattformen, Dateien, spezielle Methoden).
- Kommunikation (mit anderen Agenten und mit anderen Diensten).
- Migration.
- Verbrauch von Ressourcen.

b) Entscheidung abhängig von welchen Kriterien?

- Identität des Benutzers, des Agenten und des Autors.
- Bisher besuchte Plattformen [Chan 99].
- Bisherige Aktionen [Edjlali 98].
- Rechte (Zugriffsschutz/Capabilities) [Nagaratnam 98], [Wallach 97].
- Reputation/Bewertung des Agenten.

c) Wer trifft die Entscheidung?

- Administrator der Plattform.
- Absender des Agenten.
- Autor des Agenten.
- Dritte (Zentrale oder eine Art TÜV).

d) Wer setzt die Entscheidung durch?

- Betriebssystem des Hosts.
- Plattform selbst.
- Security Manager.
- Einzelne Agenten, für sich selbst oder stellvertretend für andere.

e) Wann wird die Entscheidung gefällt und durchgesetzt?

- Zur Laufzeit (etwa Security Manager).
- Beim Starten (etwa Code Verifyer).

3.5.2 Konfigurationssprache

Viele Architekturen versuchen, die Schutzmechanismen in der Plattform vorzugeben und über eine eigene Sprache zu konfigurieren [Karjoth 97]. Um einigermaßen umfassend sein zu können, ist deren Komplexität aber bereits nahe an einer vollständigen Programmiersprache. Die Menge an Freiheitsgraden ist hoch durch die Flexibilität mobiler Agenten und deren Plattformen sowie durch die Anzahl der beteiligten Instanzen.

Interessante Ansätze sind beispielsweise auch Trust Management und Proof Carrying Code (PCC, [Necula 98]). Letzterer enthält Hinweise im Code, wie seine Korrektheit mathematisch nachzuweisen ist. Die Korrektheit wird im Kontext von Trust Management gegen Metadaten geprüft, die die Rechte eines Agenten beschreiben. Diese Metadaten (zusammen mit einem formalen Beweis) müssen einer formalen Security Policy entsprechen, über die eine Plattform verfügt [Feigenbaum 97], [Fong 98]. Eine weitere Variante arbeitet mit einer erweiterten Interface Definition Language (IDL), die verschiedene Views (Sichten) auf Agenten erzeugt und sicherstellt, dass nur Methoden aus einem bestimmten View benutzbar sind [Hagimont 97].

Diese Ansätze benötigen aber einen beträchtlichen Aufwand, der bei neuen Agenten und bei jeder Änderung eines Agenten von neuem anfällt.

3.5.3 Selbstschutz

MobMan verfolgt daher einen anderen Ansatz: Neben einem Grundschutz, den die Plattform und der Security Manager anbieten, ist jeder Agent selbst für seinen Schutz und den Schutz der von ihm bereitgestellten Ressourcen zuständig. Dies gilt im besonderen Maße für privilegierte Systemagenten, die die Fähigkeiten der Plattform erweitern. Eine Konfiguration ist dennoch möglich: Jeder Agent erhält einen Bereich der Konfigurationsdatei, in dem unter anderem die jeweiligen Sicherheitsparameter konfigurierbar sind.

Die Plattform muss allerdings dafür sorgen, dass der Agent diesen Schutz auch wirksam durchführen kann. Sie muss etwa den Absender einer Nachricht authentifizieren und die Agenten sicher voneinander trennen (keine Referenzen). Wenn ein Agent eine Referenz auf sich oder auf eines seiner Objekte weitergeben will, so soll er das auch können. Die Sicherheit und Isolation der Agenten soll nicht deren gewollte Kooperation beeinträchtigen.

Die Idee des Selbstschutzes wurde von einigen Autoren kritisiert [Vitek 97b]: Sie führe dazu, dass jeder Agentenautor selbst Sicherheit in seine Agenten programmieren müsse, Sicherheitsprüfungen über den ganzen Code verstreut und Änderungen an den Sicherheitseinstellungen nahezu unmöglich seien. Dem ist entgegenzuhalten, dass es bei mobilen Agenten keine einheitliche Autorität gibt, die zentral an einer Stelle die Sicherheitsparameter festlegen könnte. Agenten sind per Definition autonom und müssen daher auch in der Lage sein, Entscheidungen über ihre eigene Sicherheit zu treffen.

Wichtig ist allerdings, dass die Default-Fälle sicher sind und die Architektur bereits sichere Kommunikationsmechanismen vorgibt, so dass eine potenzielle Sicherheitslücke nur gewollt eingefügt werden kann. In diesem Fall muss der Autor für den passenden Schutz sorgen. Wo möglich und sinnvoll, sollen die Sicherheitsparameter auch konfigurierbar sein. Die MobMan-Architektur geht speziell auf diese widersprüchlichen Anforderungen ein.

3.6 Erforderliche Barrieren

Aus der Tabelle der relevanten potenziellen Angreifer wurde abgeleitet, dass als Angreifer im Anwendungsfall Netzmanagement vor allem der Agent und sein Autor sowie der Benutzer in Frage kommen, zudem sind die Gefahren durch die Vernetzung zu beachten. Die zu schützenden Instanzen sind vor allem Agent, Plattform und Host sowie das Netzwerk und der Migrationspfad. Autor, Benutzer und Administrator können nicht direkt angegriffen werden, sie sind aber indirekt Opfer durch den Schaden, den ihre Agenten und Plattformen erleiden.

Unter den gegebenen Rahmenbedingungen ist die Plattform als vertrauenswürdig anzusehen: Sie ist Teil der Management-Infrastruktur und a priori bekannt. Daraus ergibt sich auch, dass die Benutzer auf allen Plattformen bekannt sind, auf die ihre Agenten migrieren sollen.

Im Vorgriff auf die folgenden Kapitel sind einige Punkte an dieser Stelle noch erwähnenswert. Die Sicherheitsmechanismen in MobMan sind konfigurierbar und vollständig abschaltbar: Bei einem Verzicht auf den Security Agent ist die Arbeit mit MobMan zwar einfacher, aber ohne die meisten Schutzmechansimen riskant. Die Architektur unterscheidet zwei Arten von Agenten: Benutzeragenten und Systemagenten, eine weitere Unterscheidung erfolgt in privilegierte und normale Systemagenten. Da die meisten Funktionen als Systemagenten realisiert sind, kann durch den Austausch dieser Komponenten das System an die unterschiedlichsten Anforderungen angepasst werden.

a) Schutz der Agenten

- Jeder Agent muss mit seinen eigenen Klassen laufen, namensgleiche Klassen anderer Agenten dürfen seine Funktion nicht beeinträchtigen oder verändern. Siehe 5.3 *Trennung der Klassen*, Seite 78.
- Agenten erhalten keine Referenzen auf andere Agenten (siehe 5.4 *Trennung der Agenten: keine Referenzen*, Seite 81). Ausnahmen sind privilegierte Systemagenten, die die Referenz etwa zur Migration benötigen. Agenten können ihre Referenzen allerdings freiwillig weitergeben. Der Zugriffsschutz der Programmiersprache ist als Schutz nicht ausreichend, da er sich auf Klassen und nicht auf einzelne Objekte bezieht.
- Um dennoch auf Informationen anderer Agenten zugreifen zu können, bietet die Plattform einen namensbasierten Agenten-Suchdienst an. Dieser gibt die IDs der Agenten zurück, die einen bestimmten Namen tragen. Die IDs können neben der eindeutigen Identifizierung eines Agenten auch als Proxy dienen, um die Properties dieses Agenten abzufragen. Siehe 5.5 *Agenten und ihre IDs/Agent Properties*, Seite 82.
- Agenten können sich authentifizieren. Die Mechanismen dafür sind austauschbar, derzeit sind User/Passwort und ein zertifikatsbasiertes Challenge-Response-Verfahren implementiert (siehe 6.2 *Authentifizierung*, Seite 95). Der Security Agent verwaltet die Zuordnung der ID zum authentifizierten Benutzer, nicht authentifizierte Agenten bleiben anonym. Jeder Agent kann über den Message-Mechanismus sicher erfahren, welcher Agent ihm eine Nachricht gesendet hat (ID) und als welcher User sich der Absender authentifiziert hat (Name).
- Somit ist eine eigene Rechteverwaltung einzelner Agenten möglich. Beispielsweise könnte ein Dateisystem-Agent eigene Zugriffsrechte implementieren. Den authentischen Namen des Agenten, der einen Zugriff wünscht, liefert der Security Agent. Die ACL würde hierbei der Dateisystem-Agent selbst führen.
- Zusätzlich existiert eine plattformzentrale Rechtezuweisung (6.3 *Rechte*, Seite 105), die als Liste von Erlaubnissen implementiert ist. Jeder Agent kann diese Liste abfragen.

- Während der Übertragung schützt das Sicherheitsprotokoll SSL/TLS die Agenten vor möglichen Angreifern (6.4 *Gesicherte Kommunikation mit SSL/TLS*, Seite 106).

b) Schutz der Plattform

- Einige Mechanismen, die Agenten schützen, sichern auch die Plattform vor Angriffen. Insbesondere ist hier die Trennung der Klassen zu nennen (5.3 *Trennung der Klassen*, Seite 78).
- Die Authentifizierung der Kommunikationspartner (6.2 *Authentifizierung*, Seite 95) schützt vor Angreifern, die keine Benutzerkennung für die Plattform besitzen.
- Nur Agenten, deren Implementierung nicht gegen die Grundregeln verstößt, werden überhaupt auf einer Plattform zugelassen. Diese Überprüfung findet auf allen Wegen statt, auf denen ein Agent eine Plattform betreten kann: Bei der Migration, beim Start der Plattform oder als Kind eines anderen Agenten. Siehe 6.1 *Überprüfung neuer Agenten*, Seite 93.

c) Weitere

- Der Security Manager sorgt für einen Grundschutz des Hosts sowie des Netzwerks (5.7 *Security Manager*, Seite 90). Er wird vom Security Agent installiert.

Um diese Barrieren ins Gesamtsystem einordnen zu können, ist eine Beschreibung der Architektur des Agentensystems notwendig.

Kapitel 4

Architektur

Eine Sicherheitsarchitektur ist niemals unabhängig von dem System, das sie schützen soll. Sie ist vielmehr als integraler Bestandteil zu sehen, daher erfolgt an dieser Stelle ein Überblick über die Architektur von MobMan. Dieses Kapitel betont die Teile, die aus dem Blickwinkel der Sicherheit besonders relevant sind. Das trifft vor allem auf die Kommunikationsmechanismen zu, an die spezielle Anforderungen zu stellen sind. Dabei treten einige Java-spezifische Fragen auf (siehe Kapitel 5). Unabhängig betrachtbare Aspekte der Sicherheit finden sich in Kapitel 6.

Ausgangspunkt der Entwicklung war ein Prototyp von MobMan, der im Rahmen der WILMA-Arbeitsgruppe am Lehrstuhl für Datenverarbeitung entstanden war [Henne 98], [Trommer 99]. Schon diese Version hatte das Netzmanagement als Anwendungsziel, speziell das Konfigurations- und Leistungsmanagement [Proell 98], [Glissmann 99], [Heldwein 98]. Spezielle Sicherheitsfunktionen waren jedoch nicht vorhanden: MobMan sollte vor allem die Machbarkeit nachweisen und realistische Daten über die Netzbelastung liefern [Reitzig 99].

Da sich Sicherheit nicht einfach nachrüsten lässt, entstanden zentrale Teile der MobMan-Architektur neu oder wurden geeignet angepasst. Ursprünglich konnten die Agenten Referenzen auf alle anderen Agenten erhalten. Die vollständige Trennung der Agenten und ihrer Klassen hatte weit reichende Folgen, von der Migration und der Kommunikation bis hin zur Einführung der Agenten-ID. Die Erweiterung auf Agenten-Properties erlaubt im Ergebnis eine sehr einfache Kommunikation. Auch die aufwändigere Kommunikation über Messages wurde stark modifiziert, um die Identifizierung und Authentifizierung sicherzustellen.

Hervorzuheben ist das Plugin-Konzept. Motivation hierfür waren die Authentifizierungsmechanismen, die Implementierung ist aber allgemein gehalten und daher universell einsetzbar, etwa für die Wegesuche. Die Authentifizierungsmechanismen sind als Client-Server-Architektur sowohl plattformintern, als auch über eine Netzverbindung einsetzbar. Sie wurden in alle Protokolle eingebunden. Bei dieser Arbeit wurde die MobMan-Protokollfamilie weitgehend neu entworfen. Die Einbindung von SSL/TLS führte zu weiteren Änderungen, beispielsweise entstanden angepasste Trust Decider und eine Mini-CA (Certification Authority).

Um die umfangreichen Sicherheitsfunktionen, die Passwortdateien und den Security Manager mit Parametern zu versorgen, wurde auch der Konfigurationsmechanismus überarbeitet. Das alte Ad-hoc-System benutzte zeilenbasierte Textdateien, das neue System ist hierarchisch strukturiert. Diese Änderung hat den größten Einfluss auf den Portierungsaufwand alter Agenten. Die Sicherheitsarchitektur selbst hat nur einen geringen Einfluss auf die Agenten, so lange sie sich gutartig verhalten.

4.1 Designkriterien

Ein komplexes System wird in der Regel nicht entwickelt, ohne vorher die Anforderungen zu klären. Aus dem konkreten Einsatzfeld ergeben sich eine Reihe von Designkriterien, die dann geeignet umzusetzen sind. [Susilo 98] nennt sechs Grundanforderungen an ein Mobile-Agenten-System, das für Managementaufgaben genutzt werden soll:

- *Unterstützung für Legacy-Systeme* (etwa SNMP)
  Da in der Praxis enorme Investitionen in bestehenden Systemen stecken, lassen sich diese nicht einfach durch eine Alternative ablösen.
- *Portabilität*
  Heutige Netze sind sehr heterogen. Neben den Host-Systemen sind auch reine Netzinfrastrukturkomponenten wie Router oder Switches in Betracht zu ziehen. Ein umfassendes Managementsystem muss dieser Heterogenität durch eine weitreichende Portabilität gerecht werden.
- *Persistenter Zustand*
  Die auf den Plattformen gewonnenen Daten müssen die Agenten auch über die Migration hinaus behalten.
- *Sicherheit*
- *Zugang zu den Host-Ressourcen*
  Kann in einer heterogenen Umgebung technisch schwierig sein, ist aber Grundvoraussetzung für das Systemmanagement.
- *Kommunikation* (zwischen Agenten)
  Agenten sollen kooperieren und gemeinsam Aufgaben lösen können.

Für MobMan wurden diese Grundanforderungen durch eine Reihe von Designkriterien ergänzt, die sich aus dem Einsatzfeld Netzmanagement ergeben:

- *Wenig Overhead bei der Übertragung*
  - Minimierung der Belastung des Netzes durch das Management selbst.
  - Im Krisenfall steht eventuell kaum noch Bandbreite zur Verfügung, daher muss das Management schonend mit ihr umgehen.
- *Wenig Abhängigkeiten, vor allem keine zentralen Dienste*
  - Einfacheres Deployment (Installation).
  - Zentrale Dienste sind im Zweifelsfall nicht erreichbar, genau dann ist das Netzmanagement aber besonders gefordert.
- *Hohe Flexibilität zur Anpassung an lokale Gegebenheiten*
  - Modularer Aufbau, in dem auch zentrale Komponenten austauschbar sind.
  - Verschiedene Modelle für verschiedene Szenarien.
  - Auch die Sicherheitsdienste sind austauschbar und nicht zwangsweise vorhanden.
  - Konfigurierbar.

- *Plattformunabhängig*
  - Sollte auf möglichst vielen Plattformen laufen, wobei hier die Architektur des Hosts sowie das Betriebsystem gemeint sind.
  - Keine Modifikation an Standardkomponenten (zum Beispiel Interpreter und Laufzeitumgebung).
- *Offen*
  - Zugang zu SNMP-MIBs.
  - Zugang zu Datenbanken.

Für die Sicherheit des Systems ist folgendes Szenario relevant:

- Nutzung für Netzmanagement.
- Alle Plattformen befinden sich in einer Hand. Sie liegen in einer administrativen Domäne, die aber in Unterdomänen aufgegliedert sein kann.
- Plattformen sind vertrauenswürdig.
- Absender sind auf allen Plattformen bekannt.

Eine Interoperabilität mit anderen Agentensystemen ist weniger wichtig; diese Forderung ist primär bei Systemen relevant, die auf E-Commerce ausgerichtet sind.

4.2 Sprache

Eng mit der Architektur verbunden ist die Wahl einer geeigneten Sprache. Grundsätzlich eignen sich nach [Cugola 97a], [Cugola 97b] vor allem folgende Sprachen für mobilen Code:

- Java
- Telescript
- Obliq
- SafeTcl [Ousterhout 98]
- AgentTcl
- TACOMA
- M0
- Tycoon
- Facile

Ein Vergleich einiger dieser Sprachen findet sich unter anderem in [Gritzalis 98b], [Moore 98] und [Thorn 97]. Auch wenn keine Sprache allein für die Sicherheit eines Systems sorgen kann [Volpano 98], weist Java doch eine Reihe von Vorteilen auf, die sie für MobMan geeignet erscheinen lassen (siehe 4.1 *Designkriterien*, Seite 38), beispielsweise:

- Plattformunabhängig (Host-Architektur und Betriebsystem).
- Auf vielen Plattformen verfügbar.
- Viele Bibliotheken vorhanden.
  - SNMP-Protokoll
  - Datenbankzugriff
  - Krypto-Bibliotheken
- Diese Bibliotheken sind auch in Fassungen verfügbar, die selbst in Java geschrieben sind und damit als Teil eines Agenten die Mobilität nicht behindern.

- Keine Modifikationen der JVM nötig, Beschränkung auf JDK 1.1 möglich (ebenfalls ohne Modifikation), um maximale Anzahl an Plattformen zu erreichen.
- Sicherheitsmechanismen direkt in der Sprache [Dean 97].
- Aus Sicherheitsüberlegungen wäre SafeTcl [Gritzalis 98b] mindestens ebenso gut geeignet. Auch Tcl ist plattformunabhängig und steht auf vielen Host-Architekturen und Betriebssystemen bereit. Alle relevanten Bibliotheken sind auch für diese Sprache verfügbar. Die Vorteile von Java gegenüber Tcl sind vor allem in der größeren Verbreitung zu sehen.
- Im Prinzip lassen sich auch mit nativem Code und speziellen Sandboxes sichere Ausführungsumgebungen erstellen [Kato 97]. Bei MobMan ist die Plattformunabhängigkeit aber gerade eine der Hauptanforderungen.

Allerdings hat Java auch einige Einschränkungen (siehe 5.2 *Beschränkungen von Java*, Seite 77), die es durch eine geeignete Architektur zu umgehen gilt.

4.3 Klassenmodell

Passend zu diesen Randbedingungen ist das MobMan-System entstanden. Die Plattform (siehe 4.4 *Plattform*, Seite 42) besteht selbst nur aus einem schlanken Kern, der durch eine Reihe von Systemagenten erst die volle Funktionalität erhält. Der `MigrationAgent` ermöglicht die Migration von Agenten, der `CommunicationAgent` ihre Kommunikation und je ein `NeighbourAgent` verweist auf eine andere Plattform (siehe Abbildung 4.1 *Vernetzte MobMan-Plattformen*, Seite 40) und realisiert so eine virtuelle Vernetzung der Plattformen.

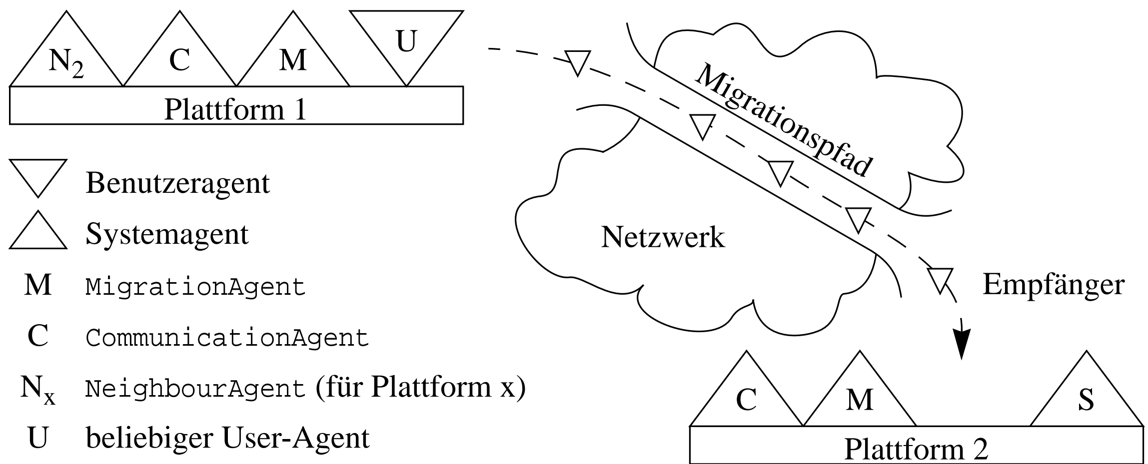

Abbildung 4.1 *Vernetzte MobMan-Plattformen*

Den Zusammenhang der wichtigsten Klassen in MobMan zeigt Abbildung 4.2 *Klassenmodell des MobMan-Systems – Übersicht*, Seite 41 in Form eines UML-Diagramms (Unified Modeling Language, [OMG 01]). Die einzelnen Klassen werden in den folgenden Kapiteln näher erläutert. Zur UML-Darstellung siehe auch Anhang H *UML-Diagramme*, Seite 168.

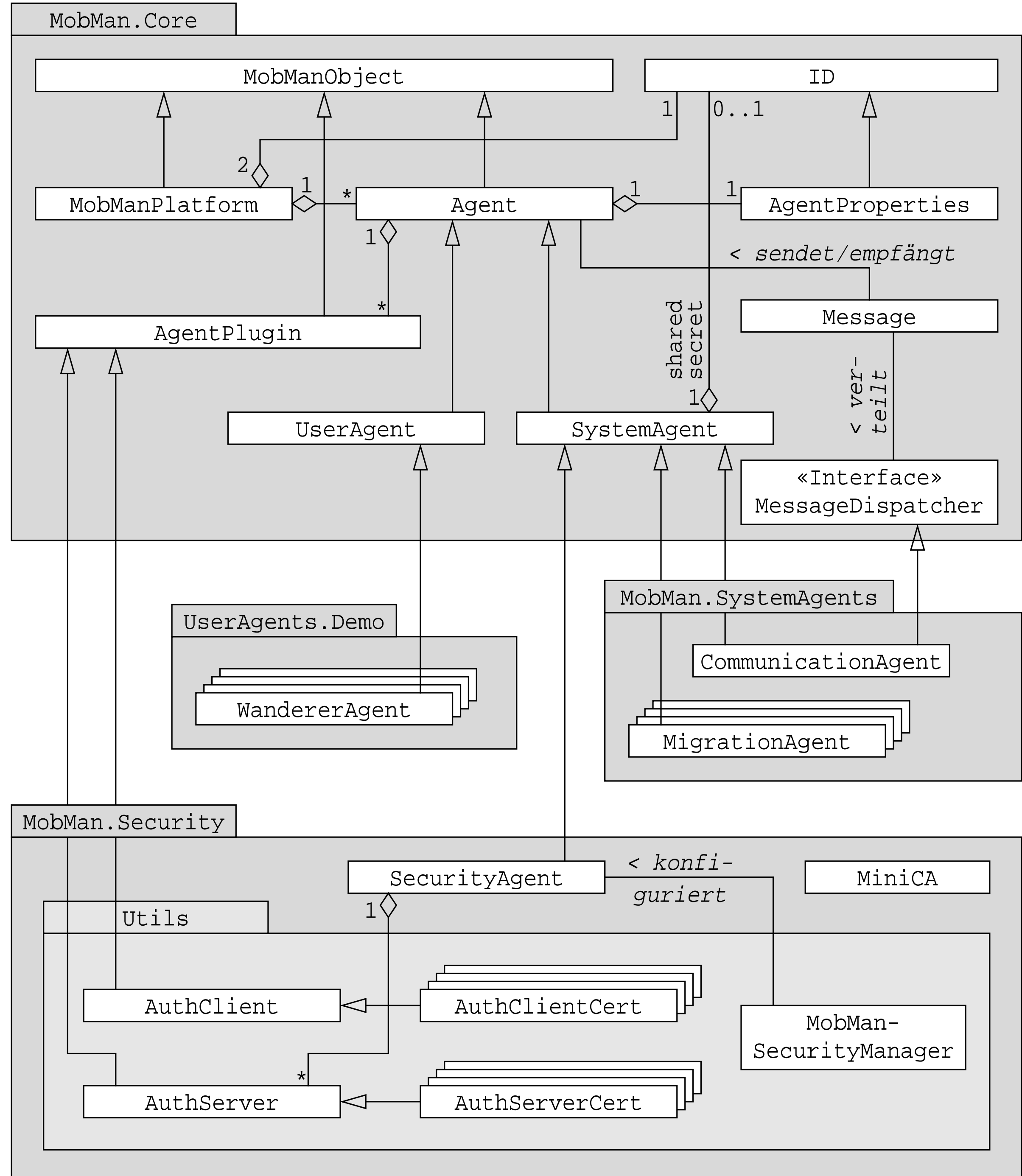

Abbildung 4.2 *Klassenmodell des MobMan-Systems – Übersicht*

Die Abbildung enthält nur die wichtigsten Klassen sowie eine Auswahl an Systemagenten, um die Übersicht nicht durch zu viele Elemente zu beeinträchtigen. Die Zugehörigkeit zu den einzelnen Java-Packages hat nicht nur organisatorische Bedeutung: Klassen aus `MobMan.Core` dürfen nicht als Teil eines Agenten auf die Plattform migrieren (siehe 6.1 *Überprüfung neuer Agenten*, Seite 93). Dadurch können sich diese Klassen gegenseitig vertrauen und mithilfe von Package Visibility kooperieren.

4.4 Plattform

Abbildung 4.3 *Klassendiagramm MobManPlatform*, Seite 42 zeigt das Klassendiagramm der `MobManPlatform`-Klasse. Bei den Zugriffsrechten bedeutet „+“ `public`, „#“ `protected`, „~“ steht für Package Visibility und „-“ für `private`. „*“ ist eine Ergänzungen gegenüber UML 1.4, es kennzeichnet eine `final`-Implementierung, die in den Kind-Klassen nicht überschreibbar ist.

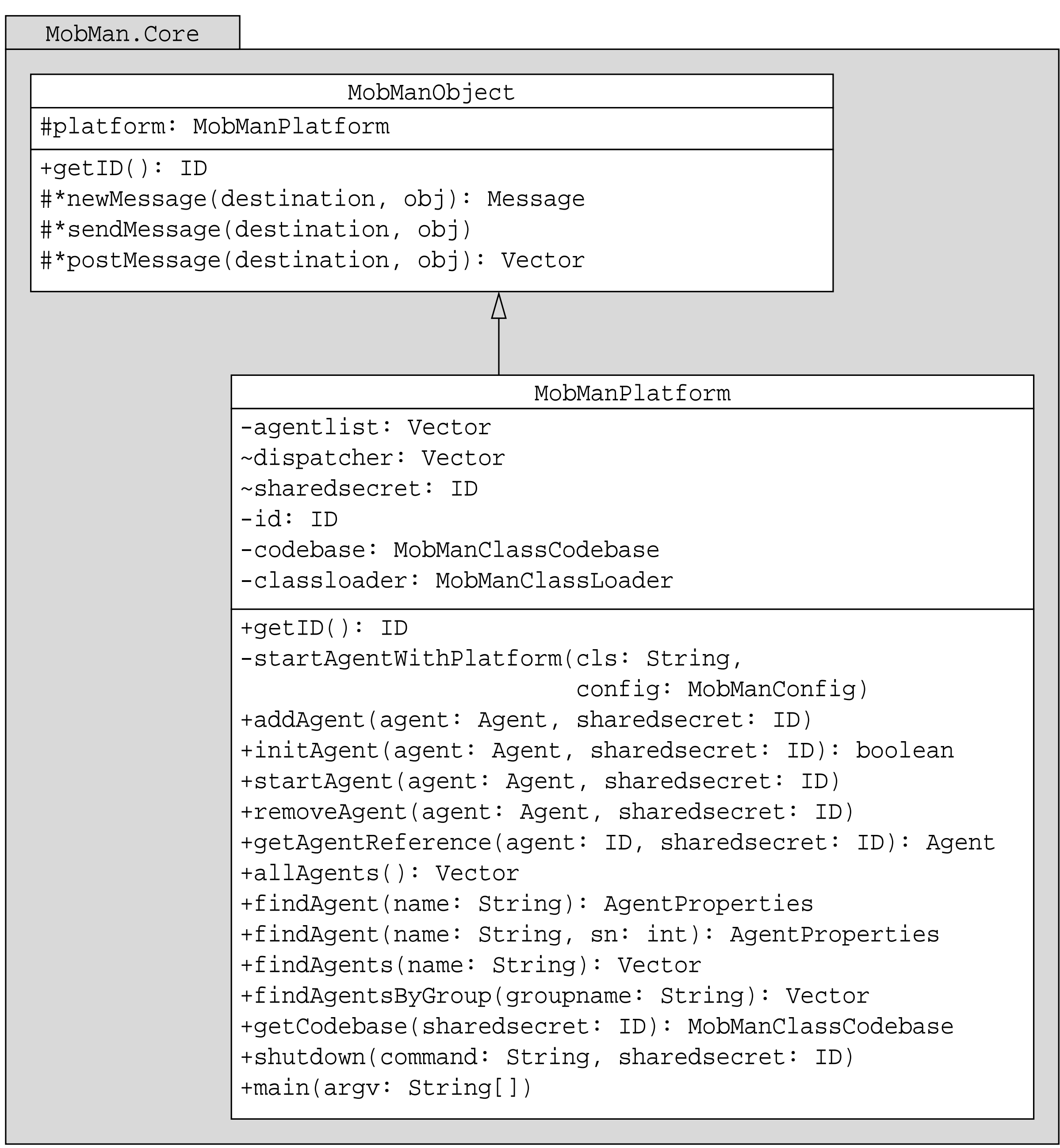

Abbildung 4.3 *Klassendiagramm MobManPlatform*

a) Überblick

Die Agentenplattform stellt die Basis des Systems dar, sie führt die Agenten aus. Bei MobMan ist die Plattform selbst nur als schlanker Kern (Klasse `MobManPlatform`) konzipiert, dessen Basisdienste durch Systemagenten ergänzt werden. Das Agentenparadigma dient hier also auch als Modularisierungskonzept. Sogar für grundlegende Dienste wie Migration und Kommunikationsaufgaben sind eigene Agenten zuständig. Für diese Aufgaben müssen einige Agenten zusammen mit der Plattform gestartet werden. Grund ist das Bootstrapping: Ohne Migration Agent kann kein Agent auf die Plattform migrieren.

Dieses Konzept bietet eine sehr weitgehende Flexibilität. Durch die Modularisierung in Form von Agenten sind die Schnittstellen dieser Systemagenten zudem einfach und standardisiert, ohne eigene APIs (Application Programming Interface) einführen zu müssen.

b) Dienste

Um dies zu ermöglichen, stellt die Plattform einige Basisdienste bereit. Die zentrale Klasse ist `MobMan.Core.MobManPlatform`, die als Container für Agenten dient. Sie verwaltet die Agenten und speichert ihren Code in einer Codebase. Beim Start des Systems initialisiert sie ihre Datenstrukturen, liest die Konfiguration ein (siehe Anhang A *Konfigurationsdateien von MobMan*, Seite 141) und startet die darin angegebenen Agenten.

Die Plattform bietet einige ihrer Dienste nur privilegierten Systemagenten an. Dabei handelt es sich um Systemagenten, die gleichzeitig mit der Plattform gestartet wurden. In dieser Art des Starts drückt sich das Vertrauen des Administrators in die Sicherheit eines Agenten aus. Die Plattform gibt diesen Agenten als Shared Secret eine unfälschbare Referenz (Klasse `ID`) mit, durch die sich die Agenten später ausweisen können. Privilegierte Agenten können mithilfe der Plattform etwa Referenzen auf andere Agenten erhalten (`getAgentReference`), Agenten initialisieren, sie zur Plattform hinzufügen und starten. Sie dürfen Agenten auch beenden.

Systemagenten können auch im späteren Verlauf auf die Plattform migrieren, ein Beispiel wäre der Neighbour Agent. Dieser Agent verweist auf andere Plattformen und ist damit beim Routing behilflich. Da auch nicht privilegierte Systemagenten (durch Vererbung und Paketzugehörigkeit) mehr Rechte als normale Benutzeragenten haben, sollte nicht jeder Benutzer Systemagenten starten dürfen. Bei MobMan sorgt der Migration Agent dafür, dass nur Benutzer mit diesem Recht auch Systemagenten starten können.

Die bereits angesprochene Codebase ist der zentrale Agenten-Code-Cache einer Plattform. Durch diesen Mechanismus muss neuer Code nur einmal zu einer Plattform übermittelt werden, womit die Menge des zu übertragenden Codes minimiert wird. Privilegierte Systemagenten haben Zugriff auf diesen Cache, so dass der Migration Agent den Mechanismus nutzen kann.

Alle Agenten können mithilfe der Plattform die IDs und Properties (siehe 5.5 *Agenten und ihre IDs/Agent Properties*, Seite 82) anderer Agenten abfragen. Dies ermöglicht eine einfache lokale Kommunikation innerhalb der Plattform. Für fortschrittlichere Mechanismen und für die Kommunikation zwischen Plattformen dient ein eigener Systemagent, der Communication Agent.

c) Benutzung

Jeder Agent erhält bei seiner Initialisierung eine Referenz auf das Plattform-Objekt (gespeichert im Feld `platform`). Damit kann er die Dienste der Plattform nutzen (siehe Abbildung 4.3 *Klassendiagramm MobManPlatform*, Seite 42). Referenzen auf andere Agenten erhält er aus Sicherheitsgründen nicht, nur Referenzen auf das Properties-Objekt anderer Agenten:

```
AgentProperties agentproperties=platform.findAgent("AgentXyz");
```

Sind mehrere Agenten mit identischem Namen auf der Plattform vorhanden, dann gibt die `findAgent()`-Methode nur die Properties des ersten Agenten zurück. Folgende Methoden liefern Vektoren mit allen passenden Agenten:

```
Vector list=platform.findAgents("AgentXyz");
Vector list=platform.findAgentsByGroup("Neighbours");
Vector list=platform.allAgents();
```

Über geschützte Methoden, die nur mit Shared Secret benutzbar sind (also nur für privilegierte Systemagenten zugänglich), liefert die Plattform auch Referenzen auf Agenten. Einige Methoden dienen der Verwaltung von Agenten und der Plattform selbst.

```
Agent agent=platform.getAgentReference(agentproperties, sharedsecret);
platform.addAgent(agent, sharedsecret); // Agent auf Plattform registrieren
platform.initAgent(agent, sharedsecret); // Neuen Agent initialisieren
platform.startAgent(agent, sharedsecret); // Agent starten
platform.removeAgent(agent, sharedsecret); // Agent entfernen
platform.getCodebase(sharedsecret); // Referenz auf Codebase
platform.shutdown("SHUTDOWN", sharedsecret); // Plattform beenden
platform.shutdown("REBOOT", sharedsecret); // Plattform neu starten
```

d) Sicherheitsüberlegungen

- Jeder Agent erhält eine Referenz auf die Plattform, deren Methoden müssen also selbst für die passende Zugriffskontrolle sorgen.
- Nur bestimmte Systemagenten sollen Zugriff auf eine Reihe von Methoden erhalten. Diese Entscheidung soll aber nicht anhand der Klasse oder des Packages erfolgen, sondern abhängig davon, ob sie vom Administrator zusammen mit der Plattform gestartet wurden.
- Lösung mithilfe eines gemeinsamen Geheimnisses. Die beteiligten Agenten müssen sich dazu gegenseitig vertrauen, kein Agent darf die Referenz auf das Shared Secret an nicht vertrauenswürdige Dritte weitergeben.

4.5 Agenten und ihre Komponenten

Dieses Kapitel stellt die Agenten aus Benutzersicht vor. Zu einigen Details sind an späteren Stellen Hintergründe dargestellt sowie alternative Lösungsmöglichkeiten. Für das Verständnis erscheint es aber sinnvoll, zunächst einen Überblick zu geben.

```
MobMan.Core

MobManObject
#platform: MobManPlatform
+getID(): ID
#*newMessage(destination, obj): Message
#*sendMessage(destination, obj)
#*postMessage(destination, obj): Vector

Agent
-propID: AgentProperties
-*name: String
-*groupname: String
-*version: double
#description: String
#owner: String
#replaceAllowed: boolean
#multipleAllowed: boolean
#properties: Vector
#results: Vector
#requiredClasses: Vector
#plugins: Hashtable
+*getName(): String
+*getGroupName(): String
+*getVersion(): double
+*getID(): ID
+*setNewID():
#*setPlatform(platform: MobManPlatform)
+init(): boolean
+receiveMessage(msg: Message): Object
+getProperty(property: String): Object
#*addResult(obj: Object)
+*addPlugin(name: String, plugin: AgentPlugin)
#*callPlugin(name: String, command: String,
             parameter: Object): Object
#*addRequiredClasses(cls: String)
+agentArrived(agent: AgentProperties)
+agentLeft(agent: AgentProperties)
#requestBirth(classname: String,
              config: MobManConfig)
#*requestCopy(hostaddress: String,
              port: int): boolean
#*requestKill()
#*requestMove(String hostaddress, int port)
```

Abbildung 4.4 *Klassendiagramm Agent*

4.5.1 Agenten

a) Überblick

Wie bereits erwähnt, unterscheidet MobMan zwischen Systemagenten und normalen Benutzeragenten. Bei beiden ist zusätzlich zwischen aktiven und passiven Agenten zu trennen. Ein passiver Agent agiert nicht aus eigenem Antrieb, er stellt nur Dienste und Methoden bereit, die andere Agenten nutzen können. Passive Agenten verhalten sich damit als Server. Bei Systemagenten gibt es aus Sicht der Plattform noch die Einteilung in privilegierte und nicht privilegierte Systemagenten. Diese unterscheiden sich lediglich dadurch, dass sie die Referenz auf eine bestimmte ID kennen oder eben nicht.

b) Praxis

Ein einfacher Benutzeragent besteht minimal aus folgendem Code. Die kursiv gedruckten Teile sind als Platzhalter gedacht, die durch konkrete Werte zu ersetzen sind.

```
package UserAgents.MyPackage;

import MobMan.Core.*;           // UserAgent
import MobMan.Core.Utils.*;     // MobManConfig

public class MyAgent extends UserAgent
{
  public MyAgent(MobManConfig config)
  {
    super("MyAgent", 1.0, "MyGroup");
  }
}
```

Dieser Agent ist passiv, er wird nicht von sich aus aktiv, sondern lediglich auf Anfragen reagieren. Er hat keinen eigenen Ausführungs-Thread. Um selbständig zu handeln, also ein aktiver Agent zu sein, muss er das Interface `Runnable` und eine Methode mit Namen `run()` implementieren. Fett gedruckt sind die Teile, die sich von einem passiven Agenten unterscheiden:

```
public class MyAgent extends UserAgent implements Runnable
{
  ...
  public void run() { ... }
}
```

Von der Basisklasse `Agent` erbt der Agent *`MyAgent`* (über die Zwischenstufe `UserAgent`) eine Reihe von Eigenschaften:

- `name`, `groupname` und `version`
  Diese Variablen sind private Elemente der `Agent`-Klasse. Der Konstruktor dieser Basisklasse belegt sie, der Agent selbst kann sie zwar nicht verändern, aber mit den Methoden `getName()`, `getGroupName()` und `getVersion()` lesen. Die Werte sind auch über die Properties des Agenten verfügbar.
- `description`
  Enthält eine kurze Beschreibung des Agenten. Typischerweise wird diese Variable im Konstruktor gesetzt. Sie kann über die Properties gelesen werden.

- `owner`
  Rein informelle Angabe, wem dieser Agent gehört. Der Defaultwert ist `anonymous`. Nur Agenten, die zusammen mit der Plattform gestartet werden, haben keinen Eigentümer (und dieser Parameter ist ein leerer String). Der Owner ist auch über die Properties des Agenten verfügbar.
  Wichtig ist, dass es sich hier nur um eine informative Angabe handelt, und nicht um den authentifizierten Benutzer. Für sicherheitsrelevante Entscheidungen ist dieses Feld daher nicht geeignet.

Eine Reihe von Methoden der Agentenbasisklasse kann der Agent überschreiben. Diese Methoden werden etwa von der Plattform genutzt.

- `public MyAgent(MobManConfig config) // Konstruktor`
  Die Startkonfiguration erhält der Konstruktor als `MobManConfig`-Baum. Darin enthalten ist nur der Teil, der diesen Agenten betrifft. Der Konstruktor wird nur einmal bei der Erzeugung aufgerufen, nicht aber nach einer Migration.
- `public boolean init()`
  Diese Methode wird nach der Ankunft auf einer Plattform aufgerufen, bei aktiven und auch bei passiven Agenten. Die Plattformreferenz ist bereits gesetzt. Durch den Rückgabewert kann der Agent entscheiden, ob er er auf dieser Plattform ausgeführt werden will (True) oder nicht (False).
- `public void agentArrived(AgentProperties agent)`
  Ein neuer Agent ist auf der Plattform angekommen.
- `public void agentLeft(AgentProperties agent)`
  Ein Agent hat die Plattform verlassen.

c) Sicherheitsüberlegungen

- Das Prinzip des Selbstschutzes (siehe 3.5.3 *Selbstschutz*, Seite 33) wird durch die Architektur unterstützt.
- Viele Teilaspekte sind in den folgenden Kapiteln erläutert.

4.5.2 Plugins

a) Überblick

Manche Aufgaben müssen viele Agenten erfüllen, beispielsweise die korrekte Authentifizierung oder die Wahl des nächsten Migrationsziels. Ein naheliegender Mechanismus, um verschiedene Agenten mit diesen Fähigkeiten auszustatten, ist Vererbung. In vielen Fällen ist dies aber nicht zielführend, da der Autor des Agenten noch nicht weiß, welche Authentifizierungsmechanismen der Benutzer des Agenten verwenden wird. Als Lösung für diese Aufgabe bieten sich Plugins an. Im Anwendungsfall Wegewahl erfüllt ein einfaches Plugin den selben Zweck wie eine aufwändige Zwei-Agenten-Architektur, in der ein Agent die eigentliche Funktionalität trägt, während ein anderer die geeignete Route auswählt [Satoh 02].

Plugins erweitern den Funktionsumfang eines Agenten, ohne dessen Code zu ändern. Verschiedene Plugins können die selbe Aufgabe erfüllen. Der Absender des Agenten entscheidet, welche Plugins er seinem Agenten mit auf den Weg gibt und kann damit, ohne in den Code einzugreifen, die Logik des Agenten beeinflussen.

In MobMan sind beispielsweise verschiedene Authentifizierungsmechanismen implementiert. Ob sich ein Agent mit Name und Passwort oder mit einem Zertifikat und dem dazu passenden privaten Schlüssel authentifiziert, entscheidet der Absender. In beiden Fällen lesen die Plugins die Daten (Name und Passwort oder Zertifikat und Schlüssel) aus der Konfigurationsdatei, mit der der Absender seine Agenten parametrisiert.

b) Praxis

Plugins erben von der Klasse `AgentPlugin` und überschreiben folgende Methoden:

- `public MyPlugin(MobManConfig config) // Konstruktor`
  Der Konstruktor muss überschrieben werden. Er erhält die Konfigurationsdaten als Parameter.
- `protected void init()`
  Wird bei der Ankunft auf jeder neuen Plattform aufgerufen.
- `public Object execute(String command, Object parameter)`
  Diese Methode wird ausgeführt, wenn der Agent seine `callPlugin()`-Methode aufruft.

Ein Agent kann ein Kommando mit einem Parameter an das Plugin schicken. Das Kommando ist dabei eine Zeichenkette, der Parameter ist vom Typ `Object`. Wenn mehrere Parameter benötigt werden, kann es sich hierbei auch um einen Vektor handeln. Aus Sicht des Agenten sieht der Aufruf folgendermaßen aus:

```
callPlugin(name, command, parameter);
```

Ist ein Plugin mit passendem Namen verfügbar, wird bei ihm folgende Methode aufgerufen:

```
execute(command, parameter)
```

Plugins können beispielsweise durch die Konfiguration des Agenten eingefügt werden. Der Konfigurationsabschnitt für den Agenten muss nur einen entsprechenden Unterbaum enthalten:

```
plugins {
  Name {
    class = name.der.klasse
    Parameter
  }
}
```

Plugins können auch zur Laufzeit hinzugefügt werden: `agent.addPlugin("Name", plugin)`. Sie lassen sich aber nicht mehr entfernen, und es kann unter jedem Namen nur ein Plugin geben.

c) Vorhandene Plugins

- `MobMan.Utils.Plugins.WandererPlugin`
  Der Wanderer lässt den Agenten durch das Netz zu allen bekannten Zielen wandern. Der Agent migriert bei jedem Aufruf des Plugins mit `callPlugin("Wanderer", "next", null)` zu der Plattform, auf der er am längsten nicht mehr war. Die möglichen Zielplattformen kennt das Plugin durch die Neighbour-Agenten.
- `MobMan.Utils.Plugins.DistributePlugin`
  Kopiert den Agenten auf jeden bekannten Nachbarn der Plattform.

- `MobMan.Security.Utils.AuthClient...`
  Authentifizierungs-Client, mit dessen Hilfe sich ein Agent authentifizieren kann. Verschiedene Plugins sind für Benutzer-Passwort-basierte Authentifizierung oder für ein Challenge-Response-Verfahren mit Zertifikaten und RSA-Schlüsseln zuständig.

d) Authentifizierungs-Plugins

Ein Plugin wird zum Authentifizierungs-Client, wenn sein Name auf `Authentifier` gesetzt ist (siehe 6.2 *Authentifizierung*, Seite 95). Diese Plugins werden nicht nur vom Agenten selbst, sondern zusätzlich an folgenden Stellen aufgerufen:

- Von der Plattform, bevor sie die `init()`-Methode des Agenten aufruft (lokale Authentifizierung).
- Während der Migration, unmittelbar nach der Übertragung der Klassen, aber noch vor der Übertragung des Zustandes (Remote-Authentifizierung).
- Beim Senden einer Nachricht an andere Plattformen (Remote-Authentifizierung).

e) Sicherheitsüberlegungen

- Plugins gehören dem jeweiligen Agenten und sind von außen nicht direkt zugänglich.
- Die Authentifizierungs-Plugins stellen den zentralen Mechanismus zur Authentifizierung bereit. Durch die Implementierung als Plugin können beliebige Verfahren realisiert werden, die Architektur ist somit flexibel und erweiterbar.

4.5.3 Properties

a) Überblick

Zu den zentralen Sicherheitsfeatures in MobMan gehört die Trennung der Agenten voneinander (siehe 5.4 *Trennung der Agenten: keine Referenzen*, Seite 81). Um Agenten und andere Objekte in MobMan dennoch eindeutig zu kennzeichnen, wurde eine ID eingeführt. Bei Agenten kommt statt der Klasse `ID` eine Kindklasse zum Einsatz: `AgentProperties` (siehe 5.5 *Agenten und ihre IDs/Agent Properties*, Seite 82). Durch diese Klasse wird die ID zusätzlich zum Proxy, der den Zugriff auf einige Daten des Agenten erlaubt, ohne eine Referenz darauf preiszugeben.

b) Praxis

Jeder Agent kann eigene Properties bereitstellen, durch die andere Agenten Informationen erhalten. Einige Properties sind bereits in der Basisklasse implementiert: `NAME`, `VERSION`, `GROUP`, `DESCRIPTION`, `OWNER`, `REQUIREDCLASSES`, `MULTIPLEALLOWED`, `REPLACEALLOWED`, `RESULTS`, `AGENTID`. Eine Liste der vorhandenen Properties lässt sich durch Abfrage einer speziellen Property namens `PROPERTIES` ermitteln:

```
Vector v=(Vector)agentproperties.get("PROPERTIES");
```

Mit der `get()`-Methode lassen sich beliebige Properties abfragen:

```
Object o=agentproperties.get("property-name");
```

Will ein Agent weitere Properties bereitstellen, muss er die Liste der verfügbaren Properties ergänzen sowie die Methode `getProperty()` neu implementieren. Die `AgentProperties`-Klasse ruft diese Methode des Agenten auf.

```
public class MyAgent extends UserAgent
{
  int amount;  // Dieser Wert soll als Property abfragbar sein

  public MyAgent(MobManConfig config)
  {
    super("MyAgent", 1.0, "MyGroup");
    properties.addElement("AMOUNT");    // Neue Property bekannt geben
  }

  public Object getProperty(String property)
  {
    if (property.equals("AMOUNT"))      // Neue Property
      return new Integer(amount);       // Wert zurückgeben
    return super.getProperty(property); // Default-Properties
  }
}
```

c) Sicherheitsüberlegungen

- Properties stellen ein geschütztes Interface zu den Agenten dar. Der Agent kann auf diesem Weg beliebige Daten nach außen geben, per Default sind dies nur unkritische Informationen.
- Um eng zu kooperieren, kann ein Agent als Property (oder innerhalb der Kommunikation) auch eine Referenz auf sich selbst weitergeben.
- In der aktuellen Implementierung beantwortet der Agent selbst zur Laufzeit die Anfrage. Damit stellt die Properties-Abfrage eine Form von Kommunikation dar, bei der die Quelle der Abfrage nicht authentifiziert wird.
- Statt eine Antwort zu liefern, könnte der Agent zum Beispiel in eine Endlosschleife eintreten. Damit würde er einen Denial-of-Service-Angriff gegen den Anfrager durchführen: Da der Kontrollfluss nicht an den Anfrager zurückgeht, kommt dieser nicht mehr zur Ausführung. Eine Lösung dieses Problems wäre möglich, wenn der Agent nur Werte in einem Array (etwa als Hashtable realisiert) im Properties-Objekt setzen würde und die Abfrage gar nicht an den Agenten weitergereicht würde.
  - Bei schnell veränderlichen Werten würde dies zu vielen unnötigen Änderungen im Properties-Objekt führen, für die sich unter Umständen gar kein Agent interessiert.
  - Lösung durch Listener-Objekte möglich: Diese würden direkt als Wert-Objekt in die Hashtable eingetragen, könnten aber ebenfalls diesen Angriff durchführen.
  - Für reine Wert-Abfragen könnte man weiterhin Referenzen auf den Wert direkt in die Hashtable eintragen (Unterscheidung über die Vaterklasse).

4.5.4 Ergebnisse

Um einen einheitlichen Zugriff auf die Ergebnisse, die ein Agent gesammelt hat, zu ermöglichen, besitzen MobMan-Agenten einen Vektor `results`. Mit `addResults(obj)` hängt der Agent neue Ergebnisse an diesen Vektor an. Dabei kann er beliebige Objekte benutzen, sinnvoll sind vor allem Instanzen der Klasse `AgentResult`. Diese Klasse bietet eine Basis für standardisierte Ergebnisse.

Die `AgentResult`-Objekte speichern automatisch den Zeitpunkt, zu dem sie erzeugt wurden. Per Default enthalten sie eine Text-Beschreibung des Ergebnisses, durch Vererbung lässt sich das beliebig erweitern. Vorgesehen ist auch die Methode `show()`, mit der sich das Ergebnis als GUI-Komponente darstellen lässt.

Über die `AgentProperties` ist der Ergebnis-Vektor auch anderen Agenten zugänglich:

```
Vector v=(Vector)agentproperties.getProperty("RESULTS");
```

oder etwas einfacher in der Benutzung:

```
Vector v=agentproperties.getResults();
```

4.5.5 Wichtige Systemagenten

Die meisten Systemagenten sind im Package `MobMan.SystemAgents` enthalten. Es gibt allerdings einige Ausnahmen: Der Security Agent befindet sich im Paket `MobMan.Security` (siehe auch Abbildung 4.2 *Klassenmodell des MobMan-Systems – Übersicht*, Seite 41). Der Platform Interface Agent ist ebenfalls in einem eigenen Package angesiedelt: `MobMan.Pia`.

a) Communication Agent

Der Communication Agent ist für externe und interne Kommunikation zuständig. Er stellt das Server-Socket der Plattform bereit, über das alle Verbindungen laufen, also vor allem die Agentenkommunikation und die Migration. Eingehende Verbindungen verteilt er an die Agenten, die für die jeweiligen Protokolle zuständig sind (siehe 4.8 *Protokolle*, Seite 64). Interessierte Agenten können sich beim Communication Agent für ein Protokoll anmelden:

```
sendMessage("CommunicationAgent", "subscribe WAMP"); // Migration
sendMessage("CommunicationAgent", "subscribe GET");  // HTTP
```

Der Communication Agent liest den MAP-Header (MobMan Application Protocol), untersucht, welcher Agent sich für das nächsthöhere Protokoll angemeldet hat und sendet diesem Agenten ein Socket-Objekt mit dieser Verbindung. Die Nachricht lautet `connect` und enthält das Socket-Objekt im ersten Attachment. Ein Agent kann sich mit einer `unsubscribe`-Nachricht wieder abmelden. Dies geschieht automatisch, wenn der Agent die Plattform verlässt.

Der Communication Agent ist auch für die Zustellung interner Nachrichten zuständig (siehe 4.7 *Kommunikation*, Seite 57). Wegen seiner grundlegenden Bedeutung für die Architektur muss er bereits mit der Plattform gestartet werden.

Java-Applets dürfen im Browser nur Netzverbindungen zu dem Rechner aufbauen, von dem aus sie geladen wurden, siehe f) *Web Agent*, Seite 54. Die Web-Agent-Applets können daher zunächst nur auf eine Plattform zugreifen. Wer diese Einschränkung umgehen will, kann das Proxy-Plugin einsetzen; mit dessen Hilfe können sich die Applets zu beliebigen anderen Hosts verbinden lassen. Der Einsatz dieses Plugins will aber wohl überlegt sein, umgeht es doch ein wichtiges Sicherheitsfeature des Browsers. Durch die Realisierung als Plugin ist es jederzeit möglich (ohne Änderung am Communication Agent), ein eigenes Plugin zu nutzen, das hier eine zusätzliche Zugriffskontrolle durchführt.

b) Migration Agent

Der Migration Agent ist für die Migration der Agenten zuständig. Er implementiert dazu das WAMP-Protokoll, siehe 4.8.2 *WAMP: WILMA Agent Migration Protocol*, Seite 66. Er verarbeitet einige Nachrichten von Agenten, die diese über vereinfachende Methoden ihrer Basisklasse senden: `requestMove()`, `requestCopy()`, `requestKill()` und `requestBirth()`. Durch einen speziellen Mechanismus (Distribute) kann der Migration Agent alle Agenten, die dies wünschen, auf alle Plattformen verteilen, die sich über einen Neighbour Agent bekannt gemacht haben.

Der Migration Agent muss zusammen mit der Plattform gestartet werden, ohne ihn kann kein Agent auf die Plattform migrieren – und damit natürlich auch er selbst nicht. Er ist auch für die Prüfung zuständig, ob ein Agent erwünscht ist (6.1 *Überprüfung neuer Agenten*, Seite 93).

c) Security Agent

Der Security Agent ist für zentrale Sicherheitsbelange in MobMan verantwortlich. Er kümmert sich um die Authentifizierung (siehe 6.2 *Authentifizierung*, Seite 95) und wird dazu von anderen Systemagenten genutzt, etwa vom Communication Agent und vom Migration Agent. Er verarbeitet `verify`-Nachrichten, die alle benötigten Daten und, je nach Verfahren, auch ein Verbindungsobjekt enthalten.

Der Security Agent kann die MobMan-Plattform auch mit SSL-Unterstützung ausstatten (siehe 6.4 *Gesicherte Kommunikation mit SSL/TLS*, Seite 106). Für diesen Zweck tauscht dieser Systemagent das Socket-Interface-Objekt durch SSL-Sockets aus. Die Konfiguration muss hierfür folgende Einträge enthalten:

```
MobMan.Security.SecurityAgent {
  Socket {
    class = MobMan.Security.Utils.MobManSocketSSL
    certfile = certs/platform-NameDerPlattform.p12
    passphrase = cleartextPassphrase
    trustedsigners {
      certs/ca-NameDerCA.der
    }
  }
}
```

Der Security Agent nutzt dazu die Methode `setSocket(config)`, die die Klasse `MobManSocket` bereitstellt. Ab diesem Zeitpunkt sind alle Sockets, die mit `new MobManSocket()` erstellt werden, SSL-Sockets. SSL lässt sich somit transparent einfügen, ohne an den Agenten Änderungen vornehmen zu müssen. Die Umstellung ist exklusiv: Entweder kommuniziert die Plattform mit SSL oder ohne, beide Varianten gleichzeitig sind nicht möglich.

Der Security Agent kann auch den Java-Security-Manager setzen, siehe 5.7 *Security Manager*, Seite 90. Auch hier erfolgt die Konfiguration über den Security Agent und dessen Eintrag in der Konfigurationsdatei:

```
MobMan.Security.SecurityAgent {
  SecurityManager {
    class = MobMan.Security.Utils.MobManSecurityManager
    PermitAcceptFrom {
      129.187.105.0-129.187.105.255
    }
    # weitere Parameter...
  }
}
```

Falls der Security Agent das Socket ersetzen soll (für SSL), muss er zusammen mit der Plattform gestartet werden, noch vor dem Communication Agent.

d) Neighbour Agent

Durch diese Agenten ist eine virtuelle Vernetzung von Plattformen möglich. Jeder Neighbour Agent repräsentiert eine andere Plattform und ist als Wegweiser zu dieser Plattform zu verstehen. Mit seiner Hilfe können Agenten den Weg zu neuen Plattformen finden. Durch eine lokale Suche nach diesen Wegweisern erfährt ein Agent, welche Nachbarn bekannt sind:

```
Vector neighbours=platform.findAgentsByGroup("Neighbours");
```

e) Platform Interface Agent

Pia, der Platform Interface Agent, stellt ein Interface zur Administration der Plattform zur Verfügung. Die eigentliche Funktionalität stellen dynamisch ladbare Module bereit (siehe Tabelle 4.1 *Pia-Module*, Seite 53). Ein eigener Agent, der Pia Service Agent (PISA), verteilt die Module. Zur Kommunikation kommt das WRAP-Protokoll zum Einsatz, siehe 4.8.3 *WRAP: WILMA Remote Administration Protocol*, Seite 69. Für den Client sind zwei Implementierungen verfügbar, ein textbasiertes Interface und ein GUI auf Basis des Java-AWT.

Modul	Aufgabe
`GetPlatforms`	Liste aller Plattformen abfragen.
`GetAgents`	Liste aller Agenten auf einer bestimmten Plattform oder auf allen Plattformen abfragen.
`ChangeModules`	Pia-Module ändern.
`ChangeUser`	Liste der bekannten User ändern (betrifft `SecurityAgent`).
`Shutdown`	Plattform beenden oder neu starten.
`KillAgent`	Agent beenden.
`TerminateAgent`	Agent auffordern, sich selbst zu beenden.
`StartAgent`	Neuen Agenten starten.
`ViewAgentResult`	Ergebnisse der Agenten anzeigen.
`SaveAgentResult`	Ergebnisse lokal auf der Plattform speichern.
`AgentMessage`	Wartet auf eine Nachricht des Agenten.

Tabelle 4.1 *Pia-Module*

Für einige Aufgaben benötigen der Platform Interface Agent und einige seiner Module besondere Privilegien, daher muss dieser Agent zusammen mit der Plattform gestartet werden.

f) Web Agent

Der Web Agent stellt ein weiteres administratives Interface für die Plattform dar. Er arbeitet als HTTP-Server und stellt ein HTML-Interface bereit, das mit jedem Webbrowser genutzt werden kann. Beispielsweise zeigt dieser Systemagent eine Liste aller Agenten, die sich auf der Plattform befinden. Durch ein Java-Applet kann er die Anzeige dieser Liste im Browser auch permanent aktuell halten. Bietet ein Agent ein eigenes GUI an, so kann man dieses über den Browser ebenfalls nutzen.

g) GUI Agent

Der GUI Agent stell lokal auf dem Host, auf dem die Plattform läuft, eine grafische Oberfläche dar. Andere Agenten können auch Zugriff auf dieses GUI erhalten, indem sie dem GUI Agent eine Nachricht mit der entsprechenden AWT-Komponente (Label, Panel, ...) senden.

h) Police Agent

Der Police Agent zeichnet alle Verstöße gegen Sicherheitsregeln auf und stellt sie anderen Agenten über sein Results-Interface zur Verfügung. Eine erweiterte Version könnte auf Verstöße passiv reagieren (E-Mail an Administrator) oder auch selbst aktiv werden, etwa um einen auffällig gewordenen Agenten zu beenden.

i) Log Agent

Der Log Agent implementiert, ebenso wie der Communication Agent, das `MessageDispatcher`-Interface. Dadurch erhält er Zugriff auf alle Nachrichten, die auf der Plattform versendet werden. Er protokolliert diese Messages in einer Datei. Aus Sicherheitsgründen (und weil er ein `MobMan.Core`-Interface implementiert) muss dieser Agent zusammen mit der Plattform gestartet werden – vorausgesetzt, dieser Dienst ist überhaupt erwünscht. Sinnvoll ist er etwa für Debugging-Zwecke.

j) Blackboard

Über das Blackboard können Agenten öffentliche Mitteilungen asynchron austauschen, vergleichbar einem News-Server. Um eine Mitteilung im Blackboard abzulegen, sendet sie ein Agent einfach als Attachment einer Add-Message-Nachricht an diesen Systemagenten. Um die Nachricht „Hallo!“ dort abzulegen, ist folgender Aufruf nötig:

```
newMessage("Blackboard", "addMessage").addAttachment("Hallo!").send();
```

Mehr zum Message-Mechanismus, seiner Anwendung und zu obiger Syntax ist in 4.7.5 *Realisierung der Nachrichten in MobMan*, Seite 62 zu finden. Das Blackboard kann über das Results-Interface abgefragt werden.

k) Mailbox

Über die Mailbox können Agenten private Nachrichten asynchron austauschen:

```
sendMessage("Mailbox", "mailto AgentAbc: Hallo, wie geht's?");
```

Agenten können nur die Nachrichten lesen, die auch für sie bestimmt sind. Dabei dient allerdings derzeit nur der Name des Agenten als Unterscheidungskriterium.

```
Vector reply=sendMessage("Mailbox", "read");
```

l) Persistence Agent

Dieser Systemagent stellt ein automatisches Backup zur Verfügung. Jeder Agent, der nach dem Persistence Agent auf der Plattform ankommt, wird auf der lokalen Festplatte gesichert. Die Sicherung löscht der Persistence Agent erst wieder, wenn der Agent erfolgreich auf eine andere Plattform migriert ist. Beim Neustart kann die Plattform die gespeicherten Agenten aus dem Backup lesen und neu starten. Dabei wird die `init()`-Methode nicht mehr aufgerufen, da dies bereits vor dem Backup geschehen ist.

Um zusätzliche Daten zu speichern, kann sie ein Agent als Save-Data-Nachricht an den Persistence Agent senden. Nach dem Neustart sendet der Persistence Agent diese Daten in einer Restore-Data-Nachricht zurück an den Agenten.

```
newMessage("PersistenceAgent", "saveData").addAttachment(obj).send();
```

Der Persistence Agent muss zusammen mit der Plattform gestartet werden.

4.6 Migration

4.6.1 Migration zwischen Plattformen

a) Überblick

Die Migration zu anderen Plattformen ist die besondere Fähigkeit mobiler Agenten. MobMan überträgt dabei nur die auf der Zielplattform noch nicht vorhandenen Klassen. Der Name der Klassen allein ist zur Unterscheidung aber nicht ausreichend. Verschiedene Agenten können Klassen mit gleichem Namen verwenden, oder verschiedene Versionen einer weitgehend identischen Klasse. Die Unterscheidung der Klassen muss daher unabhängig von ihrem Namen sein. Dazu kann nur der konkrete Code benutzet werden, da andernfalls alle Autoren kooperieren müssten oder eine zentrale Instanz zur Namensvergabe nötig wäre. Beides ist mit den Designkriterien von MobMan nicht vereinbar.

Die Lösung benutzt kryptografisch starke Hash-Algorithmen. Diese Hashes gelten als kollisionsfrei, daher haben auch nur geringfügig unterschiedliche Klassen völlig unterschiedliche Hash-Werte. Die Größe des Hash-Codes ist konstant, er ist deutlich kleiner als das komplette Class-File, so dass dieses Verfahren beide Ziele erfüllt: Datenreduktion bei der Übertragung und sicheres Erkennen unterschiedlicher Klassen.

Da MobMan auf zentrale Instanzen verzichtet (siehe 4.1 *Designkriterien*, Seite 38), muss bereits bei der Migration neben dem Zustand der vollständige Code mit übertragen werden, soweit er auf der Zielplattform noch nicht vorhanden ist. Zusätzlich benötigte Klassen können nicht erst bei Bedarf nachgeladen werden. Um tatsächlich alle benötigten Klassen zu erfassen, führen die Agenten eine Liste ihrer Klassen mit sich, siehe c) *Benötigte Klassen*, Seite 56. Zu den Details der Instanziierung von Klassen auf der Plattform siehe 5.3 *Trennung der Klassen*, Seite 78.

Andere Plattformen, etwa Ajanta, nutzen ein zentrales Code-Repository, zu dem immer eine Verbindung möglich sein muss. Dadurch können sie Teile des Agentencodes jederzeit nachladen. Diese Codeserver können zwar bei einigen Systemen redundant ausgelegt sein [Hohl 97b], es ist aber stets eine Verbindung von der Plattform zu einem der Codeserver nötig und damit eine weitere Abhängigkeit geschaffen. Zusätzlich muss bei redundanten Code-Repositories der Code auf die einzelnen Server repliziert und konsistent gehalten werden.

b) Praxis

Die schwache Migration in MobMan überträgt zwar Code und Daten, nicht aber den Callstack. Bei der Ankunft auf einer neuen Plattform wird daher immer die `run()`-Methode aufgerufen, vorausgesetzt es handelt sich um einen aktiven Agenten. Folgende Methoden lösen die Migration aus:

```
requestMove("host", port);
requestCopy("host", port);
```

Bei `requestMove()` verlässt der Agent die aktuelle Plattform. Bei `requestCopy()` kopiert er sich auf die neue Plattform, läuft lokal aber weiter. Schlägt die Migration fehl, dann liefern beide Methoden False als Rückgabewert zurück. Mit `requestKill()` kann sich ein Agent selbst beenden. Durch eine `return`-Anweisung in der `run()`-Methode wird aus einem aktiven Agenten ein passiver. Mit der Methode `requestBirth(classname, config)` kann ein Agent auch einen neuen Agenten starten.

Wenn auf der neuen Plattform bereits ein Agent mit gleichem Namen vorhanden ist, hängt der weitere Ablauf vom Parameter `multipleAllowed` ab. Bei Benutzeragenten ist er per Default auf True gesetzt, bei Systemagenten auf False. Nur bei True wird der neue Agent zusätzlich auf der Plattform zugelassen. Wenn diese Variable False ist, kommt die Variable `replaceAllowed` (Default: True) noch zum Zuge. Wenn sie True ist (und `multipleAllowed` False), dann ersetzt der neue Agent den alten.

c) Benötigte Klassen

Für die Migration ist eine Liste der Klassen nötig, die zu einem Agenten gehören. Die Klasse des Agenten selbst und alle Vaterklassen fügt MobMan automatisch zu dieser Liste hinzu. Alle anderen benötigten Klassen muss der Konstruktor des Agenten selbst benennen:

```
requiredClasses.addElement("UserAgents.etc.ClassName");
requiredClasses.addElement("UserAgents.etc.ClassName$InnerClassName");
```

Klassen aus dem Paket `MobMan.Core` dürfen in dieser Liste nicht enthalten sein, sie würde der Classloader der neuen Plattform aus Sicherheitsgründen nicht akzeptieren. Die MobMan-eigenen Klassen sind auf der Zielplattform natürlich vorhanden.

d) Sicherheitsüberlegungen

- Bei der Migration muss die Plattform auf die korrekte Trennung der Klassen achten (siehe 5.3 *Trennung der Klassen*, Seite 78).
- Die Entscheidung, ob ein Agent zugelassen wird oder nicht, kann an mehreren Stellen erfolgen. Neben den üblichen Java-Mechanismen werden auch die Absenderplattform und der Agent authentifiziert. Siehe 6.1 *Überprüfung neuer Agenten*, Seite 93.

- Die `multipleAllowed`- und `replaceAllowed`-Mechanismen bergen eine hohe potenzielle Gefahr. Mit `multipleAllowed=false` kann ein Benutzer einen Denial-of-Service-Angriff ausführen, gleichnamige Agenten könnten nicht mehr auf die Plattform migrieren. Wenn ein Agent von sich aus `replaceAllowed=true` setzt, kann ein Angreifer diesen Agenten komplett ersetzen.
- Sinnvoll und sicher sind beide Mechanismus eigentlich nur, wenn der vorhandene Agent und der neu auf die Plattform migrierende Agent dem selben Benutzer gehören, sich also mit der selben Kennung authentifiziert haben, oder wenn der bereits vorhandene Agent ein Systemagent ist.

4.6.2 Migration auf die erste Plattform

Um überhaupt den Weg auf eine Plattform zu finden, bieten sich zwei Lösungen an: Ein Agent kann zusammen mit der Plattform gestartet werden oder durch einen Launcher auf die erste Plattform migrieren. Der Start zusammen mit der Plattform wird durch deren Konfigurationsdatei festgelegt, im Eintrag `startAgents` stehen die Namen dieser Agenten:

```
MobMan.Core.MobManPlatform {
  startAgents {
    MobMan.Security.SecurityAgent
    MobMan.SystemAgents.CommunicationAgent
    MobMan.SystemAgents.MigrationAgent
    MobMan.Pia.PlatformInterfaceAgent
    MobMan.SystemAgents.WebAgent
  }
}
```

Die Konfiguration der Agenten wird dann in weiteren Abschnitten festgelegt (siehe Anhang A *Konfigurationsdateien von MobMan*, Seite 141).

Für die zweite Lösung – Agenten nachträglich in eine Plattform laden – ist die Klasse `Agent-Launcher` im Package `MobMan.Utils` zuständig. Sie hat eine `main()`-Methode und ist damit als eigenständiges Programm lauffähig. Sie nimmt die Klassen `MobManConfig` und `Migration-Agent` zu Hilfe und verhält sich gegenüber dem Migrationsziel wie eine weitere MobMan-Plattform. Die Anwendung dieser Klassen wird durch das Shellskript `launch` vereinfacht, siehe Anhang B *Das Launch-Programm*, Seite 152.

4.7 Kommunikation

MobMan setzt, im Gegensatz zu vielen anderen Plattformen [Baumann 97a], auf eigene Kommunikationsmechanismen und verzichtet darauf, universelle Methoden wie RMI zu benutzen. Dies dient vor allem der besseren Überschaubarkeit und leichteren Verifizierbarkeit der Implementierung. Durch die schlanke Realisierung lässt sich das komplette Konzept leichter nachvollziehen. Über die `AgentProperties` ist bereits eine sehr einfache Form der Kommunikation möglich (siehe 4.5.3 *Properties*, Seite 49); nachfolgend werden komplexere Mechanismen betrachtet.

4.7.1 Aufgabenstellung

Folgende Teile von MobMan sollen miteinander kommunizieren können:

- Agenten (aktive und passive)
  - Messages senden/empfangen.
  - Properties anbieten/abfragen.
  - Events empfangen.
- Plugins (und Pia-Module)
  - Messages senden (unter der ID ihres Agenten).
  - Properties abfragen.
- Plattform
  - Messages senden. Empfangen ist unnötig, da alle Agenten eine Referenz auf die Plattform besitzen und somit Java-Methoden nutzen können.
  - Keine eigenen Properties. Diese sind ebenfalls wegen der Referenz unnötig.
  - Events senden. Empfangen ist nicht nötig.

Gerade bei der Kommunikation ist besonderes Augenmerk auf den Aspekt Sicherheit zu legen [Vitek 97a]. Wenn ein Agent nur auf bestimmte anfragende Agenten reagieren soll, muss er sicher wissen, von welchem Agenten eine Anfrage tatsächlich stammt. Diese Aufgabe muss das Agentensystem lösen. Als Kommunikationsmechanismen kommen verschiedene Verfahren in Frage, etwa Datagramme oder Methodenaufrufe [Vitek 97b].

4.7.2 Events

Beim Auftreten bestimmter Ereignisse müssen die Agenten informiert werden. Events haben keinen Rückgabewert, können aber Parameter enthalten. Bei MobMan sendet nur die Platform Events. Da diese Quelle als sicher modelliert ist, ist keine Authentifizierung nötig. Eine Entkopplung des Senders vom Empfänger ist dennoch erforderlich, um die Plattform nicht angreifen zu können.

a) Aktuell: Methoden der Basisklasse

Im aktuellen MobMan werden Events über Methoden der Basisklassen `Agent` und `AgentPlugin` zugestellt. Die Methoden `agentArrived` und `agentLeft` informieren alle Agenten und ihre Plugins über die Migrationsbewegungen anderer Agenten. Über eine Schleife ruft die Plattform diese Methode bei allen im System vorhandenen Agenten und ihren Plugins auf, wobei die Default-Implementierung leer ist.

Vorteile:

- Sehr einfach zu benutzen. Der Agent muss nur die entsprechende Methode neu implementieren.

Nachteile:

- Skaliert schlecht mit der Anzahl an Agenten. Jeder weitere Agent führt zu einem zusätzlichen Methodenaufruf.
- Um auch Plugins informieren zu können, ist die Schleife noch aufwändiger.
- Weitere Events lassen sich nur durch Änderung der Agenten-Basisklasse hinzufügen.
- Ein Event-Empfänger kann die weitere Zustellung blockieren: Er muss dazu nur eine Endlosschleife in der gerufenen Funktion implementieren.

b) Einzelne Methode mit zwei Parametern

Unterschiedliche Events könnten auch über eine einzelne Methode an die Agenten weitergegeben werden, etwa `processEvent(String eventType, Object parameter)`. Der Agent müsste dann in der Implementierung die Event-Typen unterscheiden.

Vorteile:

- Neue Events sind sehr leicht hinzuzufügen.

Nachteile:

- Auch hier muss der Event an jeden Agenten weitergereicht werden, unabhängig davon, ob er daran interessiert ist oder nicht.
- Zusätzlich ist jedes mal ein Vergleich der Event-Typen notwendig. Dadurch entsteht ein noch größeres Performance-Problem als in der ersten Variante.

c) Broadcast-Messages

MobMan-Messages lassen sich auch für Events benutzen, es wird einfach die Nachricht an alle lokal vorhandenen Agenten gesendet. Agenten, die nicht an diesem Event interessiert sind, ignorieren die Nachricht.

Vorteile:

- Kein zusätzlicher Mechanismus nötig.
- Weitere Events lassen sich einfach hinzufügen, ohne die Basisklasse zu modifizieren.

Nachteile:

- Schlechtere Performance, aufwändiger.
- Skaliert ebenfalls schlecht mit der Anzahl an Agenten.
- Plugins können derzeit keine Messages empfangen.

d) Subscription

Events werden nicht einfach an alle Agenten gesendet, sondern nur an diejenigen, die vorher ihr Interesse an einem bestimmten Event angemeldet haben. Ein ähnlicher Mechanismus kommt derzeit bei den Protokollen zur Anwendung (4.8.1 *MAP: MobMan Application Protocol*, Seite 66), allerdings kann sich dabei nur ein einziger Agent für jeden eingehenden Verbindungstyp interessieren. Bei Events sollten sich beliebig viele Agenten anmelden können, auch Plugins.

Der Event-Empfang selbst sollte über Messages, oder besser über eine eigene Methode abgewickelt werden, vergleiche b) *Einzelne Methode mit zwei Parametern*, Seite 59. Letzteres würde es beliebigen Objekten (die ein Interface implementieren, oder einfach allen Objekten des Typs `MobManObject`) ermöglichen, Events zu empfangen. Bei der Anmeldung wäre dann nur eine Referenz anzugeben.

Vorteile:

- Skaliert besser, da unnötige Kommunikation/Methodenaufrufe vermieden werden.

Nachteile:

- Alle Events laufen immer noch über eine Methode. Ein Listener-Prinzip hätte hier Vorteile, wäre aber aufwändiger und komplizierter.
- Höherer Verwaltungsaufwand: Es gilt eine weitere Liste zu verwalten. Die Agenten und ihre Objekte sind beim Beenden des Agenten auch wieder auszutragen. Hierfür wäre eine zweite Liste sinnvoll, die angibt, welcher Agent (ID) welche Objekte als Empfänger eingetragen hat.

4.7.3 Klassifikation von Nachrichten

Nachrichten unterscheiden sich eher subtil von Events. Events lassen sich im Grunde auch mit Nachrichten zustellen. Allerdings kann durch die Einschränkung der Event-Quelle auf die Plattform oder auf privilegierte Systemagenten auf einfachere, weniger abgesicherte Kommunikationsmechanismen zurückgegriffen werden, bis hin zum direkten Methodenaufruf.

Bei allgemeinen Nachrichten oder Messages (jedes `MobManObject` kann Nachrichten senden) sind zusätzliche Vorkehrungen nötig. Nachrichten lassen sich beispielsweise entsprechend folgender Kriterien klassifizieren:

a) Anzahl Empfänger

- *Unicast:* Genau ein Empfänger, vom Absender bestimmt.
- *Multicast:* Gruppe von Empfängern,
  - vom Absender einzeln festgelegt oder
  - die Empfänger tragen sich selbst in eine Gruppe ein.
- *Anycast:* Genau ein Empfänger aus einer Gruppe möglicher Empfänger, etwa für Load Balancing geeignet.
- *Broadcast:* Nachricht an alle.

b) Synchron/Asynchron

- *Synchron*
  - Der Absender wartet auf den Empfänger.
  - Die Ergebnisse können direkt zurückgegeben werden.
- *Asynchron*
  - Der Absender wartet nicht.
  - Die Ergebnisse können nur durch eine erneute Nachricht zugestellt werden.

4.7.4 Zustellungsmechanismen für Nachrichten

Für die Zustellung der Nachrichten bieten sich eine Reihe verschiedener Grundmechanismen an. Im Folgenden werden einige kurz vorgestellt.

a) Message Queue

Bei einer Message Queue werden die Nachrichten in eine Warteschlange eingetragen, aus der sich der Empfänger seine Nachrichten abholt. Dieses Verfahren lässt sich mit der Briefzustellung oder mit E-Mail-Systemen vergleichen. Die Message Queue kann sich an verschiedenen Orten befinden:

- *Outgoing Queue*
  - Jeder Absender erhält eine eigene Liste.
  - Diese Variante löst das Problem zeitweilig nicht zustellbarer Nachrichten. Diese können hier zwischengespeichert werden, um nach einer gewissen Zeit eine erneute Zustellung zu versuchen.
- *Zentrale Queue*
  - Eine einzelne Queue, in der alle Messages landen.
  - Die Eigenschaften sind ähnlich der Outgoing Queue, nur kümmert sich nun ein zentraler Dienst um die Zustellung.

- *Incoming Queue*
  - Jeder Empfänger unterhält eine eigene Liste.
  - Der Agent holt sich die nächste Nachricht selbst ab, sobald er wieder Nachrichten verarbeiten kann.

Die ersten beiden Versionen sind eher für die Inter-Plattform-Kommunikation geeignet; innerhalb einer Plattform erscheint die letzte Variante als am besten geeignet.
Vorteile:

- Entkopplung von Sender und Empfänger.

Nachteile:

- Queue-Verwaltung nötig, eigener Receive-Thread oder Receive-Aufruf nötig.
- Synchrone Messages sind schwierig zu implementieren (Warten auf Antwort). Es besteht die Gefahr der Blockierung: Eine Nachricht am Ende der Queue wird nicht verarbeitet, die nächste wäre aber wichtig.

b) Aufruf einer einzelnen Methode

Aktuell ruft ein Systemagenten direkt eine Methode der Empfänger auf (siehe 4.7.5 *Realisierung der Nachrichten in MobMan*, Seite 62).
Vorteile:

- Einfache Implementierung.
- Kaum Verwaltungsaufwand.

Nachteile:

- Keine Entkopplung der Threads. Bei synchronen Nachrichten kann der Empfänger, der im Thread des Absenders läuft, diesen Absender-Thread beenden.
- Agent muss sich um die Synchronisation kümmern, da der Nachrichtenempfang immer in einem anderen Thread läuft als der Agent selbst.

c) Verbindung

Die Kommunikation kann natürlich auch über eine direkte Verbindung stattfinden, etwa in Form von verbundenen `ObjectStream`-Objekten oder durch Sockets. Teilweise (etwa ffMAIN) wird HTTP benutzt.

d) Wahlfreier Methodenaufruf

Im Unterschied zu b) *Aufruf einer einzelnen Methode*, Seite 61 bestimmt bei Verfahren wie RPC, RMI oder CORBA der Absender (Client), welche Methode er aufruft, und nicht nur, welche Daten er übergibt. Aufgrund ihrer höheren Komplexität und mangelnden Überschaubarkeit kommen solche Verfahren bei MobMan nicht zum Einsatz.

e) Treffpunkt

In einigen Agentensystem (etwa Ara und Mole) finden sich virtuelle Treffpunkte innerhalb einer Plattform, in denen sich ein Agenten befinden muss, um mit anderen Agenten zu kommunizieren. Alle beteiligten Agenten müssen sich im selben Treffpunkt aufhalten. Treffpunkte können von Agenten dynamisch zur Laufzeit angelegt werden, der Zugang kann beschränkt und nur für bestimmte Agenten erlaubt sein. Beim Eintritt in einen Treffpunkt handelt es sich um keine Migration, sondern im Grunde um den Eintrag in eine Liste.

4.7.5 Realisierung der Nachrichten in MobMan

a) Adressierung

Der wichtigste Kommunikationsmechanismus in MobMan benutzt Messages. Der Empfänger einer Nachricht weiß dabei immer sicher, von welchem Agent sie abgesendet wurde (siehe 5.6 *Caller-ID*, Seite 83). Eine Message kann an den Namen eines Agenten gerichtet sein. Wenn mehrere Agenten mit diesem Namen auf der Plattform vorhanden sind, erhalten sie die Nachricht nacheinander. Alternativ kann auch die ID eines Agenten angegeben werden, dann ist die Zustellung eindeutig. Allerdings muss der Absender dazu die ID des Empfängers kennen.

Eine Adressierung ist mithilfe der `GlobalAgentID` auch über Plattformgrenzen hinweg möglich, siehe 4.8.4 *RAMP: Remote Agent Messaging Protocol*, Seite 71. Da die `GlobalAgentID` im Gegensatz zur lokalen ID nicht mehr fälschungssicher ist, kann der Empfänger durch die Methode `isFromRemote()` feststellen, ob die Nachricht lokal zugestellt wurde oder von einer entfernten Plattform stammt.

Nicht direkt adressierbar sind derzeit Plugins. Eine denkbare Lösung wäre, das Plugin über den Namen oder die ID seines Agenten anzusprechen, ergänzt durch den Plugin-Namen. Plugins könnten auch eine eigene ID erhalten (derzeit benutzen sie die ID ihres Agenten). Beim Absenden einer Nachricht sollte das Plugin dann sowohl als Individuum auftreten, als auch im Namen seines Agenten handeln können.

b) Praxis

Um eine Nachricht zu senden, benutzt der Agent eine der beiden folgenden Methoden aus seiner Basisklasse:

```
Vector v=sendMessage("destination", obj);
postMessage("destination", obj);
```

Bei `sendMessage()` handelt es sich um synchrone Kommunikation, der Absender wartet, bis alle Empfänger geantwortet haben und erhält einen Vektor mit allen Rückgabewerten zurück. Die Nachricht wird innerhalb des Threads des Absenders bearbeitet. Im Gegensatz dazu ist `postMessage()` asynchron, der Aufruf kehrt also unmittelbar zurück. Die einzelnen Empfänger werden durch einen unabhängigen Thread kontaktiert, der Absender erhält keine Ergebnisse.

Der Agent kann seine Nachricht (Java-Klasse: `Message`) auch selbst aufbauen. Er muss lediglich eine Methode der Basisklasse `MobManObject` benutzen, um das Objekt zu instanziieren (dies ist aus Sicherheitsgründen nötig, um den Absender identifizieren zu können). Das Message-Objekt kann so auch Anhänge erhalten:

```
Message msg = newMessage("destination", obj);
msg.addAttachment(par1);
msg.addAttachment(par2);
Vector v=msg.send() // oder: msg.post();
```

Dieser Ablauf lässt sich auch kompakter formulieren:

```
Vector v = newMessage("destination", obj).addAttachment(par1).addAttachment
           (par2).send()
```

Dies funktioniert, da die Methode `addAttachment()` als Rückgabewert eine Referenz auf die Nachricht zurückgibt (`this`-Referenz). Java wertet obige Zeile von links nach rechts aus: die Methode `newMessage()` (in der `Agent`-Basisklasse definiert) erzeugt ein `Message`-Objekt. Auf dieses Objekt wird die Methode `addAttachment()` angewendet, die eine Referenz auf das selbe `Message`-Objekt zurückgibt. Das ist im Beispiel für den zweiten Parameter wiederholt, bis schließlich die `Message`-Methode `send()` für das Versenden sorgt.

Die Liste der möglichen Empfänger einer Nachricht kann über Flags (Methode: `addFlag()`) des Message-Objekts weiter eingeschränkt werden. Zwei Flags sind möglich:

- `Message.SYSTEMAGENTS`: Nachricht ist nur für Systemagenten bestimmt.
- `Message.USERAGENTS`: Nachricht ist nur für Benutzeragenten bestimmt.

Zur eindeutigen Adressierung eines Agenten kann als Empfänger-Angabe statt des Namens auch dessen ID benutzt werden. Um Nachrichten zu Agenten auf anderen Plattformen zu schikken, kann als Empfänger eine Instanz der Klasse `GlobalAgentID` eingetragen werden. In dieser Klasse (sie erbt von `HostAndPort`) sind neben dem Host und dem Port der Plattform vor allem der Name des Agenten und optional eine Identifikationsnummer vermerkt. Diese Nummer dient dazu, gleichnamige Agenten zu unterscheiden.

Die Identifikationsnummer stammt aus der Basisklasse `ID`, sie kann mit `getSequenceNumber()` abgefragt werden. Eine statische Klassen-Variable, die der Konstruktor inkrementiert, sorgt dafür, dass die Nummer innerhalb einer Plattform eindeutig ist – so lange kein Überlauf eintritt und so lange die Plattform nicht neu gestartet wird. Da diese Nummer aber nur als zweites Kriterium eingesetzt wird, zur Unterscheidung namensgleicher Agenten, ist das Risiko minimal.

c) Messages empfangen und verarbeiten

Um Nachrichten zu empfangen, muss ein Agent die Methode `receiveMessage()` neu implementieren. Sie erhält das Message-Objekt als Parameter. Der Empfänger kann über dieses Objekt auch den Absender ermitteln.

```
public Object receiveMessage(Message msg)
{
  String sendername=msg.getName();
  String owner=msg.getOwner();
  ID id=msg.getID();

  if (msg.equals("Test"))
  {
    String par1=(String)msg.getAttachment(1);
    Vector par2=(Vector)msg.getAttachment(2);
    ...
    return new Boolean(true);
  }
  else if (msg.getMessage() instanceof Hashtable)
  {
    Hashtable hash=(Hashtable)msg.getMessage();
    ...
  }
  else
    return super.receiveMessage(msg);

  return null;
}
```

Falls die Nachricht nur aus einem String besteht, kann ihr Inhalt direkt mit der `equals()`-Methode des `Message`-Objekts verglichen werden. Andernfalls ist der Zugriff auf den Inhalt der Nachricht mithilfe von `getMessage()` nötig. Der Test mit `instanceof` hilft bei der Typumwandlung.

d) Ablauf

Das Senden von Nachrichten findet bei MobMan in zwei Schritten statt. Im ersten Schritt erhalten alle Systemagenten, die das Interface `MessageDispatcher` implementieren, jede Nachricht. Sie sind dann für die weitere Zustellung verantwortlich. Derzeit erledigt das der Communication Agent.

Durch das zweistufige Verfahren kann der Dispatcher leicht ausgetauscht oder ein weiterer eingefügt werden. Dadurch ist der Log Agent möglich geworden: Dieser Systemagent protokolliert alle Nachrichten, die in der Plattform versendet werden.

e) Sicherheitsüberlegungen

- MobMan bietet dem Empfänger eine eindeutige Identifizierung des Absenders (siehe 5.6 *Caller-ID*, Seite 83). Dadurch kann der Empfänger eine qualifizierte Entscheidung treffen, wie er die Nachricht bewerten soll.
- Die Nachricht darf nur vom jeweiligen Empfänger gelesen werden. Dieser Schutz ist gegeben, allerdings ist der Name des Agenten als Adresse nicht ausreichend, da er beliebig vergeben werden darf.
- Die bloße Kommunikation teilt beiden Partnern noch keine Referenzen auf den jeweils anderen Agenten mit, dadurch bleibt deren Trennung gewahrt.
- Empfänger von Nachrichten können einen Denial-of-Service-Angriff durchführen: Wenn sie statt auf die Nachricht zu antworten eine Endlosschleife implementieren, blockieren sie den Absender (bei synchronen Nachrichten) und eventuelle weitere Empfänger erhalten die Message nicht mehr.
- Da der Empfang synchroner Messages im Thread des Absenders läuft, kann der Empfänger diesen Thread beenden.
- Eine Lösung wäre durch eigene Threads pro Kommunikation, verbunden mit Timern, möglich. Der zusätzliche Overhead würde die Performance allerdings negativ beeinflussen. In der jetzigen Implementierung muss der Absender ein Minimum an Vertrauen in seine Empfänger aufweisen, oder selbst einen solchen Schutzmechanismus implementieren.

4.8 Protokolle

MobMan definiert eigene Protokolle zur Migration und Kommunikation von Agenten, zur Administration der Plattform (beziehungsweise allgemeiner für den Zugang zur Plattform), und verwendet gleichzeitig HTTP. Alle Protokolle werden über den selben TCP-Port abgewickelt. Um eine Trennung zu ermöglichen, sind die MobMan-Protokolle in zwei Schichten unterteilt (siehe Abbildung 4.5 *MobMan-Protokolle*, Seite 65); die Nummerierung entspricht der DoD-Einteilung.

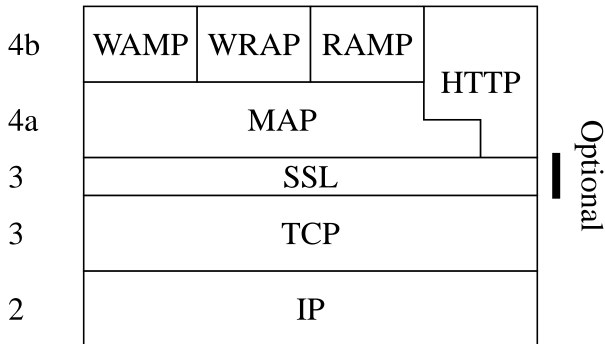

Abbildung 4.5 *MobMan-Protokolle*

Die einzelnen Schichten erfüllen folgende Aufgaben:

- *SSL, Schicht 3* (Secure Sockets Layer)
  - Optional.
  - Zwischen den Schichten 3 und 4 angesiedelt.
  - Transparent für die oberen Schichten.
  - Dienst: Authentifizierung, Unversehrtheit der Daten und Verschlüsselung.
- *MAP, Schicht 4a* (MobMan Application Protocol)
  - Dünne Protokollschicht direkt über TCP (oder SSL).
  - Dient der Auswahl des verwendeten Applikationsprotokolls.
  - Implementiert in `CommunicationAgent`.
- *WAMP, Schicht 4b* (WILMA Agent Migration Protocol)
  - Setzt auf MAP auf.
  - Überträgt die Agenten.
  - Implementiert in `MigrationAgent`.
- *WRAP, Schicht 4b* (WILMA Remote Administration Protocol)
  - Setzt auf MAP auf.
  - Administration (und allgemeiner Zugang zu) einer Plattform.
  - Implementiert in Pia (Platform Interface Agent).
- *RAMP, Schicht 4b* (Remote Agent Messaging Protocol)
  - Setzt auf MAP auf.
  - Kommunikation der Agenten miteinander.
  - Implementiert in `Message` (Client) und `CommunicationAgent` (Server).
- *HTTP, Schicht 4* (Hypertext Transfer Protocol)
  - Wird von MAP erkannt, daher parallel über den selben Port möglich.
  - Liefert Informationen über die Plattform.
  - Zugang zur Plattform auch über das MobMan-Applet (benutzt WRAP-Protokoll).
  - Implementiert in `WebAgent`.

4.8.1 MAP: MobMan Application Protocol

Das MobMan Application Protocol MAP dient vor allem dazu, verschiedene Protokolle über einen einzelnen Port zu leiten. Das eigentliche Multiplexing übernimmt bereits das TCP-Protokoll, es kann gleichzeitig mehrere Verbindungen zu einem Port offen halten und unterscheiden. Aufgabe des Communication Agent bei der MAP-Verarbeitung ist es, das konkret verwendete Protokoll zu erkennen und die Verbindung an den zuständigen Agenten weiterzuleiten. Nach dem MAP-Header übernimmt also ein anderer Agent die Protokollverarbeitung.

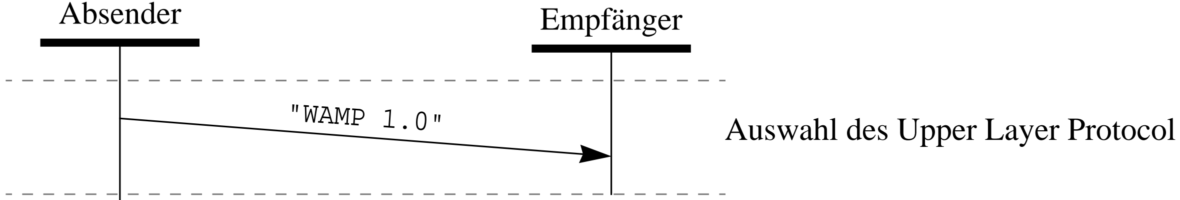

Abbildung 4.6 *MAP-Protokoll*

Zu Beginn der Verbindung überträgt der Absender nur eine einfache ASCII-Zeichenkette. Damit ist die Kompatibilität zu HTTP möglich: die erste Zeile beginnt dort bei einem normalen Aufruf von Webseiten mit „`GET` *`URL`*“. Je nach Protokollbezeichner wird die weitere Bearbeitung an den Agenten übergeben, der sich vorher für dieses Protokoll beim Communication Agent angemeldet hat.

Der Protokoll-Header kann zusätzliche Parameter enthalten (in Abbildung 4.6 *MAP-Protokoll*, Seite 66 ist dies „`1.0`“). Diese Parameter werden vom jeweiligen Agenten ausgewertet und können beispielsweise dazu dienen, verschiedene Protokollversionen zu unterscheiden.

4.8.2 WAMP: WILMA Agent Migration Protocol

Mithilfe von WAMP (WILMA Agent Migration Protocol) migrieren Agenten auf andere Plattformen. Zu diesem Zweck können sich die Plattformen zunächst authentifizieren und entscheiden, ob die Migration auch tatsächlich erwünscht ist.

a) Protokoll

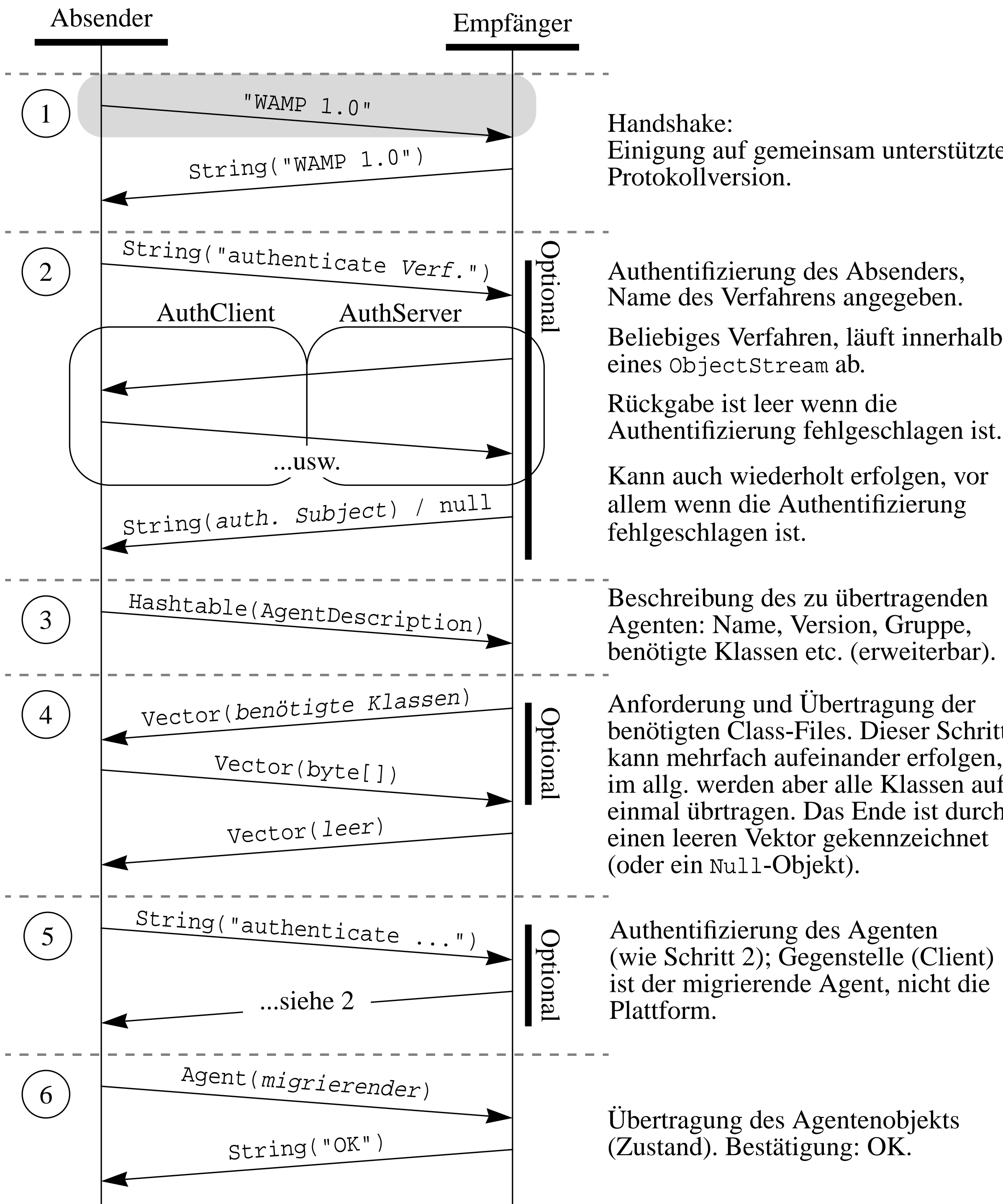

Abbildung 4.7 *WAMP-Protokoll*

Das Protokoll (Abbildung 4.7 *WAMP-Protokoll*, Seite 67) basiert auf Java-Streaming, es werden serialisierte Objekte übertragen. Lediglich die erste Zeile (grau hinterlegt) stammt aus MAP und wird als reiner ASCII-Text übertragen. Die einzelnen PDUs lassen sich durch die Klasse des übertragenen Objekts unterschieden, teilweise (etwa bei Strings) auch durch den Inhalt. An jeder Stelle kann `String("DENIED")` vorkommen; dies führt zum Abbau der Verbindung.

Die Authentifizierung des Agenten selbst findet erst nach der Übertragung der Klassen (Code, noch keine Daten) statt. Dadurch kann der Code auch für die Authentifizierung benutzt werden, beispielsweise bei signiertem Code ist dies eine Grundvoraussetzung. Der Zustand des Agenten eignet sich hierfür nicht, da er sich laufend ändert. Die Überprüfung findet so früh wie möglich statt; siehe dazu auch [Binder 02].

b) Sicherheitsüberlegungen

- Die Zielplattform fordert (um Bandbreite zu sparen) nur die Klassen an, die noch nicht lokal vorhanden sind. Der hierfür nötige Cache kann nicht einfach nur auf den Namen der Klassen basieren, da zum einen verschiedene Entwickler den gleichen Klassennamen für verschiedene Klassen verwenden können, aber auch verschiedenen Versionen einer Klasse gleichzeitig im Umlauf sein können. Zur Identifikation einer Klasse ist ihr Name also nicht ausreichend. Den vollständigen Bytecode zu verwenden wäre zwar ideal, er müsste dazu aber auch vollständig übertragen werden, wodurch der Vorteil des Caches verloren ginge. Die Lösung besteht in Hash-Codes, die aber kryptografischen Ansprüchen genügen müssen, um folgenden Angriff zu verhindern:
  - Programmierer A entwickelt Klasse X.
  - B schreibt ebenfalls eine Klasse mit Namen X und gleichem Interface, aber modifiziertem Verhalten. Er sorgt dafür, dass ihr Bytecode den selben Hash-Wert hat wie die Klasse X des Programmierers A.
  - B sendet einen Agenten, der X verwendet, zur Plattform P. Dort ist Klasse X noch unbekannt, es wird die Implementierung von B geladen.
  - Nun sendet A seinen Agenten zu P. P hat bereits eine, wie sie glaubt, passende Klasse X und verwendet diese statt der Implementierung von A

  Der Hash-Algorithmus muss also kollisionsfrei (eigentlich kollisionsarm) sein, um diese Situation nicht absichtlich oder versehentlich erzeugen zu können. Es darf nicht möglich sein, einen Bytecode zu konstruieren, der einen vorgegebenen Hash-Code ergibt. Üblicherweise werden hierfür die Crypro-Hash-Funktionen SHA-1 (Secure Hash Nr. 1) und MD5 (Message Digest Nr. 5) verwendet.
- Die Übertragung an sich ist natürlich beliebig angreifbar. Als Gegenmaßnahme kommt SSL zum Einsatz, siehe 6.4 *Gesicherte Kommunikation mit SSL/TLS*, Seite 106.
- Die Plattformen können sich zwar innerhalb der Verbindung authentifizieren. Ohne eine zusätzliche Sicherung der Übertragungsschicht etwa mit SSL fehlt allerdings die Bindung an die weitere Übertragung. Ein aktiver Angreifer könnte die Verbindung übernehmen (Session Hijacking) und damit die Rechte des Absenders erlangen. Die Sicherheit wäre vergleichbar mit der von Telnet-Logins.
- Die Authentifizierung in SSL und in Schritt 2 bezieht sich auf die beteiligten Plattformen, daher ist für die Authentifizierung des Agenten ein eigenes Verfahren nötig (in Schritt 5). Die Information hieraus ist aber, je nach Verfahren, nicht an den tatsächlich übertragenen Agenten gebunden. Die Absende-Plattform könnte einen anderen Agenten senden als sie authentifizieren lässt; dieser Angriff wäre nur bei signiertem Code zu ver-

hindern. Es ist also im Allgemeinen eine weitere lokale Authentifizierung nötig. Dennoch ist der Schritt 5 eine wertvolle Entscheidungshilfe, ob ein Agent überhaupt auf eine Plattform migrieren darf, noch bevor ein Stück seines Codes ausgeführt wurde.

- Der Zustand des Agenten (Daten) wird erst nach der Authentifizierung übertragen. Neben der offensichtlichen Bandbreiteeinsparung beim Fehlschlagen der Authentifizierung hat dies zur Folge, dass die Zielplattform, wenn sie den Agenten ablehnt, nicht über seine privaten Daten verfügt. Dieser Schutz ist aber nur gering, da die Plattform den Agenten zunächst akzeptieren könnte, um ihn dann doch nicht auszuführen. Auf diesem Weg wäre sie wieder im Besitz der Daten.
- Wie wichtig dieser zweistufige Schritt ist, zeigen auch neuere Arbeiten. [Binder 02] verdeutlicht, dass bereits beim Instanziieren neuer Klassen Code des Agenten ausgeführt werden kann.

4.8.3 WRAP: WILMA Remote Administration Protocol

Zur Administration der Plattform selbst und ihrer Agenten kommt Pia, der Platform Interface Agent, zum Einsatz. Dieser Agent wird vom Pia-Client über das WILMA Remote Administration Protocol WRAP kontrolliert; Abbildung 4.8 *WRAP: Beteiligte Instanzen*, Seite 69 zeigt die Modularisierung.

a) Bestandteile

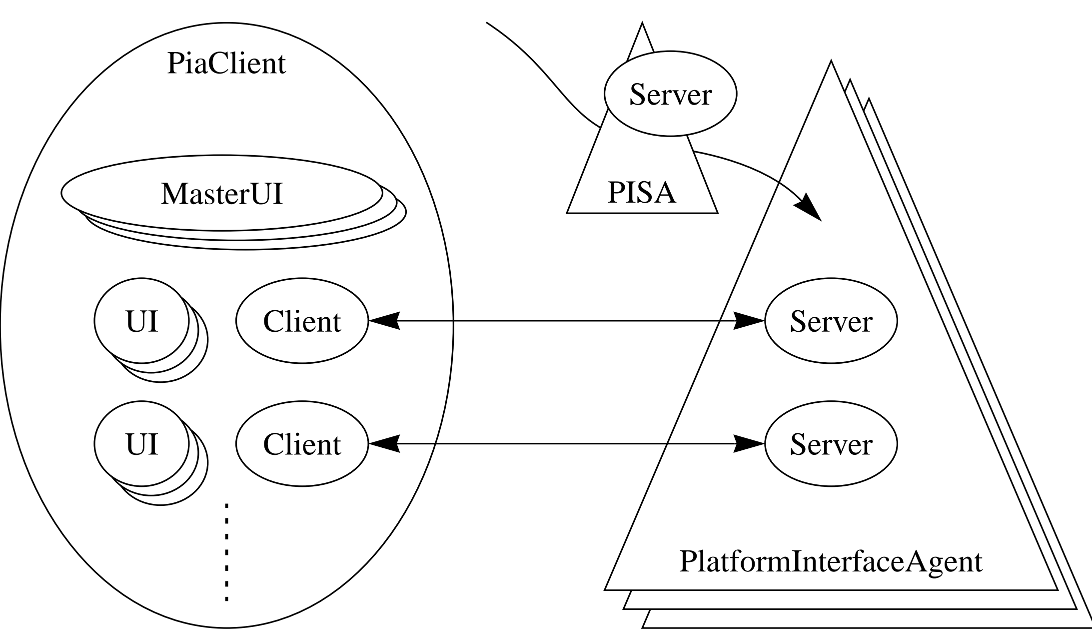

Abbildung 4.8 *WRAP: Beteiligte Instanzen*

Das Master-UI stellt den Rahmen für die einzelnen User-Interfaces (UI) dar. Es bietet dem Benutzer die Auswahl der darzustellenden UIs. Es gibt mehrere, alternative Master-UIs (derzeit Text-UI und AWT-GUI). Ein UI fragt Parameter vom Benutzer ab und übergibt diese an den Client. Auch stellt das UI die Ergebnisse des Clients dar, soweit vorhanden. Jeder Client hat für jedes Master-UI jeweils ein passendes UI.

Die Clients verarbeiteten die Parameter des UIs und kommunizieren mit einem oder mehreren Servermodulen. Diese können sich auf verschiedenen Plattformen befinden. Der Server führt die Aktionen dann innerhalb der Plattform aus.

Pia-Client und Platform Interface Agent dienen nur als Container mit möglichst wenig eigener Funktionalität; die eigentliche Logik steckt in den Modulen (Clients und Server). Ein weiterer Agent (Pia Service Agent, PISA) installiert die Server-Module.

b) Protokoll

Der Pia-Client kommuniziert mit dem Platform Interface Agent. Allerdings gibt der die Kontrolle über die Verbindung an die Hilfsklasse `PiaAction` weiter.

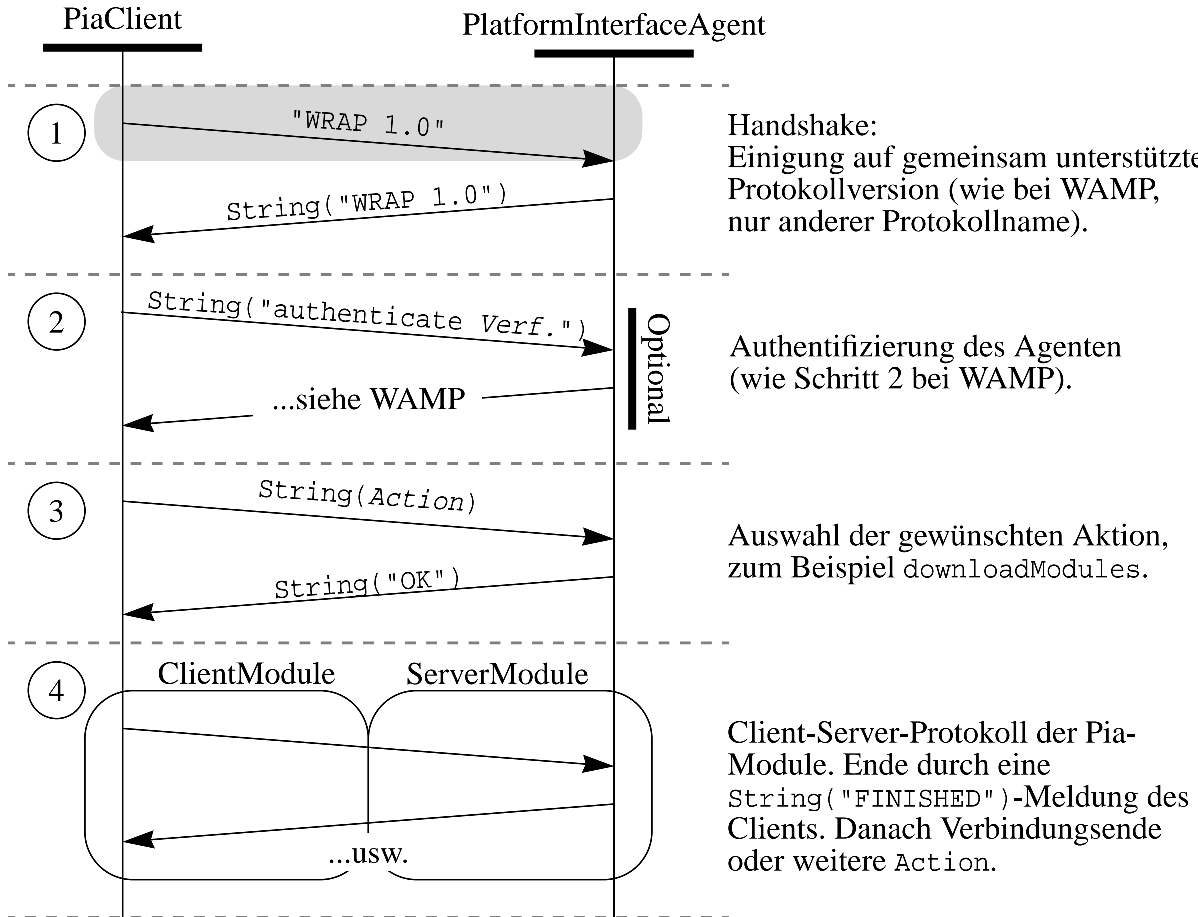

Abbildung 4.9 *WRAP-Protokoll*

c) Sicherheitsüberlegungen

- Anders als bei WAMP ist hier der Benutzer auch direkt der Kommunikationspartner (keine Unterscheidung Plattform/Agent).
- Die Rechte der Server-Module werden aus den Rechten des Benutzers abgeleitet. Jedes Modul überprüft, wenn nötig, ob der Benutzer auch über die jeweils nötigen Rechte verfügt. Um Module überhaupt starten zu dürfen, benötigt ein User das Recht `PiaUser` (siehe 6.3 *Rechte*, Seite 105).

4.8.4 RAMP: Remote Agent Messaging Protocol

a) Protokoll

Durch das Remote Agent Messaging Protocol RAMP können MobMan-Agenten auch über Plattformgrenzen hinweg miteinander kommunizieren. Wie in 4.7 *Kommunikation*, Seite 57 erklärt, dient zur Adressierung in diesem Fall die Klasse `GlobalAgentID`.

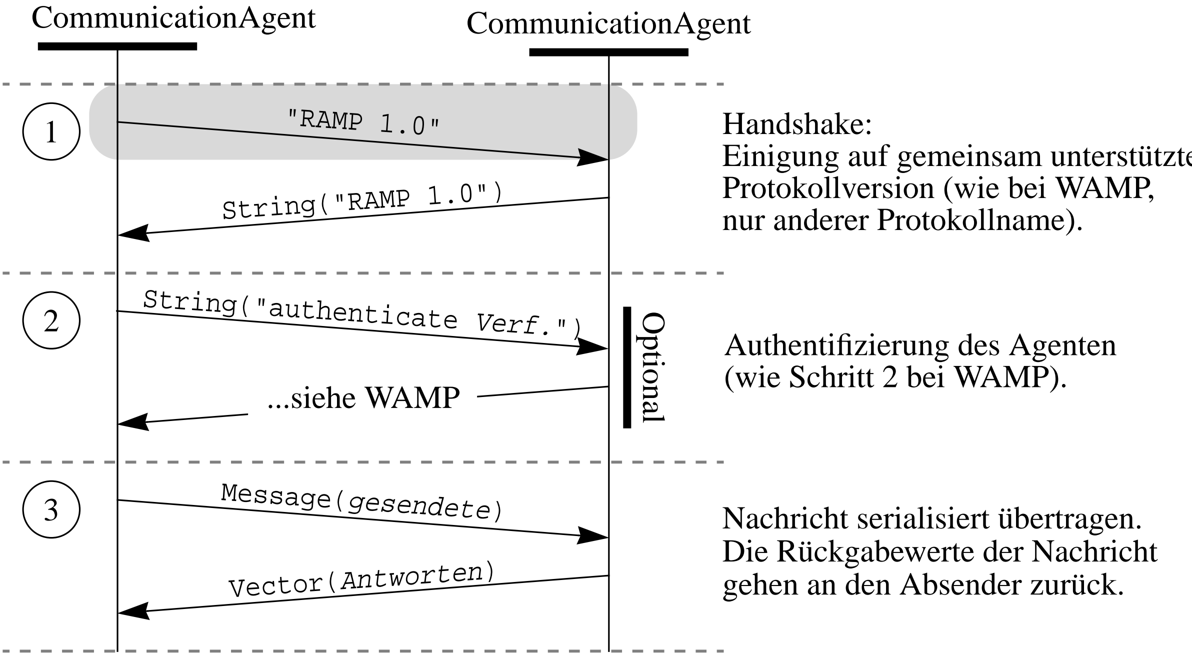

Abbildung 4.10 *RAMP-Protokoll*

Auf der Absenderseite erkennt der Communication Agent, dass das Ziel auf einer anderen Plattform liegt (durch die Zielangabe im `Message`-Objekt). Er verbindet sich mit der Ziel-Plattform, authentifiziert sich und reicht die Nachricht (als serialisiertes `Message`-Objekt) an den Empfänger-Communication-Agent weiter. Dieser gibt die Nachricht an den Empfänger-Agenten weiter. Gibt es mehrere gleichnamige Agenten, erhalten alle die Nachricht, wie bei lokalen Messages. Durch die optionale Identifikationsnummer kann der Absender auch explizit einen Agenten auswählen. Die Antworten gehen als serialisierter Vektor zurück an den Absender.

b) Sicherheitsüberlegungen

- Beide Plattformen müssen sich gegenseitig vertrauen, da nur so eine korrekte Zustellung gewährleistet werden kann.
- Innerhalb der Plattform geschieht die Zustellung wie bei lokaler Kommunikation, es gelten daher identische Sicherheitsüberlegungen.
- Die Klassen aller Objekte, die in einer Nachricht enthalten sind, müssen auf der Zielplattform bereits bekannt sein. Beispielsweise bei Klassen aus `MobMan.Core` ist dies generell der Fall. Da der Empfänger die Klasse auch verarbeiten können muss, ist eine Übertragung der Klassen während der Kommunikation nicht sinnvoll. Der Absender muss geeignete Klassen verwenden, die der Empfänger auch kennt.

4.8.5 Beurteilung der Protokolle

Ziel der MobMan-Protokolle war, bei einfacher Implementierung eine hohe Flexibilität zu ermöglichen. Die Flexibilität und Erweiterbarkeit wird vor allem durch die mehrschichtige Architektur und das Handshake-Verfahren ermöglicht. Unter Beibehaltung der Kompatibilität zu HTTP erlaubt MAP, weitere Protokolle hinzuzufügen. Durch die Versionsangabe im MAP-Header ist es auch möglich, verschiedene Varianten der Protokolle zu unterscheiden und kompatibel zu implementieren. Das Handshake-Verfahren erlaubt es, ältere Programmversionen unverändert zusammen mit neueren betreiben zu können.

Die Eigenschaften von Java und von TCP ermöglichen es, die Implementierung einfach zu halten. TCP sorgt für den zuverlässigen Transportdienst, so dass keine Sicherung innerhalb der Anwendungsschicht mehr nötig ist. Durch das Object Streaming von Java wird die Plattformunabhängigkeit sichergestellt: das Marshalling (systemunabhängige Darstellung der Daten) übernehmen die Standardmechanismen der Sprache. Die Sprache stellt auch sicher, dass der Empfänger die Daten nur innerhalb der zulässigen Typumwandlungen interpretiert. Andernfalls treten Exceptions auf, die einen Fehlererkennung und -Behebung ermöglichen.

Auch bei Verbindungsabbrüchen sorgen Exceptions für die Fehlererkennung. Die Implementierung war damit sehr klar und übersichtlich möglich, ohne aufwändige Zustandsautomaten. Die Protokolle konnten in vielen MobMan-Anwendungen und -Tests ihre Funktionsfähigkeit unter Beweis stellen.

Ähnliche Protokolle waren auch schon in frühen Versionen von MobMan implementiert. Neben der Integration von Authentifizierung und weiteren Sicherheitsmechanismen (SSL/TLS) wurde im Rahmen dieser Arbeit die Mehrschicht-Architektur und der RAMP-Mechanismus entworfen. Bei der Authentifizierung wurde Wert darauf gelegt, ein einheitliches Verfahren einzusetzen: Der selbe Mechanismus wird auch innerhalb einer Plattform angewendet, siehe 6.2 *Authentifizierung*, Seite 95. Diese Wiederverwendung ist durch den Protokollentwurf als Java-Objektstrom möglich.

Kapitel 5

Java-Spezifika

Mobile-Agenten-Systeme stellen eine Reihe von Anforderungen an ihre Laufzeitumgebung. Beispiele sind die Kapselung der Agenten und die sichere Identifizierung der Kommunikationspartner. Einen Teil der notwendigen Dienste stellt die Java-Laufzeitumgebung bereits zur Verfügung. Allerdings sind deren Sicherheitseigenschaften für mobile Agenten nicht ausreichend.

Dieses Kapitel beginnt mit dem Sicherheitsmodell von Java und zeigt die Einschränkungen im gegebenen Einsatzfeld auf. Eine Reihe sicherheitstechnisch bedenklicher Eigenschaften der Sprache sind auch in [Binder 02] beschrieben. Kapitel 5.3 bis 5.6 zeigen mögliche Lösungen für die Sicherheitsprobleme, die für MobMan relevant sind. Zum Abschluss wird der in MobMan genutzte Java Security Manager vorgestellt.

5.1 Java-Sicherheitsmodell

Das Kapitel beginnt mit wesentlichen Sprachkonzepten, die die Sicherheitsarchitektur von MobMan beeinflussen. Alle Angabe beziehen sich auf JDK 1.1 (Java Development Kit), da diese Version die maximale Verbreitung von mobilen Agenten erlaubt. Neuere Java-Versionen können alten Code weiterhin ausführen. Würde MobMan aber eine aktuelle Version voraussetzen, wären viele ältere JVMs (Java Virtual Machine) nicht mehr geeignet.

Das Sicherheitsmodell von Java ist darauf ausgerichtet, den Anwender (genauer seinen Host) vor nicht vertrauenswürdigem Code zu schützen. Ein typischer Anwendungsfall sind Applets, die aus einer beliebiger Quelle stammen und auf dem Host des Anwenders ausgeführt werden (COD, Code on Demand). Diese Applets sollen keinen unkontrollierten Zugriff auf den Host erlangen können [Venners 97]. Vereinfacht dargestellt unterbindet die JVM folgende Aktionen:

- Lokale Dateien lesen oder schreiben.
- Netzwerkverbindungen (nur zu dem Host erlaubt, von dem das Applet stammt).

- Neue Prozesse starten.
- Neue Bibliotheken laden.
- Native Methoden aufrufen (Maschinencode).

Java selbst ist, wie jedes größere Softwaresystem, nicht frei von Fehlern. Die Vergangenheit hat gezeigt, dass durch Implementierungsfehler auch in Java Sicherheitslücken auftreten können [Dean 96]. Diese Lücken werden aber durch korrigierte Software-Versionen (Patches) rasch geschlossen und sind nicht grundsätzlicher Natur.

5.1.1 Überblick

Beim Ausführen von Java-Programmen finden an mehreren Stellen Sicherheitsüberprüfungen statt. Abbildung 5.1 *Schritte bei der Ausführung von Java-Applets*, Seite 74 skizziert den Ablauf von der Erzeugung bis zur Ausführung. Der Java-Compiler übersetzt den Quellcode in Bytecode. Bereits an dieser Stelle finden viele Überprüfungen statt, etwa ob die Kapselung von Objekten eingehalten wird.

Im Prinzip ist es möglich, Java-Programme unmittelbar als Bytecode zu programmieren, ohne Java-Quellcode und einen Compiler zu benutzen. Die Überprüfungen des Compilers sind damit leicht zu umgehen. Sie sind nur als Hilfe für den Entwickler zu sehen und nicht als Bestandteil der Sicherheitsarchitektur geeignet. Der Bytecode wird daher vom Classloader und dem Bytecode Verifier zusätzlich überprüft, bevor die JVM die Programmteile ausführt. Während der Ausführung auf der JVM überprüft der Security Manager die sicherheitskritischen Methoden. Erst nach seiner Zustimmung gibt Java die Kommandos an das Betriebssystem weiter.

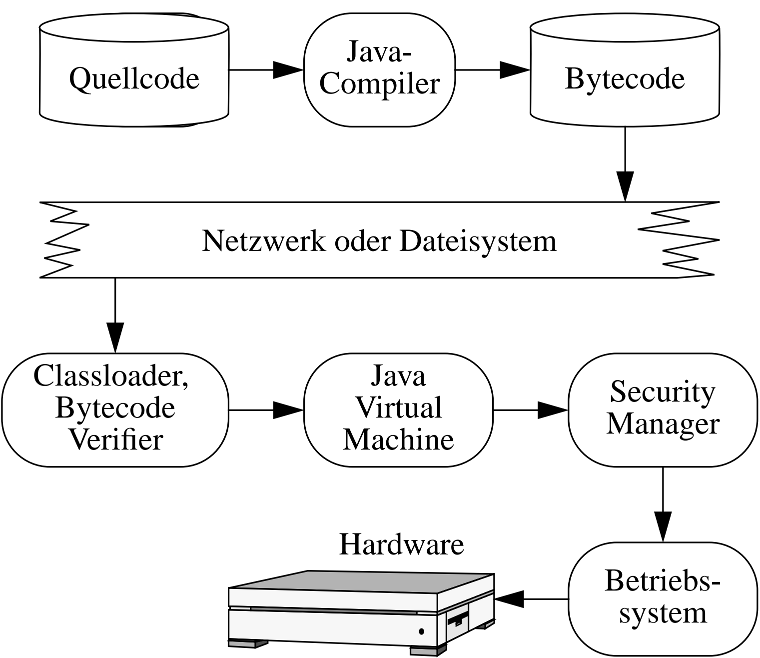

Abbildung 5.1 *Schritte bei der Ausführung von Java-Applets*

5.1.2 Die Sandbox

In klassischen Computer-Anwendungen gilt der ausgeführte Code in der Regel als vertrauenswürdig. Jedes Programm erhält alle Rechte seines Besitzers (derjenige, der das Programm ausführt), es gibt keine weitere Unterscheidung. Unter diesen Randbedingungen muss der Anwender jedem Programm, das er startet, vertrauen. Capability-basierte Systeme stellen hier eine Ausnahme dar.

Um unbekannten, fremden und damit nicht vertrauenswürdigen Code gefahrlos ausführen zu können, wurde in Java das Sandbox-Konzept eingeführt. Dabei besitzt die Laufzeitumgebung zwar die vom System vorgegebenen Rechte, schränkt aber selbst die Rechte des von ihr ausgeführten (interpretierten) Programms ein (Abbildung 5.2 *Die Java-Sandbox*, Seite 75). Klassen, die von der lokalen Installation stammen, unterliegen nicht diesen Kontrollen.

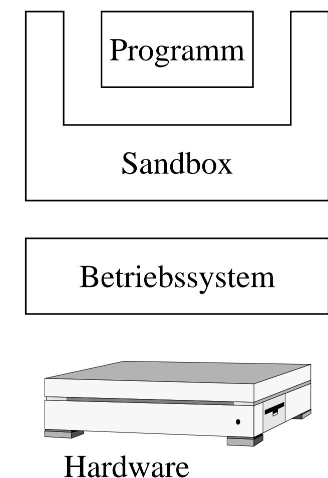

Abbildung 5.2 *Die Java-Sandbox*

In späteren Versionen des JDK wurde die einfache Unterscheidung in vertrauenswürdigen, lokalen Code und nicht vertrauenswürdigen, über das Netz geladenen Code feiner aufgegliedert. Je nach Klasse und Quelle des Codes können in Java 2 (JDK 1.2) unterschiedliche Sicherheits-Policies eingesetzt werden [Gong 97], [Gong 98b], [Falk 99]. Es ist aber keine Unterscheidung der Objekte vorgesehen (siehe 5.6.1 *Problem der Identifikation des Aufrufers einer Methode*, Seite 83).

5.1.3 Schritte der Sicherheitsprüfung

Wie bereits skizziert, finden bei der Ausführung von Java-Code an mehreren Stellen sicherheitsrelevante Überprüfungen statt, die zum Teil aufeinander aufbauen. Die einzelnen Schritte werden im Folgenden etwas genauer beleuchtet.

a) Speicherverwaltung zur Laufzeit

Bei Java erfolgt die Speicherzuweisungen zur Laufzeit, sie ist nicht vom Compiler vorbestimmt. Die konkrete Umsetzung hängt von der jeweiligen Plattform ab, Java-Programme können daher keine Zeiger verwenden. Die Speicherplätze werden mittels symbolischer Handles referenziert, die der Java-Interpreter erst zur Laufzeit in reale Speicheradressen umsetzt.

b) Security Manager

Der Security Manager wird von der Anwendung implementiert und sorgt für die Einhaltung der Sicherheitsregeln. Nach dem Start der JVM kann nur ein einziger Security Manager eingerichtet werden. Diese Klasse prüft, ob eine bestimmte Operation in der aktuellen Umgebung erlaubt ist. Alle Methoden beliebiger Klassen, die eine potenziell gefährliche Operation durchführen (Zugriff auf das Dateisystem oder ähnliches), fragen selbst den Security Manager, ob die jeweilige Aktion zugelassen ist.

Dem Security Manager stehen für die Entscheidung folgende Informationen zur Verfügung:

- Klassen und Methoden aus dem Callstack, jedoch keine Objektreferenzen.
- Quelle dieser Klassen (lokal/remote, in späteren Java-Versionen feiner untergliedert).
- Welcher Zugriff worauf.

c) Classloader

Der Classloader definiert ein bestimmtes Protokoll, mit dessen Hilfe Klassen zur Laufzeit geladen werden können [McManis 96]. Klassen, die aus nicht vertrauenswürdiger Quelle stammen, existieren in einem eigenen Namensraum, abgegrenzt von den Klassen des lokalen Systems. Diese Trennung verhindert Namenskonflikte und vereitelt den Versuch, Standardklassen durch fremde zu ersetzen. Der Code wird vor seinem Einsatz an den Bytecode Verifier zur Überprüfung weitergeleitet.

d) Bytecode Verifier

Der Bytecode Verifier überprüft [Yellin 97] unter anderem, ob

- es sich bei dem Code um erlaubten Java-Bytecode handelt,
- die Gefahr eines Stack-Überlaufs vermieden wird und
- Datentypen nicht illegal umgewandelt werden.

Dadurch sollen das Fälschen von Referenzen und das Ausführen sicherheitsgefährdender Speicherarithmetik verhindert werden. Außerdem soll dies verhindern, dass schädlicher Code den Interpreter in einen undefinierten Zustand versetzt. Dies könnte Sicherheitslücken zur Folge haben. Da die Prüfungen komplex und somit schwer nachvollziehbar sind, unternehmen einige Arbeiten [Goldberg 98] den Versuch, Classloader und Bytecode Verifier mit einer mathematischen Spezifikation zu beschreiben und formal zu verifizieren.

e) Packages

Anders als in C++ können sich in Java die Klassen innerhalb eines Packages gegenseitig zusätzliche Rechte geben. Könnte sich eine Klasse als Teil eines Packages ausgeben, zu dem es nicht wirklich gehört, würde sie zusätzliche Rechte erlangen. Ein Classloader wird daher Klassen aus nicht vertrauenswürdigen Quellen nicht zulassen, wenn sie sich als Teil der Java-API ausgeben. Bei MobMan kommen zu dieser Verbotsliste noch Klassen aus `MobMan.Core.*` hinzu, sie dürfen nicht als Teil eines Agenten auf die Plattform gelangen.

f) Kapselung und Sichtbarkeit

Java kennt folgende Zugriffsrechte für Methoden und Felder:

- private
- protected
- public
- none (package)

Tabelle 5.1 *Kapselung*, Seite 77 zeigt, welche Klassen mit welchen Zugriffsrechten die jeweiligen Felder und Methoden benutzen dürfen. Zusätzlich können Methoden und Felder als `final` deklariert sein. Dies kennzeichnet den Entwicklungsstand als endgültig. Final-Methoden sind nicht überschreibbar, also nicht virtuell.

	Zugriffsrechte			
Benutzer der Methoden und Felder	public	protected	package	private
Die selbe Klasse	ja	ja	ja	ja
Beliebige andere Klasse im selben Paket	ja	ja	ja	nein
Subklasse (Erbe) aus anderem Paket	ja	ja	nein	nein
Fremde Klasse (kein Erbe), anderes Paket	ja	nein	nein	nein

Tabelle 5.1 *Kapselung*

5.2 Beschränkungen von Java

Die JVM an sich ist zwar geeignet, mehrere unabhängige Applikationen parallel auszuführen. Allerdings geht das JDK an einigen Stellen implizit davon aus, dass immer nur eine einzelne Anwendung innerhalb der JVM läuft [Balfanz 97]. Die Autoren vermissen mehrere notwendige Eigenschaften und Fähigkeiten:

1. Applikationen innerhalb der JVM starten und beenden.
2. Parallel laufende Anwendungen sollen unterscheidbar sind.
3. Java-Code nicht nur anhand seines Ursprungs kennzeichnen, sondern ihm einen User zuordnen, der ihn ausführt.
4. Log-In in die JVM.
5. Codebase-basierende Sicherheits-Policies mit User-orientierten Policies kombinieren.
6. Systemcode, der mehrere parallele Applikationen unterstützt.
7. Event-Dispatcher, der mehrere parallele Applikationen unterstützt.
8. Unterscheidung des Zustands einer Applikation und des Gesamtsystems.
9. JVM-weite Sicherheits-Policies und applikationsbezogene Policies kombinieren.

Diese Einschränkungen treffen in abgewandelter Form auch auf mobile Agenten zu. Beispielsweise entsprechen Login und der Start von Applikationen (4. und 1.) der Migration.

Java enthält keinen Mechanismus, der den aktuellen Zustand eines Threads sichern und wieder herstellen könnte [Bryce 99a]. Dies lässt sich nur durch Modifikationen an der JVM erreichen [Bouchenak 99], [Bouchenak 00] – allerdings wurde diese Option in 4.1 *Designkriterien*, Seite 38 explizit ausgeschlossen. Eine Alternative besteht in instrumentiertem Code, der während der Ausführung den Zustand des Threads laufend sichert. Dies beeinträchtigt allerdings die Performance stark [Bouchenak 99], [Fünfrocken 98c] und kommt daher für MobMan nicht in Frage.

In den meisten Fällen ist starke Migration nicht nötigt. Die Authentifizierung und unterschiedliche Rechte und Möglichkeiten auf verschiedenen Plattformen heben die Eleganz der starken Migration wieder auf: Der Agent muss sich in jedem Fall auf neue Randbedingungen einstellen.

5.3 Trennung der Klassen

Aus Sicherheitsgründen ist die Trennung der Agenten eine wesentliche Notwendigkeit eines Agentensystems [Bryce 99b]. Dies beginnt bereits bei der Trennung der beteiligten Klassen: Typische Ad-hoc-Verfahren zur Vermeidung von Namenskollisionen greifen bei mobilen Agenten nicht. In klassischen Applikationen weiß der Autor, welche Namen er für Klassen, Pakete und Ähnliches verwendet. Er ist letztlich dafür zuständig, Überschneidungen von Namen (etwa in Bibliotheken) zu verhindern.

Bei Agentensystemen sind viele voneinander unabhängige Autoren beteiligt, die Agenten aus unterschiedlichen Quellen sollen aber in einer JVM coexistieren können. Die bei Bibliotheken verwendeten Präfixe und Package-Namen helfen zwar, Überschneidungen unwahrscheinlicher zu machen, sie sind aber keine Garantie. Jeder Autor kann beliebige Namen verwenden, außerdem können verschiedene Versionen eines Agenten oder eines seiner Bestandteile gleichzeitig im Umlauf sein [Fischer 99]. Es ist daher eine zusätzliche Ebene zur Unterscheidung nötig. Ähnliche Situationen treten auch bei Komponentenarchitekturen auf [Hawblitzel 98].

Sicherheitsprobleme bei Namenskollisionen sind beispielsweise:

- *Unterschieben einer falschen Implementierung*
  Der angreifende Code würde alle Rechte erhalten, die der angegriffene Agent hatte.
- *Auslesen von geschützten Daten*
  Auch bereits gesammelte Daten, die von anderen Agenten nicht gelesen werden können, befinden sich im Zugriff dieses Codes.

Eine Reihe von Ansätzen ist im Prinzip geeignet, diese Überschneidungen zu verhindern. Jeder hat aber weitere Seiteneffekte, die es abzuwägen gilt.

5.3.1 Getrennte Laufzeitumgebungen

Der konsequenteste Ansatz ist, jedem Agenten eine eigene Laufzeitumgebung innerhalb der verwendeten Programmiersprache zuzuteilen. Diese Umgebungen können eigene Prozesse oder mehrere Interpreter in einem Prozess sein. Das Betriebssystem oder die Programmiersprache sorgen für die Trennung.

Vorteile:

- Vollständige Trennung der Agenten.
- Gut überblickbar.

Nachteile:

- Kommunikation sehr aufwändig, je nach Variante als Interprozesskommunikation oder als Nachrichten zwischen getrennten Interpretern.
- Skaliert schlecht, hoher Aufwand pro Agent.

5.3.2 Verschiedene Classloader

Jede Klasse wird innerhalb der JVM durch ein Objekt vom Typ `Class` dargestellt. Durch das Anlegen neuer `Class`-Objekte können in Java dynamisch zur Laufzeit neue Klassen eingeführt werden, für das Laden der Klassen ist ein Classloader zuständig. Der Name einer Klasse ist nur für das Auffinden der Klasse nötig. Ob zwei Klassen identisch sind, entscheidet nur die Referenz auf das `Class`-Objekt. Somit können mehrere gleichnamige Klassen gleichzeitig existieren, die aber einen anderen Typ darstellen und auch inhaltlich grundverschieden sein können.

Üblicherweise hält jeder Classloader einen eigenen Cache von bereits geladenen Klassen. Dieser Cache stellt sicher, dass für gleichnamige Klassen immer die selbe `Class`-Referenz zurückgeliefert wird. Dies ist auch notwendig, da nur so einzelne Teile eines Systems Objekte dieser neuen Klasse gemeinsam nutzen können. Wird ein anderer Classloader für das Laden einer gleichnamigen Klasse benutzt, dann kennt dieser den alten Cache nicht und legt ein neues `Class`-Objekt an. Somit hat die neue Klasse zwar den gleichen Namen, wird aber über eine andere Referenz angesprochen und gilt als anderer Typ [McManis 96]. Dieser Zusammenhang lässt sich für eine gute Trennung der Agenten nutzen, indem jeder Agent mit allen seinen Klassen über einen eigenen Classloader geladen wird [Balfanz 97], [Busse 98].

Vorteile:

- Vollständige Trennung der Namensräume.

Nachteile:

- Kommunikation ist nicht über Klassen möglich, die von den Agenten selbst stammen, da jeder Agent unter dem gleichen Namen einen anderen Typ versteht.
- Skaliert schlecht, da identische Klassen mehrfach geladen werden.

5.3.3 Verschiedene Classloader bei Erkennung identischer Klassen

Die Nachteile der vorherigen Lösung betreffen vor allem identische Klassen, die in der Folge mehrfach vorhanden sind. Die aktuelle Fassung von MobMan vermeidet dies, indem sie zwar jedem Agenten einen eigenen Classloader zuordnet, über diesen aber nur die Klassen lädt, die vorher noch nicht im Agentensystem vorhanden waren. Bei identischen Klassen wird die bereits geladene Klasse verwendet.

Ob zwei gleichnamige Klassen wirklich identisch sind, ließe sich durch einen Byte-weisen Vergleich des Bytecodes feststellen. MobMan beschleunigt dies und vergleicht die kryptografisch starken MD5-Hashes beider Bytecodes.

Vorteile:

- Identische Klassen werden nur einmal geladen.
- Kommunikation zwischen Agenten über eigene, mitgebrachte Klassen, ist möglich.

Nachteile:

- Statische Klassenvariablen werden auch zwischen Agenten geteilt.
- Probleme mit Package Visibility: Klassen, die bereits von einem anderen Agenten geladen wurden, befinden sich in dessen Namensraum. Damit werden sie einem anderen Package zugeordnet als neue Klassen des neuen Agenten. Das Paket trägt zwar den gleichen Namen, ist aber doch nicht identisch.

Für die Package Visibility kommt neben dem Namen des Pakets zum Tragen, im welchem Namensraum die Klasse geladen wurde. Dieser Namensraum wird identifiziert durch die Referenz auf den Classloader, der die Klasse geladen hat [Venners 97]. Nur wenn diese Referenzen identisch sind, werden gleichnamige Packages als identisch angesehen und Package-bezogene Zugriffsrechte wie erwartet erteilt.

So lange eine Klasse nur von einem einzelnen Agenten benutzt wird, treten bei dieser Form der Klassentrennung keine Probleme auf. Wenn ein Agent eine Klasse benutzt, die sich im selben Package befindet wie er selbst und er die Zugriffsregeln der Package Visibility nutzt, kann es zu Problemen kommen: Nutzt bereits ein anderer Agent dieselbe Klasse, dann wurde diese vom Classloader des anderen Agenten geladen und befindet sich aus Sicht der JVM damit in einem anderen Package. Folglich werden die Zugriffsrechte nicht wie erwartet zugeteilt.

Ein Ausweg könnte sei, auf diese Zugriffsregeln zu verzichten. Dies kann aber nicht die Ultima Ratio sein und ist vor allem bei vorgefertigten Archiven nicht möglich. Etwas Linderung bietet das folgende Verfahren.

5.3.4 Sonderbehandlung von Archiven

Bislang werden Jar- und Zip-Archive zwar bearbeitet, aber nur als ein weitere Quelle von Klassen, aus denen bei Bedarf die einzelnen Klassen geladen werden. Dafür kommt der Classloader des jeweiligen Agenten zum Einsatz. Man könnte alle in einem Jar- oder Zip-Archiv enthaltenen Klassen stattdessen immer über den selben Classloader laden.

Vorteile:

- Package-Beziehungen innerhalb eines Archivs bleiben erhalten.

Nachteile:

- Identische Klassen, die bereits vorher außerhalb des Archivs benutzt wurden, gelten nicht mehr als identisch.

5.3.5 Sonderbehandlung spezieller Klassen

Im Gegensatz zu den bisherigen Verfahren könnten alle Klassen eines Agenten über seinen eigenen Classloader geladen werden, unabhängig davon, ob sie bereits vorhanden sind. Lediglich Klassen, deren Objekte zwischen Agenten ausgetauscht werden sollen, werden gemeinsam genutzt. Diese Klassen müsste der Agent speziell kennzeichnen. Zusätzlich sollte für diese Klassen das bereits beschriebene MD5-Hash-Verfahren eingesetzt werden.

Vorteile:

- Keine Probleme bei nicht speziell gekennzeichneten Klassen.
- Übersichtlich.

Nachteile:

- Klassen aus Jar- oder Zip-Archiven könnten von einem Agenten auch als übertragbar gekennzeichnet werden, so dass die beschriebenen Probleme immer noch eintreten können (aber nur für Agenten, die diese Klasse als übertragbar kennzeichnen).
- Höherer Aufwand, da identische Klassen mehrfach vorhanden sind. Leider gelingt das Entfernen von Klassen bei vielen VMs in der Praxis nicht, so dass sich immer mehr Klassen ansammeln würden.

5.3.6 Zusammenfassung

Eine ideale Lösung ist ohne Unterstützung der JVM nicht möglich. Es gilt also, die einzelnen Vor- und Nachteile abzuwägen. Die vollständige Trennung wäre aus Gründen der Konsistenz am besten, eignet sich in der Praxis aber kaum. Wünschenswert wäre, auf die Package Visibility zu verzichten. Dies lässt sich für eigens entwickelte Agenten fordern; bei fertigen Bibliotheken ist dies nicht möglich.

5.4 Trennung der Agenten: keine Referenzen

Um zu verhindern, dass zu offen gewählte Zugriffsrechte auf Methoden der Agenten von anderen Agenten missbraucht werden, erhalten User-Agenten keine Referenzen auf andere Agenten [Busse 98]. Dies fügt quasi eine Kapselung der Agenten (siehe Abbildung 5.3 *Kapselung der Agenten*, Seite 81) in Erweiterung zur Kapselung von Klassen ein. In Java ist der Zugriff auf andere Objekte nur über Referenzen möglich, da die Sprache keine Pointer benutzt. Diese Referenzen sind zudem typisiert und erlauben nur den Zugriff auf die von der Klasse freigegebenen Daten und Methoden.

Wie sich aus Kapitel 5.3.3 *Verschiedene Classloader bei Erkennung identischer Klassen*, Seite 79 ergibt, sind in der Praxis häufig `public`-Zugriffsberechtigungen nötig, wo `package` eigentlich erwünscht wäre. Damit gewinnt die Kapselung der Agenten zusätzlich an Bedeutung.

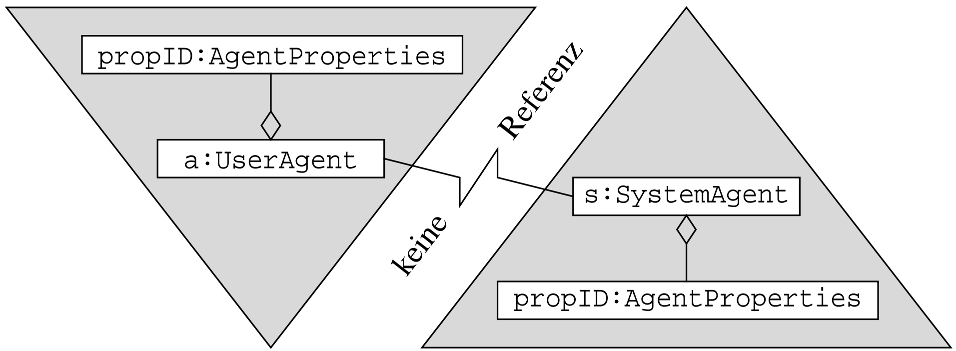

Abbildung 5.3 *Kapselung der Agenten*

Um trotz der Kapselung kommunizieren zu können, wird neben den Messages und Events die `AgentProperties`-Klasse eingesetzt (siehe 5.5 *Agenten und ihre IDs/Agent Properties*, Seite 82). Agenten können im Übrigen eine Referenz auf sich selbst freiwillig weitergeben, wodurch eine enge Kooperation unter Umgehung der relativ aufwändigen Kommunikationsmechanismen von MobMan möglich wird (siehe Designanforderungen). Dies ließe sich zwar verhindern [Hawblitzel 98], ist bei MobMan aber durchaus gewollt.

Damit ist das Verfahren einfacher, flexibler und besser an die Agenten-Problematik angepasst als etwa das Konzept der Guarded Objects [Gong 98a], das nur den Zugriff auf das Objekt an sich regelt und dann die Referenz zurückgibt.

5.5 Agenten und ihre IDs/Agent Properties

Weil Agenten keine Referenz aufeinander erhalten, benötigen sie andere Mechanismen, um sich eindeutig zu identifizieren. Um dies zu gewährleisten, benutzt MobMan IDs als Kennzeichnung. Diese müssen unfälschbar und eindeutig sein – genau diese Eigenschaft besitzen Java-Referenzen. Für jedes zu identifizierende Objekt wird ein zusätzliches Objekt instanziiert, die Referenz auf dieses zweite Objekt dient als unfälschbares Kennzeichen.

Durch ihre Eigenschaft sind IDs auch als Shared Secret geeignet, das die Plattform den privilegierten Systemagenten mitteilt. Sie sind nicht fälschbar, auch nicht durch Brute-Force-Verfahren. Dies ist möglich, da die IDs nur innerhalb der JVM gültig sind und die JVM selbst sicherstellt, dass jede Referenz eindeutig ist.

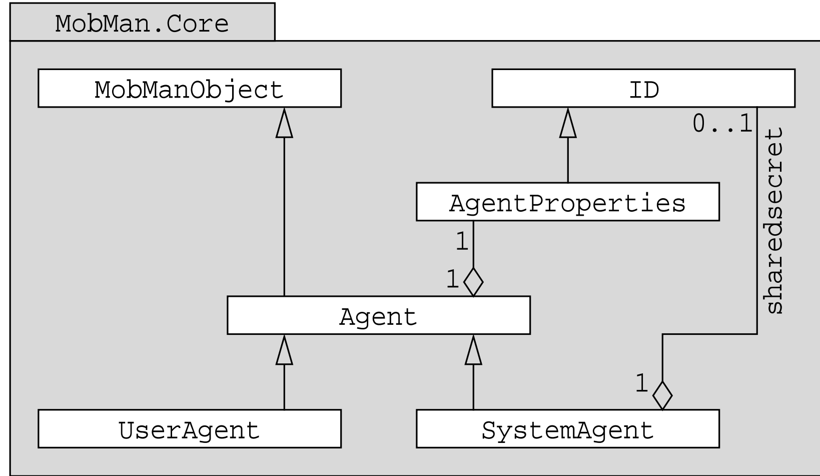

Abbildung 5.4 *Agenten und IDs*

Die Klasse `ID` erbt von der Java-Klasse `Object`, fügt ihr aber keine wesentliche eigene Funktionalität hinzu (nur einen Zähler, der für `GlobalAgentID` benötigt wird, siehe 4.7.5 *Realisierung der Nachrichten in MobMan*, Seite 62). Nur die Referenz dient als Kennzeichen, nicht der Inhalt des `ID`-Objekts. Die `ID`-Klasse dient so als Ersatz für numerische Geheimnisse oder Kennungen. Eine neue ID wird durch Instanziieren eines neuen ID-Objekts erzeugt. Hierdurch ist sichergestellt, dass es zu keinen Überschneidungen oder Mehrfachverwendungen kommen kann.

Eine Erweiterung von `ID` ist die `AgentProperties`-Klasse. Neben der Funktion als ID stellt diese Klasse einen Zugriff auf die Eigenschaften (Methode `getProperty`) eines Agenten zur Verfügung. Durch die Vererbung können `AgentProperties` als Ersatz für `ID` eingesetzt werden. Ein `AgentProperties`-Objekt ersetzt für viele Anwendungen direkte Referenzen auf Agenten: Mit ihrer Hilfe können Agenten ohne Umweg über das Messaging öffentliche Informationen eines anderen Agenten auslesen. `AgentProperties` stellen auch semantisch eine Erweiterung der ID dar: ein Agent wird auch durch seine Eigenschaften gekennzeichnet.

Die Informationen werden nicht direkt in das `AgentProperties`-Objekt kopiert, sondern zur Laufzeit vom Agenten abgefragt (siehe 4.5.3 *Properties*, Seite 49). Hierfür enthält diese Klasse eine private, nicht auslesbare Referenz auf den Agenten.

Sicherheitsüberlegungen:

- Möglicher Angriff: `AgentProperties`-Objekts serialisiert übertragen, am neuen Ziel deserialisieren mit einer anderen Klasse, die die Referenz als `public` deklariert. Diese Referenz kann dann zurückgeschickt werden.
- Lösung: Die Referenz ist als `transient` deklariert und wird daher nicht übertragen.
- Zusätzlich ist `AgentProperties` nicht serialisierbar (erfüllt das Interface `Serializable` nicht, und die entsprechenden Methoden sind in der Basisklasse als `final` enthalten).

5.6 Caller-ID

5.6.1 Problem der Identifikation des Aufrufers einer Methode

An vielen Stelle in MobMan besteht die dringende Notwendigkeit, sicher festzustellen, welches Objekt oder welcher Agent eine Methode aufgerufen hat. Dies trifft vor allem beim Versenden von Nachrichten zu (`Message`-Klasse). Hierbei ist nicht die Authentifizierung gemeint, sondern allein die zuverlässige Identifizierung: Eine eindeutige und unfälschbare ID wurde bereits in Kapitel 5.5 *Agenten und ihre IDs/Agent Properties*, Seite 82 vorgestellt. Wer sich hinter diesem Kennzeichen tatsächlich verbirgt, klärt die Authentifizierung, siehe 6.2 *Authentifizierung*, Seite 95. Sie stellt die authentische Zuordnung einer ID zu ihrem Besitzer her und ist pro Agent nur einmal nötig.

Kennt ein Agent die ID desjenigen, der seine Dienste anfordert, dann kann er qualifiziert entscheiden, ob und wie er reagieren will. Aus der ID lassen sich der authentifizierte Benutzer und dessen Rechte ableiten. Die Schwierigkeit ist, sicher festzustellen, woher eine Nachricht stammt oder allgemeiner, wer eine Methode aufgerufen hat.

In Java ist es nur vorgesehen, einzelnen Methoden den Zugriff auf Ressourcen zu erlauben oder zu untersagen (siehe 5.1.2 *Die Sandbox*, Seite 75). Als Entscheidungsgrundlage dienen dem Security Manager im Wesentlichen der Callstack [Wallach 97], [Wallach 98], auf den die `SecurityManager`-Basisklasse zugreifen kann, sowie der Thread, vom dem aus die Methode aufgerufen wurde. Ein Security Manager könnte diese Information zwar auch an andere Klassen weitergeben, allerdings sind in der Darstellung des Callstacks, auf die der Security Manager Zugriff hat, lediglich die Klassen und Methoden verzeichnet, nicht aber die Objekte. Auch Java 2 benutzt nur die Klasse: Ihre Codebase (URL, von der die Klasse geladen wurde, und eventuell der Signierer) bestimmt, welche Rechte zugeteilt werden. Die Zuordnung ist konfigurierbar.

Es soll bei MobMan aber möglich sein, identischen Code im Namen und mit den Rechten unterschiedlicher Besitzer auszuführen. Eine Möglichkeit wäre es nun, jedem Besitzer eine eigene Codebase zuzuordnen, aber dann würden identische Klassen wieder mehrfach geladen werden (siehe 5.3.3 *Verschiedene Classloader bei Erkennung identischer Klassen*, Seite 79). Das widerspricht den Designanforderungen.

5.6.2 Lösungsansätze

Das Identifizierungsproblem lässt sich unter bestimmten Annahmen durch eine Reihe verschiedener Techniken lösen. Keiner der Ansätze bietet aber eine vollständig zufriedenstellende, allgemein gültige Lösung. Bei den gegebenen Rahmenbedingungen (siehe 4.1 *Designkriterien*, Seite 38) verbietet sich eine Modifikation der JVM (Java Virtual Machine), so dass ein Kom-

promiss zu finden ist. Im Folgenden sind eine Reihe von Ansätzen aufgeführt. Detailliert dargestellt sind neben der in MobMan genutzten Variante vielversprechende Lösungen anderer Projekte sowie Ansätze, die früher in MobMan genutzt wurden. Denkbare weitere Varianten werden kurz vorgestellt, im Anhang finden sich dazu weitere Informationen.

Die Identifizierung des Absenders wird primär beim Erzeugen, Versenden und Zustellen von Nachrichten betrachtet. Dies ist bei MobMan der zentrale Kommunikationsmechanismus. Der Empfänger einer Nachricht muss eindeutig feststellen können, wer der Absender war. Da die Plattform keine Agentenreferenz an andere Agenten weitergibt, ist beim allgemeinen Methodenaufruf die Identifizierung kein vorrangiges Ziel, es wäre aber ebenfalls wünschenswert, hierfür eine Lösung zu finden. Die Ansätze werden daher möglichst allgemein betrachtet.

a) Variante „Vertrauen“

In der einfachsten Version trägt der Absender einer Nachricht eine Referenz auf sich selbst in das `Message`-Objekt ein. Diese Referenz identifiziert ihn eindeutig. In aktuellen MobMan-Versionen erhalten die Agenten von der Plattform keine Referenzen auf andere Agenten. Ein potenzieller Fälscher kennt daher keine Referenzen, die er in ein `Message`-Objekt eintragen könnte, er kann somit auch keinen falschen Absender angeben.

Das `Message`-Objekt muss nur noch prüfen, ob der angegebene Absender wirklich ein möglicher Absender ist, oder nur ein Objekt, das der echte Absender selbst erzeugt hat. Es muss auch aus der Agenten-Referenz dessen ID ermitteln, der Empfänger darf nur die ID lesen.

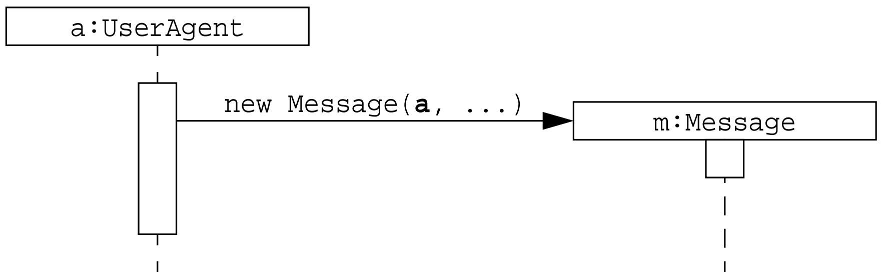

Abbildung 5.5 *Caller-ID-Variante „Vertrauen“*

Vorteile:

- Einfachster Aufbau.
- Einfaches Handling der Message-Objekte.

Nachteile:

- Sobald ein Agent eine Referenz auf sich selbst preisgibt, kann er auch als Absender gefälscht werden. Er kann sich davor nicht mehr schützen.

Fazit:

- Bestechend einfach, aber das (gewollte) Weitergeben von Referenzen hat schwerwiegende Folgen. Jeder Agent, der die Referenz eines anderen kennt, kann in dessen Namen Nachrichten versenden.
- Das Weitergeben der Referenzen soll aber ohne zusätzliche Gefahren möglich sein, dieses Verfahren ist daher nicht geeignet.

b) Variante „Thread Group“

Einen vielversprechenden Ansatz nutzt die Agentenplattform Ajanta [Karnik 98]. In Java sind Threads zu Thread Groups zusammengefasst, die Thread Groups sind hierarchisch angeordnet. Eine Methode kann jederzeit feststellen, in welchem Thread sie gerade ausgeführt wird. Dies macht sich Ajanta zu Nutzen: Jedem Agent ist hier eindeutig eine Thread Group zugeordnet, der Agent wird in einem Thread in dieser Thread Group ausgeführt. Neue Threads dürfen die Agenten nur innerhalb ihrer Gruppe anlegen. Eine aufgerufene Methode kann prüfen, in welcher Thread Group sie läuft, und weiß dann sicher, welcher Agent sie aufgerufen hat. Bei asynchronen Nachrichten (werden nicht im Thread des Absenders zugestellt: `postMessage`) müsste der zusätzlich benötigte Thread auch in der Thread Group des Absenders erzeugt werden.

Abbildung 5.6 *Caller-ID-Variante „Thread Group“*, Seite 85 und die folgenden Abbildungen enthalten eine Mischung aus Sequenzendiagramm (mit Focus of Control) und Klassendiagramm nach UML [OMG 01], um alle wesentlichen Zusammenhänge in einer Grafik darstellen zu können. Zur UML-Darstellung siehe auch Anhang H.2 *Sequenzendiagramme*, Seite 172.

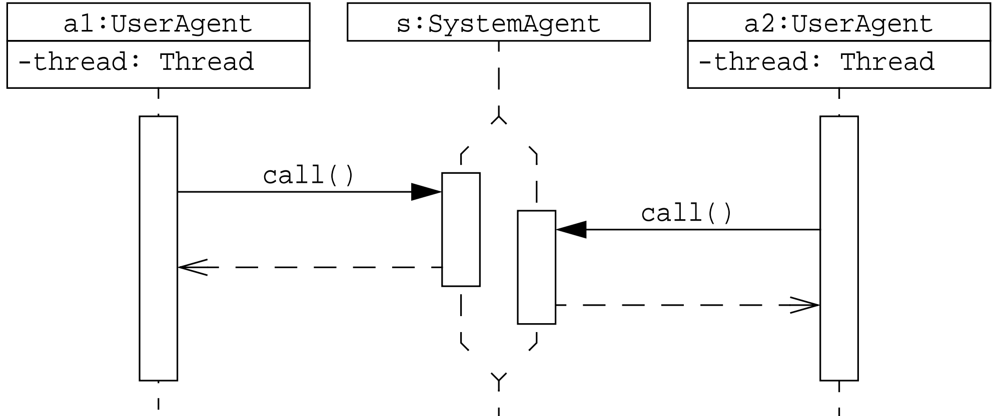

Abbildung 5.6 *Caller-ID-Variante „Thread Group“*

Vorteile:

- Einfach zu implementieren.
- Abfrage an jeder Stelle möglich.

Nachteile:

- Funktioniert nur für aktive Agenten, die in einem eigenen Thread ablaufen.
- Eine gerufene Methoden kann selbst weitere Methoden aufrufen. Diese laufen aber immer noch im selben Thread, obwohl der direkte Aufrufer ein anderer Agent ist.

Fazit:

- Zu starr für MobMan. Hier sind nicht nur aktive Agenten zu identifizieren, sondern auch passive (ohne eigenen Thread) und sogar beliebige Objekte.
- Unsicher, da der Empfänger die Identität des Absenders annehmen kann.
- Nicht geeignet.

c) Variante „Scheduler“

Ein ähnliches Verfahren nutzt die Plattform SAE [Fünfrocken 98a]. Hier kommt ein eigener Scheduler zum Einsatz, der innerhalb der JVM den einzelnen Agenten Rechenzeit zuteilt. Der Security Manager erkennt am Callstack, woher der Aufruf kommt – er kann aber, wie bereits erwähnt, nur die Klasse ermitteln. Um welchen Agent es sich konkret handelt, kann der Security Manager jedoch vom Scheduler erfahren.

Dieses Verfahren hat vergleichbare Nachteile wie die Variante „Thread Group“ und ist zudem komplexer. Nur aktive Agenten erhalten ihre Rechenzeit vom Scheduler, daher erfüllt es ebenfalls nicht die Anforderungen von MobMan.

d) Variante „Rückgriff“

In frühen MobMan-Versionen wurde eine geschützte Message-Klasse genutzt. Diese erhielt den Inhalt der Nachricht (Message Body) nicht über ihren Konstruktor oder über Methodenaufrufe, sondern holte sich die Daten selbst vom Absender. Der Absender musste dafür seine Referenz mitteilen. Mit dieser Referenz konnte die Nachricht auch die ID des Absenders ermittelt.

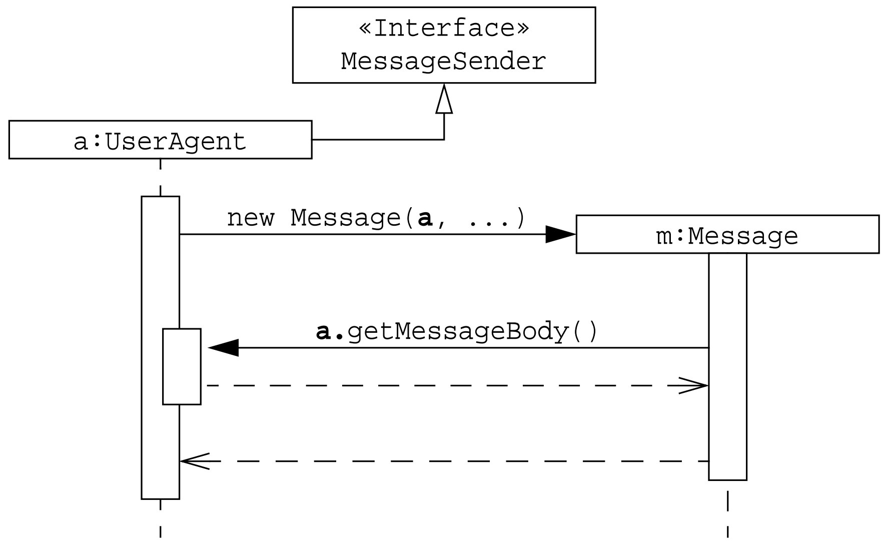

Abbildung 5.7 *Caller-ID-Variante „Rückgriff“*

Vorteile:

- Beliebige Absender-Klasse, die lediglich ein Interface implementieren muss (damals `MessageSender`).
- Auch für passive Agenten geeignet.

Nachteile:

- Nur die Bindung zwischen Absender und Message Body ist eindeutig, Empfänger und zusätzliche Parameter kann jeder, der die Referenz auf die Nachricht besitzt, ändern.
- Über einen gefälschten Absender und einen gefälschten Empfänger kann dem Opfer eine von ihm gerade zum Versenden vorbereitete Nachricht entzogen und fehlgeleitet werden.
- Race Condition: Je nachdem, ob der wirkliche Absender auch gerade eine Message versenden will, treten andere Ergebnisse auf.

Fazit:

- Besser als die bisherigen Varianten.
- Sicherheitsschwächen und wenig flexibel.
- Nicht geeignet.

e) Variante „Message Stub“

Um alle Daten sicher vom wirklichen Absender zu erhalten, muss die `Message`-Klasse das vollständige Nachrichtenobjekt über den Rückgriff vom Agenten abholen. Lösen ließe sich dies durch weitere Felder und Methoden in allen `MessageSender`-Implementierungen. Besser, weil übersichtlicher und einfacher zu erweitern, ist eine `MessageStub`-Klasse. Dieser Stub ist als Aggregation im Agenten enthalten, wird dort vom Agenten mit allen Daten gefüllt und dann von der wirklichen Message-Klasse abgeholt.

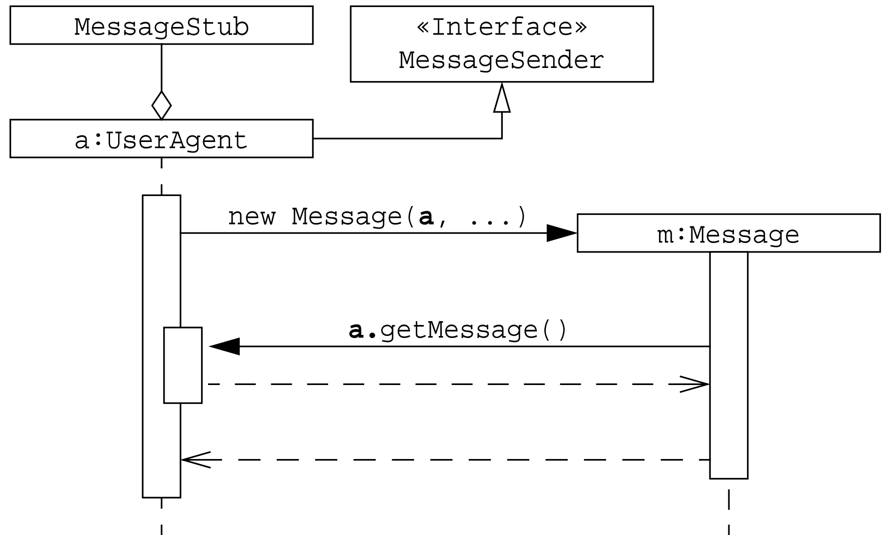

Abbildung 5.8 *Caller-ID-Variante „Message Stub“*

Vorteile:

- Jetzt stammen alle Daten sicher vom Absender.

Nachteile:

- Auch hier existiert noch eine Race Condition, wenn ein Agent einen falschen Absender angibt, der seinen Stub bereits teilweise gefüllt hat.
- Komplex durch zusätzliche Klasse.
- Es können ausschließlich Objekte senden, für die auch die Zuordnung von Objekt-Referenz zu ID bekannt ist – oder die ID wird sicher im `MessageSender` abgelegt. Das wäre aber nur durch eine Basisklasse möglich, nicht durch ein Interface.

Fazit:

- Besser als die bisherigen Lösungen, aber noch nicht ausreichend.

f) Zwischenstufen

Ähnlich der X11-Client-Server-Authentfizierung könnte ein Cookie (beliebige Zufallszahl) zum Nachweis der behaupteten Identität des Absenders dienen, siehe F.1 *Variante „Cookie“*, Seite 163. Dieses Cookie würde ein vertrauenswürdiger Dritter (etwa die Message-Klasse selbst) auswerten. Gegen dieses Verfahren spricht vor allem der relativ hohe Aufwand.

An die Stelle eines Cookies könnte eine private ID treten, die der Agent vor anderen Agenten geheim hält und nur beim Versenden von Nachrichten einsetzt (siehe F.2 *Variante „Private ID“*, Seite 164). Die Plattform würde die Zuordnung der privaten ID zur öffentlich bekannten ID des Agenten vornehmen. Dieses Verfahren glänzt zwar durch seinen klaren Aufbau, nachteilig ist aber auch hier der Aufwand. Es gilt, eine weitere Liste zu verwalten.

Ein deutlicher Vorteil ergibt sich daraus, das gemeinsame Interface `MessageSender` durch eine gemeinsame Basisklasse `MobManObject` abzulösen. In 4.3 *Klassenmodell*, Seite 40 ist dies bereits berücksichtigt. In `MobManObject` ist eine als `final` deklarierte Methode enthalten, die zum Senden von Messages benutzt wird. Nur über diese Methode kann der Agent Nachrichten senden. Da der Agent die Methode nicht manipulieren kann, ist es auf dem Weg möglich, die Original-ID sicher einzutragen. Als Lösungen bieten sich F.3 *Variante „geschützte Methode und Forwarder-Referenz“*, Seite 164 sowie der verbesserte Ansatz F.4 *Variante „geschützte Methode und Shared Secret“*, Seite 165 an. Die eleganteste Variante ist im Folgenden dargestellt.

g) Variante „geschützte Methode und Package Visibility“

In MobMan können Klassen aus dem Paket `MobMan.Core` nicht über das Netz (als Teil eines Agenten) geladen werden. Durch die Platzierung in `MobMan.Core` ist so ein Schutz des Codes verbunden, ein Agent kann die Implementierung nicht manipulieren. Zusammen mit Package Visibility lässt sich dieser Schutz auch für das Absenderproblem nutzen.

Das Absenderfeld in der `Message`-Klasse kann nur über eine Methode mit Package Visibility gesetzt werden. Somit ist sichergestellt, dass nur Klassen aus `MobMan.Core` den Absender setzen. In der `MobMan.Core.MobManObject`-Klasse gibt es eine mit `protected final` geschützte Methode, die den Absender setzt. Die Methode kann somit nur vom eigenen Agenten aufgerufen werden. Sie wird im Namensraum von `MobMan.Core` ausgeführt und hat damit Zugriff auf das Absenderfeld der Message-Klasse.

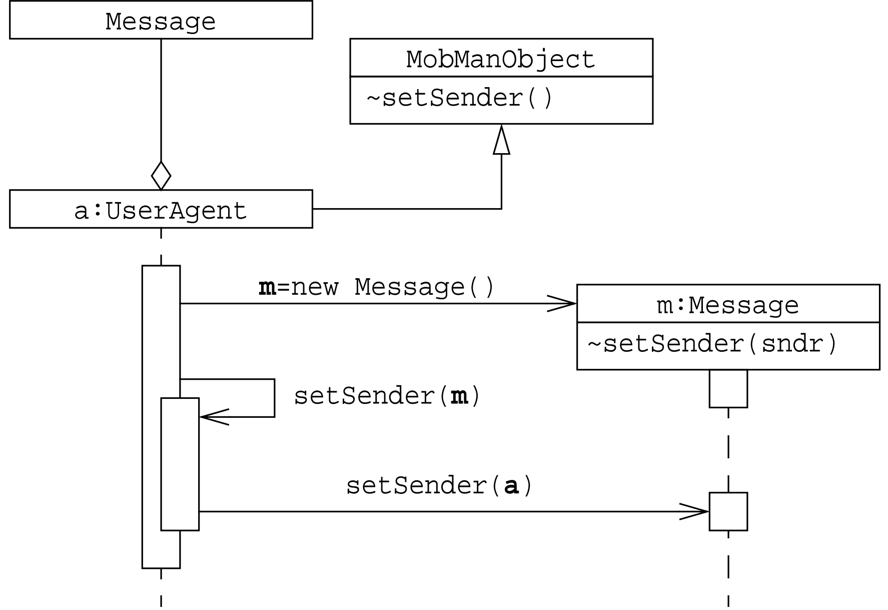

Abbildung 5.9 *Caller-ID-Variante „geschützte Methode und Package Visibility“*

Vorteile:

- Elegant, keine Kunstgriffe wie bei der Variante „Rückgriff“.
- Klarer Ablauf.
- Nachvollziehbar.

Fazit:

- Dieses Verfahren wurde für MobMan ausgewählt.
- Einige Hilfsmethoden in der `MobManObject`-Basisklasse vereinfachen das Versenden.

Die Message-Klasse speichert die Referenz auf den Sender in einem als `private` deklarierten Feld. Als Absender kommen sowohl eine ID als auch die Referenz auf den Absender selbst in Frage. Es gibt keine `getSender`-Methode, lediglich ein `getID`.

Die aktuelle Implementierung enthält in diesem Zusammenhang noch eine Sicherheitslücke: Die Zuordnung der Absenderreferenz zu dessen ID ist über `sender.getID()` gelöst. Diese Methode ist in `MobManObject` implementiert, kann aber von dessen Kindern geerbt werden. Die Default-Implementierung ist leer, da nur Agenten eine eigene ID besitzen, Plugins benutzen die ID ihres Agenten. In den Basisklassen `Agent` und `AgentPlugin` ist die Methode `getID` zwar als `final` deklariert, ein Angreifer könnte eine eigene Klasse aber direkt von `MobManObject` erben lassen, eine eigene `getID`-Methode implementieren und damit die Zuordnung fälschen.

Lösung:

- Zunächst könnte man als Absender auch nur `ID` oder `GlobalAgentID` erlauben, nicht mehr die Referenz zum Objekt oder zum Agenten. Allerdings wird die Referenz genutzt, um den Agenten bei Nachrichten für andere Plattformen zu authentifizieren (über dessen `Authentifier`-Plugin), daher kommt diese Option nicht in Betracht.
- Die `Message`-Klasse müsste lediglich eine weitere Prüfung durchführen, bevor sie `getID()` aufruft. Da diese Methode nur bei Kindklassen von `Agent` und `AgentPlugin` sicher ist, darf sie sie auch nur bei diesen Klassen benutzen (Vergleich mit `instanceof`).

- Als zusätzlichen Schutz könnte man die Methode `getID` aus `MobManObject` entfernen und neben den Typprüfungen auch die entsprechenden Typumwandlungen einsetzen.

5.6.3 Nachrichten an andere Plattformen

MobMan unterstützt auch das Senden von Nachrichten an andere Plattformen (siehe 4.8.4 *RAMP: Remote Agent Messaging Protocol*, Seite 71). Die Identifizierung des Absenders ist dabei nicht mehr mithilfe der Agenten-ID möglich, da diese ID nur innerhalb einer Plattform ihre Gültigkeit hat und nur dort eindeutig ist. Selbst bei der Authentifizierung ist Vorsicht angebracht: Nur wenn die Absenderplattform vertrauenswürdig ist, kann der Empfänger auch davon ausgehen, dass sich wirklich der Absender authentifiziert hat. Der Rückgabewert der Methode `isFromRemote` weist auf diese Einschränkungen hin.

5.7 Security Manager

Eine der Besonderheiten der Sprache Java ist das Sandbox-Modell, das sich unter anderem im Security Manager manifestiert. Wie in 5.1.3 *Schritte der Sicherheitsprüfung*, Seite 75 bereits erläutert, entscheidet diese Klasse bei potenziell gefährlichen Aktionen, ob diese tatsächlich durchgeführt werden dürfen oder nicht. Eine Applikation kann ihren Security Manager selbst wählen, um dann dynamisch Code nachzuladen, für den die Kontrollen des Security Managers bestimmt sind.

5.7.1 Überblick

Der Security Manager kann nur anhand der Klasse des Aufrufers und der Art des Aufrufs entscheiden, ob eine Aktion zulässig ist oder nicht. Bei MobMan ist aber das Objekt entscheidend, verschiedene Agenten können durchaus identische Klassen benutzen. Beispielsweise können zwei identische Agenten von unterschiedlichen Usern gestartet werden und in der Folge verschiedene Rechte besitzen. [Balfanz 97] schlägt bei ähnlichen Randbedingungen vor, mehrere Security Manager einzusetzen. Allerdings erfordert dies, dass die `System`-Klasse mehrfach geladen wird und für jeden Agenten alle seine Klassen neu geladen werden. Nur so lassen sich dann die einzelnen Agenten anhand ihrer Codebase unterscheiden.

Für die Kontrolle der Rechte einzelner Agenten ist dieses Konzept bei MobMan ungeeignet, da hier identische Klassen nur einmal geladen werden. Ein Security Manager ist aber dennoch für einen grundlegenden Schutz der Systemressourcen sehr gut geeignet, speziell um die Umgebung von MobMan und damit den Host zu schützen.

5.7.2 Realisierung

In MobMan kommt für den Grundschutz der `MobManSecurityManager` zum Einsatz, eine Kindklasse des Java-eigenen `SecurityManager`. Die MobMan-Fassung wird vom Security Agent installiert und über dessen Konfigurationsbereich parametrisiert (siehe Anhang A *Konfigurationsdateien von MobMan*, Seite 141). Der `MobManSecurityManager` unterstützt Logging und bietet folgende Zugriffsbeschränkungen:

- Dateisystem (nur ausgewählte Dateien).
- Netzwerk (einfache ACL).

- Start von Unterprozessen verhindern.
- Beenden nur für berechtigte Klassen.
- Laden von Klassen eingeschränkt.
- Manipulation der Event Queue unterbunden.
- Kein Zugriff auf das Clipboard.
- Anzeigen von Top Level-Fenstern nur mit Warnhinweis.

5.7.3 Resource Limits

Als Teil des Grundschutzes wäre es wünschenswert, auch Systemressourcen einzuschränken, beispielsweise Speicherplatz, Anzahl an Objekten, Anzahl an Klassen, Anzahl an Agenten, Laufzeit, Netzverbindungen oder den Platz auf der Festplatte. Mithilfe des Security Managers sind die meisten Ressourcen sehr gut einschränkbar. Wird den Agenten kein direkter Zugriff erlaubt, können auch agentenspezifische Beschränkungen durchgesetzt werden. Im Sinne eines Proxy [Fischer 99] können damit auch Dateigrößen und Ähnliches beschränkt werden – diese sind im Java-Sicherheitsmodell nicht vorgesehen.

Leider kann nicht für jeden einzelnen Agenten bestimmt werden, wie viel Speicher er belegt. Diese Ressource ist nur für die gesamte JVM einstellbar. Bei geteilten Ressourcen (mehrere Referenzen auf ein Objekt) wäre es auch schwer nachvollziehbar, welchem Agenten ein Objekt zuzuordnen ist. Eine Ausnahme von dieser Einschränkung sind Agentensysteme wie D'Agents, die jeden Agenten als eigenen Prozess ausführen. Damit sind Speicherlimits bei Java trivial. [Binder 01] schlägt auch eine Lösung für Agentensysteme vor, die nur aus einer Plattform bestehen. Die JVM muss dabei zwar nicht modifiziert werden, allerdings ändert sie den Bytecode ihrer Agenten.

Kapitel 6

Sicherheitsmechanismen

Viele Sicherheitsaufgaben sind nicht losgelöst von der Architektur zu betrachten, vielmehr muss die Architektur selbst sicher ausgelegt sein. Diese Aspekte wurden in den beiden vorangegangenen Kapiteln behandelt. Einige Mechanismen dienen jedoch ausschließlich der Sicherheit und sind das Thema dieses Kapitels, speziell die Authentifizierung, sichere Kommunikation und die Rechteverwaltung. Auch die Überprüfung neuer Agenten gehört in diese Kategorie.

6.1 Überprüfung neuer Agenten

Bei MobMan gibt es mehrere Wege, einen neuen Agenten zu starten:

- Zusammen mit der Plattform, die Klassen werden aus dem Dateisystem geladen.
- Migration, der Absender überträgt die Klassen als Teil des Protokolls.
- Als Kind eines anderen Agenten, dieser Agent bringt die Klassen mit.

Bei jeder dieser Varianten ist zu überprüfen, ob es sich überhaupt um einen Agenten handelt und wenn ja, ob er auf der Plattform willkommen ist. Agenten, die zusammen mit der Plattform gestartet werden, sind von einigen Prüfungen ausgenommen: Hier handelt es sich um die herkömmliche Installation von Software, bei der der Administrator dafür zuständig ist, nur „gutartige“ Programme zu installieren. Die Grundmechanismen von Java kommen aber auch bei diesen Agenten zum Zuge (Typprüfung, Bytecode Verifier, Security Manager, ...).

Will ein Agent selbst einen neuen Agenten erzeugen, so benutzt er dafür die Methode `requestBirth` aus der Basisklasse `Agent`. Diese Methode leitet seinen Wunsch per Message direkt an den Migration Agent weiter, so dass beide Fälle (Migration und Kind-Agent) weitgehend identisch behandelt werden.

Ob ein Agent willkommen ist, entscheidet der Migration Agent in sechs Schritten.

a) Authentifizierung des Absenders

Falls ein Agent neu auf die Plattform migriert, authentifiziert sich entsprechend des WAMP-Protokolls (siehe 4.8.2 *WAMP: WILMA Agent Migration Protocol*, Seite 66) zunächst der Kommunikationspartner. Das ist im ersten Schritt der Migration Agent auf der sendenden Plattform. Der User, als der sich dieser Kommunikationspartner authentifiziert, benötigt das Recht, Agenten zu starten (`StartAgents`).

b) Pre-Receive Welcome Check

Nach dem Empfang der Informationen über den Agenten (etwa Name und Version, auch der Code), aber noch vor dem Empfang des Agentenstatus (serialisiertes Objekt), überprüft die Methode `preReceiveWelcomeCheck` einige weitere Bedingungen. Ein möglicher Grund für eine Ablehnung ist ein Eintrag in der `RejectList`. Systemagenten oder die Plattform selbst registrieren hier Agenten, die nicht auf die Plattform migrieren dürfen.

Weiterhin prüft diese Methode, ob ein Agent gleichen Namens bereits auf der Plattform vorhanden ist. Wenn ja, muss dieser entweder erlauben, dass ein gleichnamiger Agent gleichzeitig existiert, oder dass der neue Agent den alten ersetzt. Andernfalls wird der neue Agent abgewiesen.

c) Prüfung der Klassen

Der Agent selbst führt eine Liste der von ihm benötigten Klassen mit. So lange diese Klassen lokal noch nicht vorhanden sind, fordert sie der Migration Agent von der Gegenstelle an und trägt sie in den lokalen Klassen-Cache ein. Der `MobManClassLoader` prüft bei Klassen, die nicht aus dem Dateisystem stammen, ob es sich um Systemklassen handelt. Dies ist der Fall, wenn der Paketname mit `java` oder `MobMan.Core` beginnt – in beiden Fällen wird diese Klasse nicht in den Cache eingetragen und auch nicht geladen.

d) Authentifizierung des Agenten

Noch vor dem Empfang des Agentenstatus kann sich der Agent selbst aus der Ferne authentifizieren.

e) Post-Receive Welcome Check

Nach dem Empfang des Agenten-Codes prüft die Methode `postReceiveWelcomeCheck`, ob der Agent ein Interface aus dem Paket `MobMan.Core` implementiert. Wenn ja, wird der Agent abgelehnt. Weiterhin muss der Agent die Basisklasse `MobMan.Core.Agent` besitzen. Handelt es sich um einen Systemagenten, muss der User über das Recht verfügen, Systemagenten zu starten (`StartSystemAgent`). Falls ein Agent unter dem Namen eines Systemagenten auf die Plattform migrieren will, dieser Systemagent aber keine gleichnamigen Agenten neben ihm zulässt, wird der Police Agent informiert.

f) Agent starten

Der neue Agent wird nun zur Plattform hinzugefügt, initialisiert und gestartet. Falls der Agent einen vorhanden ersetzen soll und darf, dann wird der alte Agent vorher noch beendet und entfernt, bevor der neue zum Einsatz kommt.

Nach der Initialisierung authentifiziert die Plattform den Agenten nun auch lokal. Erst diese Authentifizierung dient in der Folge dazu, dem Agenten Rechte auf der Plattform zuzuteilen. Der authentifizierte Benutzername wird auch an Empfänger von Messages, die dieser Agent sendet, weitergegeben.

6.2 Authentifizierung

Bei MobMan können die Agenten selbst entscheiden, welchen Plattformen sie vertrauen und ob sie auf diese auch migrieren wollen. Da für den Einsatzzweck die Plattform als sicher modelliert ist (siehe 3.4 *Relevanz für Anwendungen im Netzmanagement*, Seite 30), können die Agenten auch ihre Geheimnisse mit sich führen und sich somit selbst authentifizieren.

Einen anderen Weg beschreitet beispielsweise das Agentensystem D'Agents: Hier vertrauen sich die Plattformen gegenseitig. Nur die erste Plattform, die einen Agenten aufnimmt, authentifiziert diesen auch. Alle weiteren vertrauen den Angaben der vorherigen Plattformen bezüglich der Identität des Agenten. Verlässt ein Agent den Kreis sich gegenseitig vertrauender Plattformen und kommt dann zurück, wird er nicht mehr authentifiziert, er bleibt also anonym. Sollte er modifiziert worden sein, hat das keine besonderen Folgen; der Agent könnte als anonymer Agent auch direkt dorthin migriert sein.

Bei diesem Verfahren kann sich ein Agent nicht in mehreren Rollen authentifizieren. Die Identität bleibt auf allen Plattformen gleich, ein Agent kann also nicht entscheiden, wo er mit welcher Identität auftreten will. Ein Agent kann den Kreis nicht aus eigenem Wunsch verlassen, ohne anonym zu werden. Außerdem müssen die Vertrauensbeziehungen administriert werden, wodurch zusätzlicher Aufwand und gegenseitige Abhängigkeiten entstehen. Aus diesen Gründen verzichtet MobMan auf dieses Verfahren, jede Plattform agiert autonom.

Eine ähnliche Argumentation spricht auch dagegen, die zu migrierenden Objekte einzeln zu signieren [Gong 98a]. Für jedes migrierte Objekt wäre jeweils eine einzelne Verifikation nötig, außerdem würde nur der Absender authentifiziert und nicht der Agent.

6.2.1 Grundidee

Voraussetzung für eine Authentifizierung ist die Identifizierung des Kommunikationspartners. Diese Identifizierung muss eindeutig und unfälschbar sein, so lange die Bindung zwischen ID und authentifiziertem User besteht. Bei der Kommunikation über Netzwerke besteht diese Bindung in Form einer Verbindung. Kryptografische Protokolle wie SSL/TLS (6.4 *Gesicherte Kommunikation mit SSL/TLS*, Seite 106) kümmern sich dabei nicht nur um die Authentifizierung, Verschlüsselung und Integritätssicherung der übertragenen Daten, sie sorgen auch dafür, dass während der Verbindung kein Dritter unbemerkt Daten unterschieben kann.

Bei der plattforminternen Kommunikation ist der Aufwand eines solchen Protokolls nicht erwünscht und auch nicht notwendig. Das Grundproblem, den Aufrufer einer Funktion eindeutig zu identifizieren, wurde bereits in Kapitel 5.6 *Caller-ID*, Seite 83 beleuchtet und gelöst: Mithilfe der ID kann der Empfänger einer Nachricht sicher feststellen, wer der Absender war. Vertraulichkeit und Schutz vor Veränderungen ist innerhalb der Plattform gegeben.

Bei plattforminterner sowie bei externer Kommunikation sind vor allem folgende Anforderungen zu erfüllen:

- Die Kommunikationspartner und Agenten können sich selbst authentifizieren.
- Die mehrfache Authentifizierung einer Instanz führt dazu, dass sie die Rechte der einzelnen Kennungen kumuliert und mehrere Benutzernamen gleichzeitig annimmt.
- Bei externer Kommunikation erfolgt die Authentifizierung teilweise protokollspezifisch (notwendig bei SSL und HTTP), nach Möglichkeit aber über einheitliche Mechanismen (innerhalb der MAP-Protokollfamilie).

- Die externe und die interne Authentifizierung sollen weitgehend identische Verfahren benutzen.
- Die Authentifizierungsmechanismen sind möglichst modular zu realisieren und sollen auch nachträglich in fertige Agenten und Plattformen einfügbar sein.

6.2.2 Authentifizierungs-Client und -Server

Eine recht elegante Lösung für diese Anforderungen gelingt mit dem Plugin-Mechanismus von MobMan. Plugins können jederzeit nachgerüstet werden, ohne Änderungen am Code eines Agenten. Allein die Konfiguration entscheidet, welche Plugins ein Agent erhält.

Authentifizierung lässt sich als Client-Server-Mechanismus verstehen. Dabei authentifiziert sich der Client gegenüber dem Server.

- Die Client-Seite (Agent, Pia-Client, ...) erhält den nötigen Authentifizierer als Plugin, das mit den jeweiligen Daten (Geheimnissen) ausgestattet ist. Der Autor eines Agenten muss sich also nicht um die Authentifizierung kümmern.
- Ein Agent kann die Authentifizierung auch selbständig durchführen, hierfür muss er nur den Mechanismus selbst implementieren.
- Die Serverseite der Authentifizierung ist ebenfalls als Plugin zu lösen. Der Security Agent muss also nicht geändert werden, nur um einen weiteren Mechanismus zu implementieren.
- Auch hier gilt, dass der Security Agent den Mechanismus selbst implementieren könnte.

Durch die Realisierung als Plugin können die Implementierungen sehr einfach an anderen Stellen wiederverwendet werden. Sie sind so ausgelegt, dass sie sowohl plattformintern als auch über Plattformgrenzen hinweg funktionieren.

6.2.3 Plattforminterne Kommunikation

Innerhalb einer Plattform müssen sich die Agenten nur einmalig authentifizieren. Ab diesem Zeitpunkt genügt ihre ID, damit die Empfänger von Nachrichten feststellen können, als welcher Benutzer sich ein Agent authentifiziert hat.

a) Ablauf

Kommt ein Agent neu auf einer Plattform an, dann stößt diese eine Authentifizierung an. Sie aktiviert dazu das Authentifier-Plugin des Agenten. Das Plugin kommuniziert mit dem Security Agent und authentifiziert sich dabei. Der Security Agent weist dem Agenten dann seine authentifizierte Benutzerkennung und die dazugehörenden Rechte zu. Diese Informationen kann ein anderer Agent weiter benutzen (siehe Abbildung 6.1 *Lokale Authentifizierung*, Seite 97).

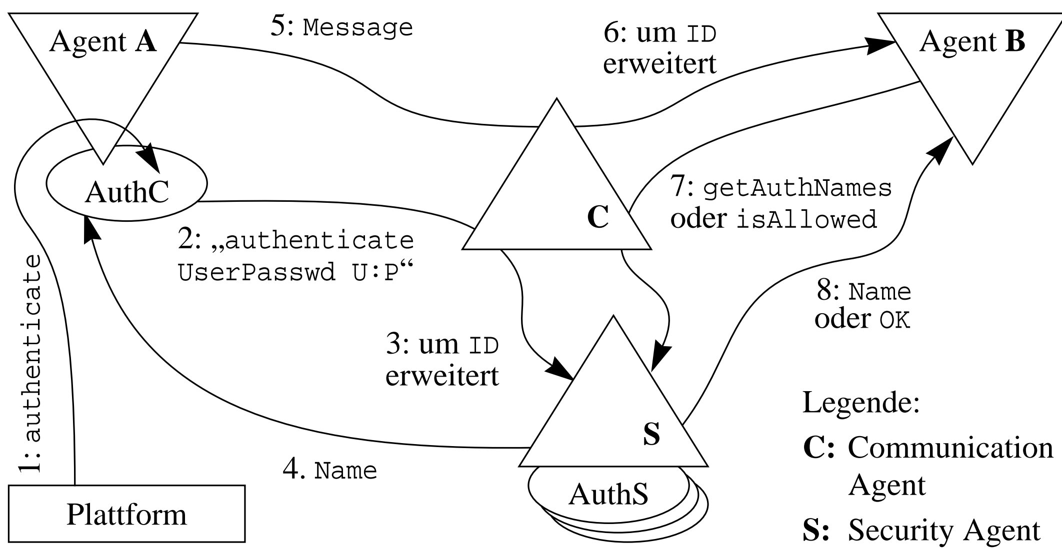

Abbildung 6.1 *Lokale Authentifizierung*

Der detaillierte Ablauf lässt sich in zwei Phasen unterteilen. In der ersten Phase findet die eigentliche Authentifizierung statt:

1. Die Plattform sucht das Authentifizierungs-Plugin (AuthC, Authentication Client) von Agent A und führt, soweit ein AuthC vorhanden ist, das `authenticate`-Kommando aus.
2. Der Agentenauthentifizierer führt stellvertretend für seinen Agenten (Agent A) die Authentifizierung durch. Dazu kommuniziert er mit dem Security Agent.
3. Über die Message-Klasse kann der Security Agent die Identität des Absenders ermitteln. Er nutzt dazu die `getID`-Methode.
4. Der Security Agent gibt die Kontrolle an das passende Authentifizierungs-Plugin (AuthS, Authentication Server) weiter, siehe b) *Details: User-Passwort-Verfahren*, Seite 98. Als Antwort liefert das Plugin eine der drei folgenden Möglichkeiten:
   – Authentifizierter Benutzername.
   – Leerer String, wenn sich der Agent nicht erfolgreich authentifizieren konnte.
   – `MobManObjectStream`-Objekt für die weitere Challenge-Response-Abwicklung.

   Der Security Agent speichert, wer sich unter der jeweiligen ID authentifiziert hat sowie welche Rechte diesem Benutzer zugeordnet werden. Bei mehrfacher Authentifizierung kumulieren die Rechte und die Benutzernamen. Verlässt ein Agent die Plattform, werden diese Einträge wieder entfernt.

In der zweiten Phase findet die authentische Kommunikation zwischen zwei Agenten statt, wobei die Agenten gegenseitig ihre Rechte überprüft können:

5. Agent A sendet eine `Message` an Agent B.
6. Auch hier ist die ID wieder durch die Message-Klasse sicher zugeordnet.
7. Agent B befragt nun den Security Agent über den Absender:
   – Anfrage nach allen authentischen Namen (`getAuthNames`). Ein Agent kann sich mit mehreren Kennungen gleichzeitig anmelden. Die ID von Agent A hat Agent B als Attachment an diese Nachricht angehängt.

  - Agent B kann auch fragen, ob Agent A (ID oder Name als Attachment) in der Plattformkonfiguration ein spezielles Recht zugestanden wurde (`isAllowed`). Diese Rechte stehen in der Datei `conf/users`.

8. Der Security Agent beantwortet diese Frage. Er liefert
   - einen Vektor mit authentischen Namen von Agent A oder
   - ein Flag, ob das Recht zugeteilt wurde.

b) Details: User-Passwort-Verfahren

Abbildung 6.2 *Lokale Authentifizierung: User-Passwort-Verfahren*, Seite 98 zeigt die Abläufe bei der Authentifizierung als UML-Sequenzendiagramm. In diesem Diagramm ist die MobMan-Nachrichtenzustellung durch einen stilisierten Brief gekennzeichnet.

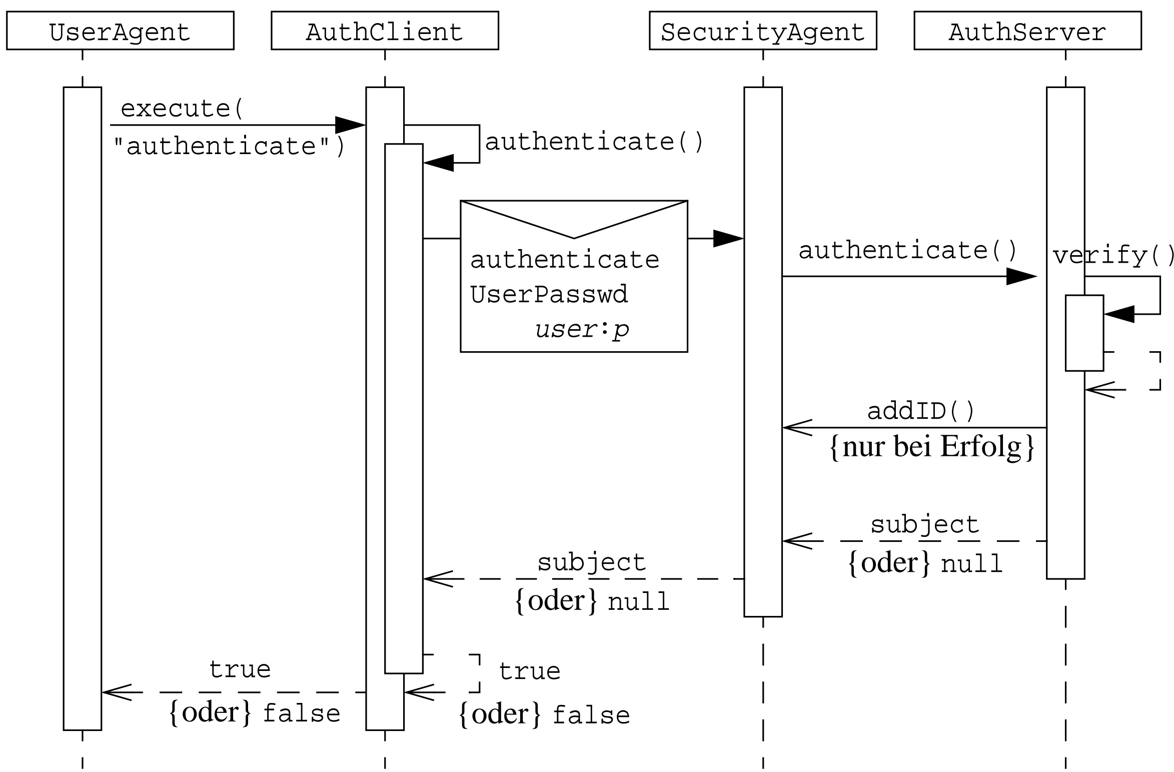

Abbildung 6.2 *Lokale Authentifizierung: User-Passwort-Verfahren*

Der Agent (oder die Plattform) rufen über die `execute`-Methode des Plugins dessen `authenticate`-Methode auf. Diese sendet eine Message an den Security Agent. Inhalt der Nachricht ist der Wunsch um Authentifizierung. Im Anhang befindet sich die Bezeichnung des gewünschten Verfahrens (`UserPasswd`) zusammen mit dem User-Namen und dem Passwort. Der Security Agent wählt das passende `AuthServer`-Plugin aus und übergibt ihm die Nachricht.

Die `verify`-Methode des Auth-Servers berechnet den MD5-Hash des übertragenen Passworts und codiert diesen als MIME-Base64-Zeichenkette. Diese Zeichenkette muss dem Eintrag in der `users`-Datei entsprechen, dann ist der Benutzer authentifiziert. Als Folge trägt das Plugin den Zusammenhang zwischen ID und authentischer Kennung im Security Agent ein und liefert den Namen als positive Antwort an den Authentifizierungs-Client zurück.

c) Details: Challenge-Response-Verfahren

Bei einem Challenge-Response-Verfahren ist etwas mehr Aufwand nötig, da hier Authentifizierungs-Client und -Server ein interaktives Protokoll abwickeln müssen. Hierzu benutzen sie einen `MobManObjectStream`, der eine transparente Verbindung zwischen Client und Server herstellt, über die beide Instanzen beliebige Java-Objekte austauschen können.

Der Erzeuger des `MobManObjectStream` muss einen neuen Thread bereitstellen, so dass der Ablauf nicht blockiert. Bei der Verbindung zwischen zwei Plattformen ist die Entkopplung durch die parallelen Prozesse bereits gegeben, der Empfänger der Verbindung stellt (im Sinne eines Proxy) seinen eigenen Thread zur Verfügung und gibt den Socket als `MobManObjectStream` weiter. Bei der lokalen Authentifizierung erzeugt der Server einen neuen Thread und gibt den `MobManObjectStream` an den Client zurück.

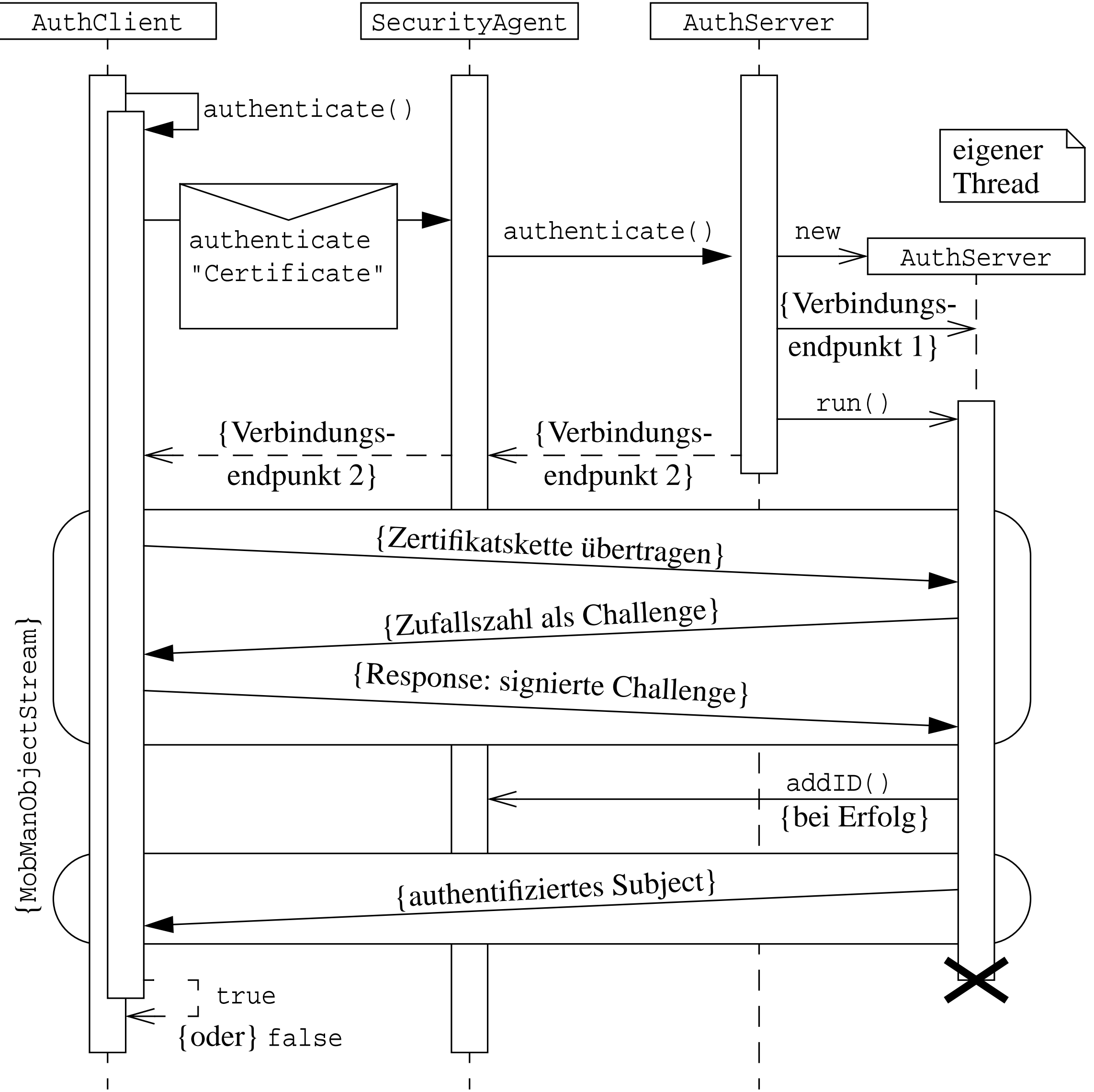

Abbildung 6.3 *Lokale Authentifizierung: Challenge-Response mit Zertifikat*

In Abbildung 6.3 *Lokale Authentifizierung: Challenge-Response mit Zertifikat*, Seite 99 ist zur Vereinfachung der User-Agent nicht mehr eingezeichnet. Seine Kommunikation mit dem `AuthClient` unterscheidet sich nicht vom User-Passwort-Verfahren. Direkt in die Grafik integriert ist die Kommunikation innerhalb der `MobManObjectStream`-Verbindung. Sie ist durch Rechtecke mit abgerundeten Ecken symbolisiert. Die Beschriftung der Pfeile bezeichnet die Objekte, die über diese Verbindung ausgetauscht werden:

- Der Client sendet die Zertifikatskette, also ein Array aus X.509-Zertifikaten, beginnend bei seinem eigenen bis hin zum Wurzelzertifikat.
- Der Server sendet daraufhin einen Zufallswert (Byte-Array), der als Challenge dient.
- Der Client signiert diese Challenge (derzeit als RSA-signierter MD5-Hash) und sendet das Ergebnis zurück an den Server.
- Konnte der Server die Signatur verifizieren, dann antwortet er mit dem authentifizierten Benutzernamen (CN, Common Name aus dem Zertifikat), andernfalls mit einer Null-Referenz.

6.2.4 Externe Kommunikation

Bei der externen Kommunikation besteht das Ziel darin, die Gegenstelle der Kommunikation zu authentifizieren. Der Communication Agent verteilt eingehende Verbindungen entsprechend dem MAP-Protokoll an die zuständigen Agenten. Diese sind dann selbst für die Protokollabwicklung zuständig. Für die Authentifizierung bietet ihnen der Security Agent zwar Unterstützung an, die Agenten können diesen Schritt aber auch selbst durchführen. In beiden Fällen werden keine Kennungen oder Rechte zugeteilt – diese würden fälschlich den lokalen Agenten betreffen.

Einige Agenten führen Aktionen stellvertretend für Benutzer durch (Pia, Migration Agent, Web Agent). Hierfür ist eine Authentifizierung des Benutzers nötig, wofür die selben Mechanismen wie bei lokaler Kommunikation eingesetzt werden.

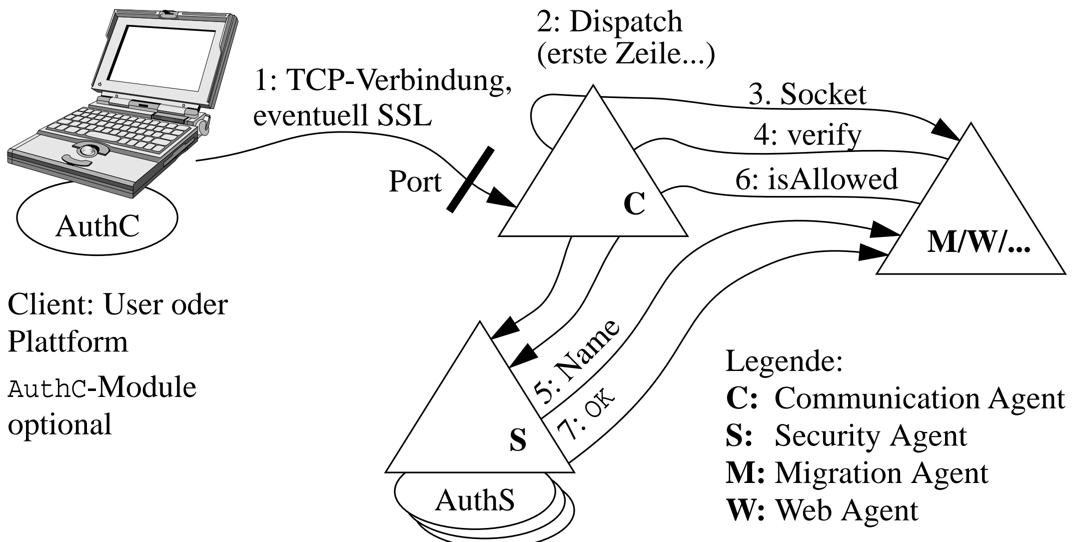

Abbildung 6.4 *Externe Authentifizierung*

Abbildung 6.4 *Externe Authentifizierung*, Seite 100 skizziert einen typischen Ablauf:

1. Der Client verbindet sich mit der Plattform über das Server-Socket des Communication Agent. Bei SSL wird der Client bereits beim SSL-Verbindungsaufbau authentifiziert. Die `AgentSocketSSL`-Klasse, vom Security Agent installiert, kümmert sich transparent um das SSL-Protokoll. Über sie sind später auch die Parameter der SSL-Verbindung zugänglich.
2. Der Communication Agent liest die erste Zeile des MAP-konformen Protokolls und entscheidet, an welchen Agenten er die Verbindung zur weiteren Bearbeitung gibt.
3. Das Server-Socket wird an den Empfänger weitergegeben, etwa an den Migration Agent oder den Web Agent. Der Name des authentifizierten Gegenübers ist bei einer SSL-Verbindung direkt über die Socket-Klasse zugänglich (Methode `getSubjectDN`).

Der Zielagent kontrolliert nun den weiteren Protokollablauf auf Applikationsebene. Es liegt in der Hand dieses Agenten, eine weitere Authentifizierung vorzunehmen, passend zum jeweiligen Client.

- `WebAgent`: Die gängigen Browser unterstützen das Basic-Authentication-Verfahren des HTTP-Protokolls, daher ist es auch im Web Agent implementiert.
- `PiaClient`: Der Client benutzt die MobMan-Authentifizierungs-Clients, daher kann auch der Server mit den entsprechenden Plugins arbeiten.

Die Authentifizierung könnte der Zielagent selbst durchführen, ein Auth-Server-Plugin verwenden, oder die Dienste des Security Agent nutzen. Auf eine `verify`-Nachricht überprüft der Security Agent die Authentizität. Der weitere Ablauf ist dann analog zur lokalen Authentifizierung, nur ändern sich hier weder die Identität noch die Rechte des Message-Absenders. Unter Verwendung eines Auth-Server-Plugins ist der Ablauf wie folgt:

4. Der Agent benutzt die `verify`-Nachricht, die im Gegensatz zur `authenticate`-Message (Schritt 2 in Abbildung 6.1 *Lokale Authentifizierung*, Seite 97) dem Anfrager zwar keine Rechte zuweist, sonst aber identisch funktioniert.
5. Der Auth-Server reagiert wie bei der internen Authentifizierung mit dem authentifizierten Namen, einem leeren String oder einem Objekt der Klasse `MobManObjectStream` (für die weitere Challenge-Response-Abwicklung).

Eine Rechteabfrage ist auch hier möglich:

6. Die Anfrage `isAllowed` enthält nicht die Agenten-IDs, sondern direkt den Benutzernamen und die Bezeichnung des abzufragenden Rechts.
7. Die Antwort darauf ist wieder ein Flag, das mitteilt, ob der Benutzer dieses Recht hat.

Die Klasse `MobManObjectStream` ist so flexibel gestaltet, dass sie nicht nur interne Kommunikation, sondern auch externe Kommunikation über Sockets abstrahiert. Dadurch können die selben Methoden der Authentifizierungs-Plugins auch bei externer Kommunikation genutzt werden. Der Client wird dazu mit einem parametrisierten Authentifizierer versehen. Er setzt einen `MobManObjectStream` auf die TCP-Verbindung auf (nach dem Senden der ersten Zeile mit dem MAP-Header) und gibt diesen an den Authentifizierer weiter.

6.2.5 Konfiguration

Der Benutzer ordnet seinen Agenten beim Start einen Agenten-Authentifizierer (`AuthClient`) zu und parametrisiert ihn. Für die Konfiguration innerhalb MobMan wurde ein eigenes Format entwickelt, dessen Syntax Anleihen bei C und Java nimmt. Ziel war ein übersichtliches, flexibles und erweiterbares Format, dessen Dateien einfach und schnell mit einem Texteditor erstellt werden können. Das Gruppieren der Parameter findet durch geschweifte Klammern „{...}“ statt und Parameter-Wert-Paare sind durch ein Gleichheitszeichen „=“ verbunden. Das genaue Format ist in Anhang A *Konfigurationsdateien von MobMan*, Seite 141 beschrieben.

a) Authentifizierungs-Client

Folgende Konfiguration weist dem Agenten `UserAgents.Demo.MessageTestAgent` neben dem Ziel der Testnachricht (`Destination`: Hostname, Portnummer und Name des Zielagenten) das Authentifier-Plugin `MobMan.Security.Utils.AuthClientPasswd` zu und konfiguriert es mit dem Benutzernamen `fjl` und dem Passwort `test`.

```
UserAgents.Demo.MessageTestAgent {
  Destination {
    host = ldvhp47
    port = 3000
    name = MessageTestAgent
  }
  plugins {
    Authentifier {
      class = MobMan.Security.Utils.AuthClientPasswd
      user = fjl
      password = test
    }
  }
}
```

Beim Challenge-Response-Verfahren benötigt der Client eine PKCS#12-Datei (Public-Key Cryptography Standards, Nummer 12) mit den Schlüsseldaten sowie die Passphrase, mit der die Schlüsseldatei geschützt ist:

```
UserAgents.Demo.MessageTestAgent {
  Destination {
    host = ldvhp47
    port = 3000
    name = MessageTestAgent
  }
  plugins {
    Authentifier {
      class = MobMan.Security.Utils.AuthClientCert
      certfile = certs/user-fjl.p12
      passphrase = MobMan
    }
  }
}
```

b) Authentifizierungs-Server

Die Auswahl des zur Anfrage passenden Auth-Server-Plugins geschieht über den Namen des Verfahrens. Dieser Name muss mit dem Namen des Plugins übereinstimmen. Den Namen setzt das Plugin selbst in seinem Konstruktor. Auth-Server-Module werden zum Security Agent hinzugefügt, ebenfalls als Plugin. Ihre Basisklasse kennzeichnet sie als Authentifizierungs-Server, daher können sie unterschiedliche Namen tragen. Folglich können auch mehrere Auth-Server gleichzeitig installiert sein. Beim `AuthServerCert`-Plugin sind neben der `users`-Datei zusätzlich die vertrauenswürdigen Zertifizierungsinstanzen (`trustedsigners`) anzugeben:

```
MobMan.Security.SecurityAgent {
  plugins {
    # Plugins, die von AuthServer erben,
    # dienen als Authentifizierungs-Server.
    AuthServerPasswd {
      class = MobMan.Security.Utils.AuthServerPasswd
      usersfile = users
    }
    AuthServerCert {
      class = MobMan.Security.Utils.AuthServerCert
      usersfile = users
      trustedsigners {
        certs/ca-LDV.der
      }
    }
  }
}
```

c) Users-Datei

In der `users`-Datei stehen die Benutzer, für jedes Verfahren getrennt, zusammen mit den zur Authentifizierung nötigen Daten. Beim User-Passwort-Verfahren ist ein MD5-Hash des Passworts MIME-Base64-codiert eingetragen. Die Datei enthält auch die zugeteilten Rechte.

```
# Benutzer für AuthServerUserPasswd-Plugin
UserPasswd {
  root {
    passwd = "oYGmA3acH5itkn5zZ8eqUQ=="
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
      PiaConfig
      KillAllAgents
    }
  }
  fjl {
    passwd = "CY9rzUYh03PK3k6DJie09g=="    # achim
    rights {
      StartAgent
    }
  }
}
```

```
# Benutzer für AuthServerCert-Plugin
# Namen müssen mit dem Common Name im Zertifikat übereinstimmen
Certificate {
  root {
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
      PiaConfig
      KillAllAgents
    }
  }
  fjl {
    rights {
      StartAgent
    }
  }
}
```

6.2.6 Authentifizierung beim Web Agent

Im Header des HTTP-Protokolls kann eine User-Passwort-basierte Authentifizierung (BasicAuth) stattfinden. Diese ist auf eine so genannte Protection Domain beschränkt, also nur für alle Seiten unterhalb (im Sinne einer Baumstruktur) der ursprünglichen Seite gültig. Bei jeder dieser Seiten überträgt der Browser das selbe Passwort erneut (HTTP ist zustandslos). Der Vorteil liegt darin, dass der Benutzer sein Passwort nur einmal eingegeben muss, ohne künstlich eine Bindung einzelner Aufrufe aneinander herstellen zu müssen. Es sind keine Cookies, Session-IDs im URL oder ähnliche Verfahren nötig.

Der Web Agent von MobMan benutzt diesen Standard-Mechanismus. Bei der Verwendung von Applets stößt dieses Verfahren jedoch an seine Grenzen, siehe Abbildung 6.5 *Authentifizierung von Applets*, Seite 104. Innerhalb des Applets ist kein Zugriff auf das im Browser gespeicherte Passwort möglich. Daher ist es nötig, dass innerhalb der Kommunikation zwischen Applet und Web Agent eine eigene Authentifizierung stattfindet. Beim Aufruf mehrerer Seiten mit eigenen Applets wird dies besonders lästig.

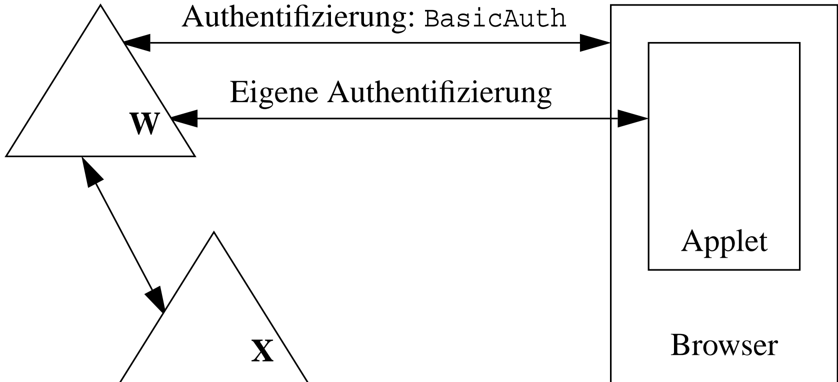

Abbildung 6.5 *Authentifizierung von Applets*

Die Seite, in der das Applet eingebettet ist, könnte zwar das Passwort als Parameter an das Applet übergeben. Allerdings liegt damit das Passwort im Quelltext vor. HTML-Seiten werden oft in mehreren Caches gespeichert, so dass hierdurch ein Sicherheitsproblem auftreten würde. Besser wäre immerhin die Vergabe eines nur einmal und für beschränkte Zeit gültigen OTPs (One Time Password). Dieses würde zwar die Gefahr vermindern, aber nicht wirklich beseitigen. Daher erscheint die mehrfache Authentifizierung sinnvoller.

6.3 Rechte

Die Rechte eines Agenten wurden bereits mehrfach angesprochen. MobMan benutzt einen allgemeinen Rechtebegriff, der vereinfachte Capabilities beschreibt. Andere Plattformen (siehe [Fünfrocken 98a]) benutzen ähnliche, meist aber komplexere Konzepte, etwa Capabilities (SAE), Permits (Telescript) oder Allowances (Ara).

Die MobMan-Rechte sind einem Agenten zugeordnet. Sie unterscheiden sich beispielsweise von Dateizugriffsrechten unter Unix, die dem zu schützenden Objekt (und nicht dem Subjekt) zugeordnet sind. Die Rechte stammen von dem Benutzer, als der sich der Agent authentifiziert hat. Es gibt bei MobMan keine feste Liste möglicher Rechte, sondern beschreibende Zeichenketten, die von den betroffenen Agenten zu interpretieren sind. Welche Folgen sich aus der Rechtevergabe ergeben, bestimmen die jeweils betroffenen Agenten. Jeder Agenten kann mithilfe des Security Agent prüfen, ob ein anderer Agent über ein bestimmtes Recht verfügt und damit entscheiden, wie er auf eine Anfrage (Message) reagiert.

Rechte werden jeweils lokal innerhalb einer Plattform vergeben. Potenzielle Namenskollisionen müssen vom Administrator der Plattform verhindert werden. Er muss auch sicherstellen, dass die Rechte nicht falsch interpretiert werden.

Recht	Bedeutung
`StartAgent`	Dieser Benutzer und seine Agenten dürfen neue User-Agenten starten.
`StartSystemAgent`	Systemagenten starten.
`SecurityConfig`	Gibt einem Pia-Client das Recht, die AuthServer-Plugins des Security Agent zu konfigurieren, etwa um neue Benutzer anzulegen.
`KillAllAgents`	Erlaubt einem Pia-Client, alle Agenten der Plattform zu beenden. Bei dieser Form des Beendens werden die Agenten nicht informiert.
`PiaConfig`	Benutzer darf den `PlatformInterfaceAgent` im laufenden Betrieb konfigurieren.
`PiaUser`	Benutzer darf prinzipiell Pia-Aktionen durchführen.

Tabelle 6.1 *Aktuell genutzte Rechte in MobMan*

Rechte können einem Agenten derzeit nicht wieder entzogen werden. Nach [Hawblitzel 98] wäre dies durchaus wünschenswert. Denkbar wäre, dass der Police Agent bei wiederholten Verstößen den Security Agent darüber informiert und dieser die Rechte entfernt und für den betroffenen Agenten dauerhaft sperrt. Interessant könnte auch eine Art freiwilliger Selbstkontrolle sein, bei der ein Agent selbst oder sein Absender durch die Konfiguration vorgeben, welche Rechte maximal einem Agenten zugewiesen werden dürfen. Statt einer einfachen Liste wäre auch ein Authorization-Plugin denkbar, das genau die nötigen Rechte anfordert.

6.4 Gesicherte Kommunikation mit SSL/TLS

An mehreren Stellen wurden bereits SSL (Secure Sockets Layer) und TLS (Transport Layer Security) genannt. TLS ist eine Weiterentwicklung von SSL Version 3 (siehe Anhang C *TLS: Transport Layer Security*, Seite 152). Dieses Protokoll dient bei MobMan unter anderem dem Schutz der Agenten bei der Übertragung [Wagner 96]. In dieser Arbeit wurde absichtlich darauf verzichtet, eigene kryptografische Protokolle zum Schutz der Agenten zu entwickeln. Viele Quellen (etwa [Roth 01a] und [Roth 01b]) bestätigen die große Gefahr, die von fehlerhaftem Design ausgehen. Diese Gefahr lässt sich durch den Einsatz bewährter Protokolle vermeiden.

SSL/TLS bietet im Wesentlichen folgende Eigenschaften:

- Schutz der Übertragung vor Abhören, Verfälschen und Unterschieben.
- Authentifizierung der Kommunikationspartner.
- Interoperabel, beispielsweise kommuniziert der Web Agent mit Webbrowsern.
- Nur eine Verbindungssicherung, keine signierten Agenten.
- Nicht für nachträgliche Nachweise geeignet (keine Nicht-Abstreitbarkeit).

JDK 1.1 bietet von sich aus zwar keine direkte SSL-Unterstützung, allerdings sind mehrere reine Java-Implementierungen verfügbar. In MobMan kommen iSaSiLk und IAIK-JCE zum Einsatz. Bei iSaSiLk handelt es sich um eine SSL-Implementierung, IAIK-JCE ist eine Java Cryptography Extension (JCE) und stellt damit einen Java Security Provider dar. Beide stammen vom IAIK (Institut für angewandte Informationsverarbeitung und Kommunikationstechnologie) der Technischen Universität Graz. Damit ist die SSL-Unterstützung jederzeit nachrüstbar, und zwar auf allen Hosts, die eine JVM besitzen. In Form eines Jar-Archivs könnte sogar ein Agent selbst die entsprechenden Klassen mit sich tragen.

Die plattformweite SSL-Umstellung wird vom Security Agent eingerichtet und transparent in das Agent-Socket-Interface integriert. Damit können sich die Plattformen gegenseitig authentifizieren. Eine eigene Wurzel-CA der Plattform-Administratoren bietet die erforderliche Unabhängigkeit von zentralen Institutionen.

6.4.1 Zertifikate

Zertifikate spielen eine zentrale Rolle bei der Authentifizierung über SSL/TLS. Ihre korrekte Handhabung ist allerdings nicht trivial. Innerhalb MobMan kommen neben den CA-Zertifikaten (Certificate Authority) zwei Arten von Zertifikaten zum Einsatz: Plattformzertifikate und Benutzerzertifikate. Der Zertifikatsaussteller bestätigt, dass ein öffentlicher Schlüssel zu dem im Zertifikat angegebenen Inhaber gehört. Zertifikate für Plattformen enthalten unter anderem:

- Angaben zum Aussteller.
- Angaben zum Inhaber (Subject).

- Angaben zur Identität (hier: Adresse und Port) der Plattform.
- Für Netscape ist der DNS-Name auch im Common Name enthalten (Netscape identifiziert ein Ziel nur anhand des DNS-Namens).
- Durch Erweiterungen könnte ein Zertifikat auch über die Rechte seines Besitzer Auskunft geben [Jansen 01].

Benutzerzertifikate sind nicht an eine IP-Adresse oder einen DNS-Namen gebunden. Der Absender identifiziert sich durch die Angaben im Zertifikat. Benutzerzertifikate können auch für die Authentifizierung von Agenten benutzt werden. Dabei wird allerdings kein SSL benutzt, und der Agent muss seinen privaten Schlüssel mit sich tragen.

Jede CA (Certificate Authority, sie stellt die Zertifikate aus) verfügt über ein eigenes Zertifikat, mit dem sie sich selbst ausweist. Eine Wurzel-CA (root level) benutzt ein von ihr selbst signiertes Zertifikat. Jede Plattform muss über eine Liste von vertrauenswürdigen Zertifikaten verfügen. Wenn es sich um CA-Zertifikate handelt, vertraut die Plattform allen von diesen CAs ausgestellten Zertifikaten.

Beim SSL-Verbindungsaufbau werden die Plattformzertifikate des Empfängers und bei gegenseitiger Authentifizierung zusätzlich die Plattform- oder Benutzerzertifikate des Absenders mit übertragen (siehe Abbildung 6.6 *Zertifikate und CAs*, Seite 107). Dabei ist die vollständige Zertifikatskette zur Root Level CA enthalten, es ist also keine weitere Infrastruktur nötig.

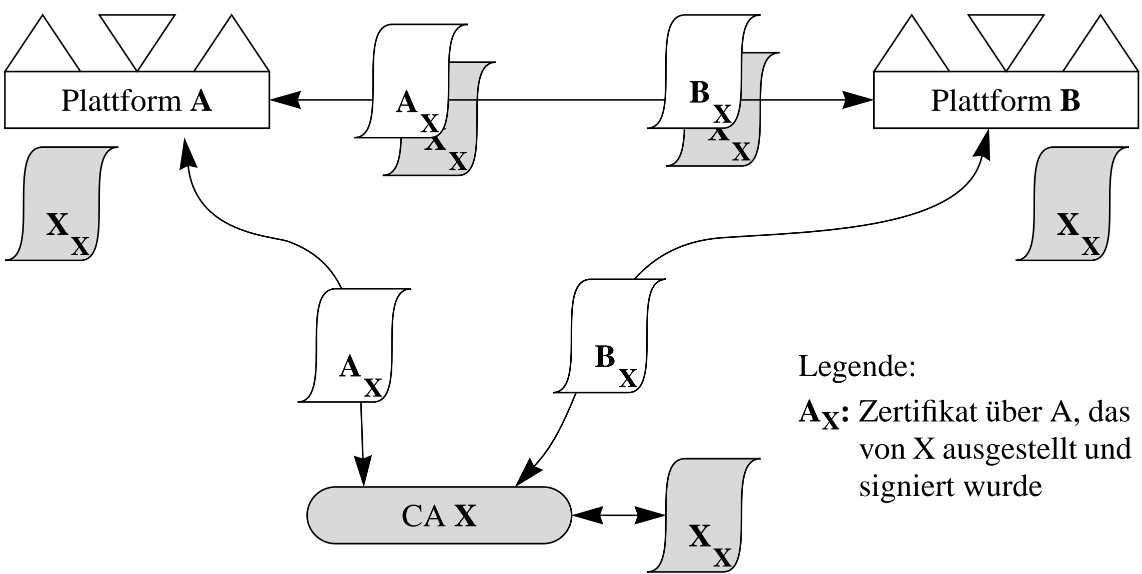

Abbildung 6.6 *Zertifikate und CAs*

Private Schlüssel und die dazugehörigen Zertifikate (oder ganze Zertifikatsketten) werden bei MobMan im PKCS#12-Format gespeichert. Dies betrifft die Plattform, die CA und die Agenten. In diesem Format ist bereits das Verschlüsseln des privaten Schlüssels mit einer Passphrase vorgesehen und wird bei MobMan auch eingesetzt. Der Dateisuffix lautet `.p12`. CA-Zertifikate werden DER-codiert (Distinguished Encoding Rules) gespeichert. Der Dateisuffix ist hier `.der`. Mit diesen Formaten lassen sich die Zertifikate auch in gängige Web-Browser importieren (siehe Anhang D *Schlüssel und Zertifikate in Browser importieren*, Seite 154).

6.4.2 Verifikation

Bei gegenseitiger Authentifizierung prüfen beide Instanzen, ob sie mit dem richtigen Partner verbunden sind. Die Verifikation muss dabei mehrere Schritte berücksichtigen. Die Überprüfung übernehmen eigene Trust Decider, die speziell für MobMan implementiert wurden.

Die Quelle weiß, mit welcher Plattform sie Verbindung aufnehmen wollte:

- Hostname: Entscheidend ist, dass der Client beim Verbindungswunsch einen Namen angegeben hat, der identisch ist mit dem Namen, der im Zertifikat als CN (Common Name) angegeben ist.
- Bei MobMan ist zusätzlich der Port entscheidend, da auf einem Rechner auch mehrere Plattformen laufen können.
- Neben dem CN wird die Standard-X.509v3-Extension „Subject Alternative Name“ im URL-Format benutzt. Es kommen die Protokollbezeichner `wamp`, `pia` und `https` zum Einsatz, jeweils zusammen mit dem Hostnamen und der Portnummer, der URL-Pfad ist leer. Diese Extension wird aber leider von Netscape Communicator 4 nicht unterstützt, daher ist zusätzlich der CN enthalten.
- Die Identifikation erfolgt durch Angabe des DNS-Namens oder der IP-Nummer zuzüglich Portnummer und Protokoll (Subject Alternative Name). Diese Daten werden mit den Daten im Zertifikat verglichen.

Das Ziel kann überprüfen, ob die Quelle wirklich diejenige ist, die sie vorgibt zu sein:

- Der Absender identifiziert sich nicht durch den Host und den Port, daher ist eine Übereinstimmung von Hostname und Common Name nicht erforderlich.
- Die Identifikation erfolgt direkt durch die Daten aus dem Zertifikat.
- Die Daten aus dem `Subject`-Eintrag sind für die Zuordnung von Rechten entscheidend (derzeit: CN-Feld).

Ein Key-Revocation-Verfahren ist derzeit nicht implementiert. Da die Plattformen sowieso administriert werden müssen, ergäbe sich auch nur ein marginaler Vorteil.

6.4.3 Mini-CA

Als Testwerkzeug enthält MobMan ein eigenes CA-Tool. Es ist nicht für den Produktiv-Einsatz gedacht und daher weder besonders flexibel noch besonders sicher. Es arbeitet als Root Level Certification Authority (keine Sub-CAs) und bietet folgende Dienste:

- Erzeugen eigener Root-Level-CA-Zertifikate.
- Erzeugen von Plattformzertifikaten nach X.509v3.
- Erzeugen von Benutzerzertifikaten nach X.509v3.
- In jedem Fall werden auch die Schlüsselpaare von der CA generiert, es sind daher keine Certificate Requests nötig.
- Das Schlüsselmaterial wird inklusive der Zertifikatskette als PKCS#12-Datei weitergegeben. Dieses Format können die gängigen Webbrowser importieren.
- Die Zertifikate werden als Datei mit DER-codiertem X509.v3-Zertifikat weitergegeben.
- Die Mini-CA kann das eigene CA-Zertifikat über HTTP anbieten. Dies vereinfacht das Importieren in die gängigen Browser.

Die Mini-CA lässt sich sehr einfach anwenden. Sie wird über das Launch-Programm aufgerufen (siehe Anhang B *Das Launch-Programm*, Seite 152).

Kapitel 7

Anwendungen

Die Praxistauglichkeit von MobMan ließ sich in einer Reihe von Anwendungen nachweisen. Das System wurde in erster Linie für das Netz- und Systemmanagement entwickelt, optimiert und getestet, konnte sich aber auch für komplexes Informationsmanagement bewähren. Darüber hinaus sind weitere Aufgabengebiete denkbar. Besonders attraktiv erscheint der Einsatz in Sicherheitsanwendungen.

7.1 Einsatzbeispiele

Im Praxiseinsatz in einer Reihe von Diplomarbeiten wurden vor allem frühe Versionen von MobMan genutzt. Auch wenn in diesen Versionen noch kaum Sicherheitseigenschaften implementiert waren, eignen sich die Anwendungen dennoch als Nachweis der Praxistauglichkeit: Die Sicherheitsfeatures sind so integriert, dass eine Portierung älterer Agenten problemlos möglich ist und bei einigen Beispielen auch erfolgreich durchgeführt wurde. Die meisten Änderungen sind eine Folge des neu gestalteten Konfigurationssystems.

7.1.1 Netz- und Systemmanagement

Passend zu seinem ursprünglichen Aufgabengebiet wurden für MobMan eine Reihe von Agenten entwickelt, die Aufgaben im Netzmanagement übernehmen [Trommer 99]. Ein Beispiel ist die Netzwerk-Leistungsmessung. Bei der passiven Leistungsmessung kommen zwei Agenten zum Einsatz. Der so genannte Planter (engl. für Sämaschine) migriert von Plattform zu Plattform und konfiguriert RMON-Agenten über das SNMP-Protokoll. Für die Wegeplanung nutzt

dieser Agent die Informationen in der Meta-MIB, einer ebenfalls am Lehrstuhl für Datenverarbeitung entstandenen SNMP-MIB, die Metainformationen über das Netzwerk vorhält. Die Ergebnisse der RMON-Messungen holt ein weiterer Agent, der Harvestor (engl. für Erntemaschine), später ab [Glissmann 99].

Bei der aktiven Leistungsmessung migrieren zwei Net Agents an je ein Ende der zu untersuchenden Strecke. Zusammen messen sie aktiv den Durchsatz dieser Verbindung. Der Net Agent spielt einmal die Rolle des Clients, einmal die Rolle des Servers. Auch diese Agenten nutzen die Informationen aus der Meta-MIB, um ihr Vorgehen zu planen [Heldwein 98]. In einem Vorgängerprojekt wurde bereits ein einfacherer Agent implementiert, der beispielsweise Ping für seine Messungen nutzt [Proell 98].

Im Rahmen dieses Projekts ist auch eine prototypische Bedienoberfläche entstanden, die über die Pia-Architektur Managementzugang zur Plattform bietet. Auf diesem Weg lassen sich die Ergebnisse der Agenten grafisch anzeigen [Görl 99].

Andere Mobile-Agenten-Plattformen konnten die Eignung der Technik für weitere Aufgaben nachweisen, etwa für das Modellieren ganzer Netzwerke inklusive der Erkennung [White 98] oder die Überwachung des „Gesundheits“-Zustands von Systemen und Netzen [Zapf 99b].

7.1.2 Informationsmanagement

Ein weiteres Einsatzgebiete für MobMan hat nur wenig mit Netzmanagement gemein: Informationsmanagement. Das Agentensystem erfüllt hier eine vermittelnde Rolle zwischen Informationsanbieter und Informationsnachfrager. Mithilfe dezentraler Metadaten-Verzeichnisse können die Agenten miteinander und mit ihren Benutzern kommunizieren. Dadurch ist eine Selbstorganisation möglich [Füssl 01]. In dieses Informationssystem lassen sich auch beliebige externe Datenquellen integrieren. Eine Abstraktion in Form von Metadaten erlaubt die einheitliche und strukturierte Darstellung heterogener Informationsquellen. Eine eigene Wegesuche unterstützt den dezentralen Aufbau des Systems [Utermann 00].

7.2 Mögliche Anwendungen

MobMan ist nicht nur auf Netzmanagement spezialisiert, sondern für viele unterschiedliche Einsatzbereiche geeignet. In der Forschung zu mobilen Agenten wurden bereits sehr früh breite Einsatzszenarien entworfen [Chess 95] [Dula 97], die bis hin zu einer Web-Bevölkerung reichen [Condict 96]. Weitere Übersichten mit möglichen Anwendungsgebieten finden sich etwa in [Fuggetta 98], [Lange 98a] oder [Fünfrocken 98b]. Für den Entwurf von Agenten wurden auch Design-Pattern entwickelt [Aridor 98b], selbst für ganze Agenten-Systeme existieren Ansätze für Design-Pattern [Silva 98a].

Teilweise wird für die Zukunft mobiler Agenten das Fehlen einer so genannten Killer-Applikation beklagt [Kotz 99], [Wong 99], die für den Durchbruch der Agententechnik und für ihren breiten Einsatz sorgen könnte. Einige Autoren gehen sogar davon aus, dass es keine Killer-Applikation geben wird [Lange 99]. Allerdings hat Netzmanagement durchaus gute Chancen, eine überragende Rolle beim praktischen Einsatz von mobilen Agenten zu spielen. Ob, wie etwa [Kotz 99] erwartete, jemals alle wichtigen Internet-Sites mobile Agenten ausführen werden, darf jedoch bezweifelt werden.

In der Anfangszeit der mobilen Agenten wurde E-Commerce als ein aussichtsreiches Einsatzfeld gesehen. Etwa ein Einkaufsassistent, der von Händler zu Händler migriert, die Produkte anhand seiner Anforderungen auswählt und am Ende die Preise vergleicht. Im Unterschied zu Anwendungen im Netzmanagement stellen sich hier Probleme, die kaum zu lösen sind:

- Der Agent muss sich vor der Plattform schützen – ein bisher praktisch nicht gelöstes Problem, da die Plattform letztlich den Agenten ausführt und damit kontrolliert (siehe 2.2.3 *Schutz der Agenten vor den Plattformen*, Seite 20).
- Selbst wenn die Plattform einen Agenten nicht unmittelbar manipuliert, kann sie ihm eine falsche Umgebung vorspielen und ihn dadurch zu falschen Schlussfolgerungen verleiten.
- Die beteiligten Instanzen kennen sich nicht, wodurch die Authentifizierung entweder nicht möglich ist oder eine eigene Infrastruktur erfordert. Selbst wenn sich die Agentenbenutzer authentifizieren, müssen daraus noch die jeweiligen Rechte abgeleitet werden.
- Auch wenn alle eben genannten Probleme gelöst wären: Kaum ein Shop-Betreiber hätte vermutlich Interesse daran, einem Agenten Rechenzeit zu geben und sich den Gefahren auszusetzen, um dann im Preisvergleich zu unterliegen.

Trotz dieser prinzipiellen Schwierigkeiten gehen viele Gruppen davon aus, dass die Sicherheitsanforderungen auch in diesem Umfeld lösbar sind [Marques 99].

7.3 Einsatz für Sicherheitsanwendungen

Vielversprechend ist der Einsatz mobiler Agenten für Sicherheitsanwendungen. Mobile Agenten werfen eben nicht nur Sicherheitsprobleme auf, durch ihre besonderen Eigenschaften könnten sie auch bei der Realisierung neuer und der Verbesserung vorhandener Sicherheitsanwendungen eine wichtige Rolle spielen.

7.3.1 Host Security

Mobile Agenten werden auf dem jeweiligen Host lokal ausgeführt. Üblicherweise leitet man daraus Gefahren für die Sicherheit ab; diese zu beherrschen war Aufgabe dieser Arbeit. Durch die lokale Ausführung können mobile Agenten aber auch die Sicherheit ihres Hosts beurteilen und erhöhen.

a) Voraussetzungen

Um die Sicherheit eines Hosts zu analysieren, ist ein direkter Zugriff auf dessen Ressourcen (Dateien, Prozesse, ...) nötig. Da Agenten zu einem Host migrieren können, haben sie prinzipiell die Möglichkeit, derartige Aktionen durchzuführen. Allerdings werden diese normalerweise durch die Sandbox, die der Security Manager aufbaut, verhindert. Es ist also nötig, die Barrieren gezielt zu lockern. Da der Security Manager nur anhand der Klassen entscheiden kann, ob ein Zugriff erlaubt ist oder nicht, kann dies keinem Agenten direkt erlaubt werden.

Als Schleuse müssten spezielle Klassen dienen, die nicht vom Benutzer eingeführt werden können (aus dem `MobMan.Core`-Package, nicht serialisierbar). Diesen Klassen würde dann der Security Manager gestatten, auf die lokalen Ressourcen zuzugreifen. Ein eigener Systemagent würde Objekte dieser Klassen herausgeben, die dann den gezielten Zugriff ermöglichen (Proxy-Prinzip). Der Systemagent würde entscheiden, wer die Schleusen-Objekte erhält, und worauf diese Objekte den Zugriff erlauben.

b) Security Scanner

Als lokal ablaufendes Stück Software können mobile Agenten beispielsweise die Version und Konfiguration der installierten Software überprüfen oder die Einträge in Log-Dateien überwachen. Damit erfüllen sie die Aufgaben von lokalen Security-Scannern wie etwa Cops oder Tiger [Humphries 00]. Im Gegensatz zu diesen klassischen Werkzeugen müssen Agenten nicht lokal installiert werden. Zusätzlich können sie durch ihre Migrationsfähigkeit die Daten mehrerer Hosts korrelieren und damit neue Erkenntnisse erzielen [Tripathi 02a]. Sie können die Daten auch mit den Ergebnissen eines Netzwerk-Scans abgleichen (siehe 7.3.2 *Netzwerk-Scanner*, Seite 112).

c) Konfiguration und Patches

Neben der Analyse der Sicherheit können mobile Agenten auch die entdeckten Probleme beheben, etwa indem sie Zugriffsrechte auf Dateien anpassen oder Patches und neue Programmversionen einspielen. Sie können auch komplexe Änderungen an den Konfigurationsdateien vornehmen. Statt einfach eine ganze Datei auszutauschen, könnte ein mobiler Agent durch seine Programmierbarkeit gezielt einzelne Einträge ändern, etwa ihre Werte modifizieren.

7.3.2 Netzwerk-Scanner

Neben dem Zugang zum lokalen Host können mobile Agenten auch auf das Netzwerk zugreifen und folglich typische Netzwerk-Scans durchführen. Hier ist der Vorteil darin zu sehen, dass der Scan von beliebiger Stelle im Netz ausgehen kann, vorausgesetzt es ist eine Plattform installiert. Damit können Agenten auch Netzkomponenten untersuchen, die von der Manager-Station aus nicht erreichbar sind (Abbildung 7.1 *Portscanner bei vorhandener Firewall*, Seite 112), etwa weil eine Firewall den Zugriff verhindert.

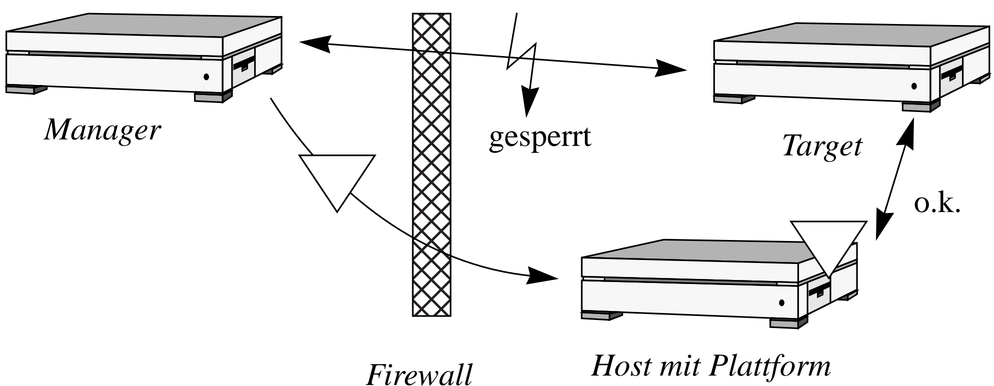

Abbildung 7.1 *Portscanner bei vorhandener Firewall*

Zusätzlich können die Agenten kooperieren, um etwa die Filterregeln einer Firewall zu überprüfen (Abbildung 7.2 *Firewall-Test mit mobilen Agenten*, Seite 113).

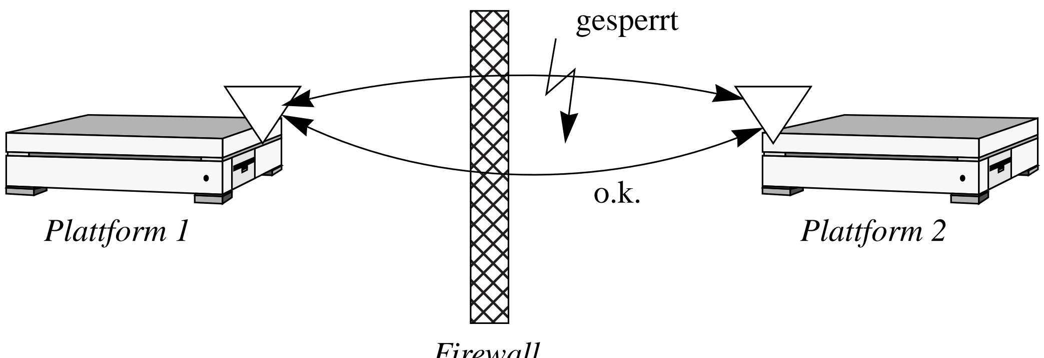

Abbildung 7.2 *Firewall-Test mit mobilen Agenten*

7.3.3 Intrusion Detection

Wenn mobile Agenten die Ergebnisse ihrer Scans selbst auswerten und daraus Rückschlüsse ziehen, können sie als Intrusion Detection System (IDS) dienen und dabei komplexere Untersuchungen durchführen als herkömmliche Systeme. [Balasubramaniyan 98] schlägt bereits autonome Agenten für IDS vor. Durch die Erweiterung um mobile Agenten [Farmer 96a] könnte ein einzelner Agent beispielsweise eine Spur verfolgen, indem er dem Angreifer folgt. Ähnlich der zentralen Auswertung einzelner Ergebnisse könnte ein solcher Ansatz unmittelbar Zusammenhänge aufdecken. Der verfolgende mobile Agent kennt die Vorgeschichte und kann daher die jeweiligen Schritte im Kontext beurteilen. Durch die Integration von Netzwerk-Management und IDS ergeben sich weitere Vorteile [Cabrera 02].

Weitere Arbeiten zum Einsatz mobiler Agenten für IDS siehe [Helmer 98], [Asaka 99] und [Fenet 01]. Einige Autoren weisen aber neben den Vorteilen mobiler Agenten für IDS-Systeme auch auf die Schwierigkeit hin, diese Systeme in der Praxis erfolgreich zu implementieren [Jansen 99b]. Eines der Hauptprobleme könnte die Performance sein: IDS-Systeme müssen oft sehr große Datenmengen verarbeiten, wofür sich interpretierte Sprachen nur bedingt eignen. Auch und gerade bei IDS-Systemen ist die Sicherheit der mobilen Agenten ein kritischer Punkt, sie stellen schließlich eine zusätzliche Angriffsfläche dar.

Kapitel 8

Zusammenfassung, Bewertung und Ausblick

Diese Arbeit stellt die Architektur eines sicheren Mobile-Agenten-Systems vor, das besonders für die Aufgaben im Netzmanagement ausgelegt ist. Sein Einsatzbereich ist aber nicht auf dieses Szenario eingeschränkt.

8.1 Zusammenfassung

Ausgangspunkt für die Sicherheitsarchitektur ist eine Analyse der Bedrohungen, denen das Agentensystem ausgesetzt ist. Dabei wird zunächst festgestellt, dass viele alte Annahmen im Kontext mobiler Agenten nicht gelten. Aus einer Bewertung der beteiligten Instanzen bezüglich ihrer Relevanz als potenzieller Angreifer leiten sich die notwendigen Schutzmechanismen ab. Eine zentrale Sicherheits-Policy, formuliert in einer eigenen Konfigurationssprache, ist für mobile Agenten kaum geeignet. Es gibt weder eine zentrale Instanz, die dieses Regelwerk festlegen könnte, noch ist a priori bekannt, welche Entscheidungen anhand welcher Daten zu treffen sind. Vielmehr müssen sich die Agenten selbst schützen. Um diesen Selbstschutz effizient durchführen zu können, sind eine Reihe von Barrieren und Basisdiensten nötig, die die Plattform und die Architektur bereitstellen müssen.

Eine Sicherheitsarchitektur ist in der Regel integraler Bestandteil des Systems, das sie schützt. Dies gilt auch im vorliegenden Fall. Es sind viele Aspekte der Gesamtarchitektur von MobMan betroffen, daher wird diese mit vorgestellt. MobMan setzt das Agentenparadigma sehr konsequent um. Die Plattform selbst stellt nur die notwendigen Basisdienste bereit, selbst grundlegende Aufgaben wie Kommunikation und Migration sind als Agenten implementiert. Diese Systemagenten sind stationär und verfügen über mehr Rechte als gewöhnliche Benutzeragenten, ihre Schnittstellen und das Programmiermodell sind jedoch identisch.

Besondere Flexibilität bieten Agenten-Plugins. Ursprünglich eingeführt, um die Authentifizierung unabhängig von der Implementierung des Agenten zu ermöglichen, können sie durch eine Verallgemeinerung beliebige Aufgaben übernehmen. Authentifizierungs-Plugins erfüllen eine Sonderrolle, da sie bei der Migration automatisch aufgerufen werden. Damit ist es möglich, neue Mechanismen und neue Identitäten nachträglich in einen Agenten einzufügen.

Große Bedeutung für die Sicherheit des Agentensystems hat die Kommunikation, sowohl plattformintern, als auch über Plattformgrenzen hinweg. Die Protokolle wurden kompatibel zu HTTP entworfen: Das Handshake-Verfahren einer eigenen Zwischenschicht trennt die Anwendungsprotokolle. Ein Web Agent kann so Web-Seiten über denselben Port anbieten, über den gleichzeitig die Migration, Administration und Kommunikation der Plattform stattfindet.

Bei der Realisierung waren eine Reihe Java-spezifischer Probleme zu lösen. Die Sprache bietet zwar eine Menge nützlicher Eigenschaften, die zur Auswahl als Implementierungssprache führten und entsprechend gewürdigt werden. Das Java-Sicherheitsmodell geht jedoch davon aus, dass innerhalb einer JVM nur eine Applikation läuft, deren Bestandteile einheitliche Rechte besitzen. Die Beschränkungen betreffen auch die Trennung der Klassen: Ad-hoc-Verfahren zur Vermeidung von Namenskollisionen greifen bei mobilen Agenten nicht. In klassischen Applikationen vermeidet der Autor solche Konflikte, an Mobile-Agenten-Systemen können jedoch viele voneinander unabhängige Autoren beteiligt sein. Die Trennung der Klassen soll aber im Sinne der Ressourcen-Ökonomie nicht dazu führen, dass identische Klassen mehrfach vorhanden sind. Durch die Verwendung verschiedener Classloader bei gleichzeitiger Erkennung identischer Klassen anhand kryptografischer Hash-Codes wird dieses Ziel erreicht.

Eine wichtige Aufgabe der Plattform ist es, die Agenten eindeutig identifizieren zu können, ohne Referenzen aufeinander preiszugeben. Hierfür kommen `ID`-Objekte zum Einsatz, die in der Sonderform als `AgentProperties` auch Informationen über den Agenten mitteilen. Bei der Kommunikation war das Problem zu lösen, dass Java einer aufgerufenen Methode keinen Mechanismus bereitstellt, um das rufende Objekt und die rufende Methode festzustellen. Bei der Kommunikation im Agentensystem muss der Empfänger aber den Absender sicher identifizieren können. Hierfür wird eine geeignete Lösung vorgeschlagen.

In MobMan wurden eine Reihe von speziellen Sicherheitsmechanismen eingeführt, die weitgehend losgelöst von der restlichen Architektur betrachtet werden können. Eine plattformzentrale Rechteverwaltung vereinfacht die Administration. Weitere wichtige Sicherheitsmechanismen sind die Überprüfung neuer Agenten sowie die Authentifizierung. Die Umsetzung als Authentifizierungs-Client und -Server in Agenten-Plugins erlaubt es, ohne Änderung am Agenten diesen mit neuen Authentifizierungsmechanismen und neuen Identitäten auszustatten. Das Verfahren ist lokal und remote über Plattformgrenzen hinweg einsetzbar. Hier bewähren sich die MobMan-Protokolle, die als Java-Objektstrom realisiert sind. Die Protokolle können zudem über SSL/TLS kryptografisch gesichert werden. Für den Laboreinsatz entstand eine Mini-CA, die X.509v3-konforme Zertifikate ausstellt.

8.2 Bewertung

Das im Rahmen dieser Arbeit weiterentwickelte Mobile-Agenten-System MobMan konnte sich in vielen Labortests bewähren. In einer Reihe von Anwendungen im Netzmanagement sowie im Informationsmanagement wurde die Praxistauglichkeit nachgewiesen. Bei diesen Arbeiten kamen zwar verschiedene frühere Versionen zum Einsatz, die Änderungen sind aus Sicht der Agenten jedoch gering. Bei der Portierung älterer Agenten ist vor allem die geänderte Konfigurationssprache zu beachten.

Für eine detaillierte Bewertung können die Anforderungen aus Kapitel 4.1 *Designkriterien*, Seite 38 dienen. Alle Kriterien sind erfüllt:

- *Unterstützung für Legacy-Systeme* (etwa SNMP): Durch reine Java-Implementierungen von SNMP ist diese Anforderung zu erfüllen. Um den Overhead zu minimieren, kann ein SNMP-Agent diese Aufgabe als Dienst für andere Agenten übernehmen. Java und Java-Klassenbibliotheken ermöglichen auch den Zugang zu Datenbanken, zu Directory-Diensten und vielen anderen Systemen. Durch die MAP-Protokollarchitektur lässt sich MobMan auch mit einem Webbrowser kontaktieren. Der Web Agent stellt hierbei eine Reihe von Administrationsdiensten bereit.
- *Portabilität* und *Plattformunabhängigkeit:* Durch die Verwendung von reinem Java ohne Änderung der JVM ist ein hohes Maß an Portabilität erreicht. Als Mindestversion ist lediglich JVM 1.1 vorausgesetzt.
- *Persistenter Zustand:* Die Agenten behalten ihre Daten auch über die Migration hinaus. Ein Persistence Agent sorgt zusätzlich auf Anforderung für ein Backup.
- *Sicherheit:* Die in Kapitel 3.6 *Erforderliche Barrieren*, Seite 34 genannten Schutzmechanismen sind implementiert. Dennoch blieb das Gesamtkonzept übersichtlich, wie das überschaubare Klassendiagramm (Abbildung 4.2 *Klassenmodell des MobMan-Systems – Übersicht*, Seite 41) zeigt.
- *Zugang zu den Host-Ressourcen:* Soweit von den Sicherheitseinstellungen erlaubt, ist dieser Zugang möglich.
- *Kommunikation* zwischen Agenten: Agenten können über den Messaging-Mechanismus kooperieren. Einfache Anfragen lassen sich einfacher über die `AgentProperties`-Objekte abwickeln. Für eine enge Kooperation können Agenten auch Referenzen auf sich selbst freiwillig an andere Agenten weitergeben und damit gegenseitig ihre Methoden aufrufen.
- *Wenig Overhead bei der Übertragung:* Jede Klasse wird nur einmal auf eine Plattform übermittelt. Durch kryptografische Hash-Codes ist dennoch sichergestellt, dass gleichnamige Klassen zuverlässig unterschieden werden.
- *Wenig Abhängigkeiten*, vor allem keine zentralen Dienste: Die Implementierung erfolgt in reinem Java, zur Ausführung ist lediglich eine Standard-JVM ab Version 1.1 nötig. Es gibt keine zentrale Registry, keinen Directory Service und keinen Class-Server. Zur Adressierung der Plattform und der Agenten ist nur der Hostname oder dessen IP-Nummer nötig. `Neighbour`-Agenten bieten eine virtuelle Vernetzung der Plattformen.
- *Hohe Flexibilität* zur Anpassung an lokale Gegebenheiten: Die konsequente Anwendung des Agentenparadigmas führt zu einer hohen Modularität. Durch den Austausch oder die Modifikation einzelner Agenten können so zentrale Aspekte wie Kommunikation, Migration und Sicherheit angepasst werden. Hervorzuheben sind die Authentifizierungs-Plugins, die diese zentrale Aufgabe hochgradig modular umsetzen.

8.3 Ausblick

Als nächster Schritt sollte ein ausführlicher Test der Sicherheitseigenschaften folgen. Hierfür wären Angriffsszenarien zu entwickeln und zu implementieren, um die vorhandenen Barrieren zu überprüfen. Der Einsatz für Sicherheitsanwendungen wie Security Scanner und Intrusion Detection ist ein vielversprechendes Feld für weitere Entwicklungen. In diesem Rahmen könnte die Praxistauglichkeit von MobMan weiter verfeinert werden.

Abkürzungsverzeichnis

ACL	Access Control List Agent Communication Language
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation Number 1
ATP	Agent Transfer Protocol
AWT	Abstract Window Toolkit
BNF	Backus-Naur-Form
CA	Certificate Authority
CIM	Common Information Model
CMIP	Common Management Information Protocol
CN	Common Name
COD	Code on Demand
CORBA	Common Request Broker Architecture
DCE	Distributed Computing Environment
DER	Destinguished Encoding Rules
DISMAN	Distributed Management
DMI	Desktop Management Interface
DMTF	Distributed Management Task Force
DN	Destinguished Name
DNS	Domain Name System
DoD	Department of Defense

DoS	Denial of Sevice
EBNF	Erweiterte Backus-Naur-Form
FIPA	Foundation for Intelligent Physical Agents
GSM	Ursprünglich: Groupe Spéciale Mobile Heute: Global System for Mobile communication
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
IrDA	Infrared Data Association
ISO	International Organization for Standardization
JAR	Java Archive
JDK	Java Development Kit
JVM	Java Virtual Machine
LAN	Local Area Network
MA	Mobiler Agent
MAC	Media Access Control
MAP	MobMan Application Protocol
MASIF	Mobile Agent System Interoperability Facility
MD5	Message Digest Nr. 5
MIB	Management Information Base
MIME	Multipurpose Internet Mail Extensions
MO	Managed Object
MOF	Managed Object Format
MobMan	Mobile Manager
NCS	Network Control Station
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open Systems Interconnection
OTP	One Time Password
PCC	Proof Carrying Code
PDA	Personal Digital Assistant

PDU	Protocol Data Unit
PGP	Pretty Good Privacy
PIA	Platform Interface Agent
PISA	PIA Service Agent
PKCS	Public-Key Cryptography Standards
RAMP	Remote Agent Messaging Protocol
REV	Remote Evaluation
RMI	Java Remote Method Invocation
RMON	Remote Monitoring
RPC	Remote Procedure Call
SHA-1	Secure Hash Nr. 1
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol
SSL	Secure Sockets Layer
TAN	Transaktionsnummer
TCL	Tool Command Language
TCP	Transport Control Protocol
TLS	Transport Layer Security
TMN	Telecommunications Management Network
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
WAMP	WILMA Agent Migration Protocol
WAN	Wide Area Network
WAP	Wireless Application Protocol
WBEM	Web-Based Enterprise Management
WILMA	Wissensbasiertes LAN-Management
WRAP	WILMA Remote Administration Protocol
WML	Wireless Markup Language
XML	Extensible Markup Language

Literaturverzeichnis

[Abdalla 97] Michel Abdalla, Walfredo Cirne, Leslie Franklin und Abdallah Tabbara: *Security Issues in Agent Based Computing.* In: 15th Brazilian Symposium on Computer Networks, São Carlos/Brasilien 1997.

[Aridor 98a] Yariv Aridor und Mitsuru Oshima: *Infrastructure for Mobile Agents: Requirements and Design*. In: Kurt Rothermel (Hg.): Proc. Second International Workshop on Mobile Agents (MA'98), Stuttgart, September 1998. Vol. 1477 Lecture Notes in Computer Science, Springer Verlag, Berlin 1998.

[Aridor 98b] Yariv Aridor und Danny B. Lange: *Agent Design patterns: Elements of Agent Application Design*. In: Proc. Second International Conference on Autonomous Agents (Agents '98), ACM Press, 1998.

[Asaka 99] M. Asaka, S. Okazawa, A. Taguchi und S. Goto: *A Method of Tracing Intruders by Use of Mobile Agents*. In: Proc. Ninth Annual Internet Society Conference, INET '99, San Jose, Kalifornien/USA, Juni 1999.

[Avvenuti 98] M. Avvenuti, A. Puliafito, and O. Tomarchio: *W-MAP: Web-accessible Mobile Agents Platform*. In: Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS98), Budapest/Ungarn, September 1998.

[Baldi 97] Mario Baldi, Silvano Gai und Gian Pietro Picco: *Exploiting code mobility in decentralized and flexible network management*. In: Kurt Rothermel (Hg.): Proc. First International Workshop on Mobile Agents (MA'97), Berlin, April 1997. Vol. 1219 Lecture Notes in Computer Science, Springer Verlag, Berlin 1997.

[Baldi 98] Mario Baldi und Gian Pietro Picco: *Evaluating the tradeoffs of mobile code design paradigms in network management applications*. In: R. Kemmerer and K. Futatsugi (Hg.): Proc. 20th International Conference on Software Engineering (ICSE'98), Kyoto, April 1998, Los Alamitos, Kalifornien/USA, IEEE Computer Society Press 1998.

[Balasubramaniyan 98] Jai Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, E. H. Spafford und Diego Zamboni: *An Architecture for Intrusion Detection using Autonomous Agents*. Department of Computer Sciences, Purdue University, Coast TR 98-05, 1998.

[Balfanz 97] Dirk Balfanz und Li Gong: *Experience with Secure Multi-Processing in Java*. Technical Report 560-97, Department of Computer Science, Princeton University, September 1997.

[Baumann 95] Joachim Baumann: Agents: *A Triptychon of Problems.* European Conference on Object-Oriented Programming (ECOOP '95), Workshop on Objects and Agents, Aarhus, 1995.

[Baumann 97a] Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis, Kurt Rothermel und Markus Straßer: *Communication Concepts for Mobile Agent Systems*. In: Kurt Rothermel (Hg.): Proc. First International Workshop on Mobile Agents (MA'97), Berlin, April 1997. Vol. 1219 Lecture Notes in Computer Science, Springer Verlag, Berlin 1997.

[Baumann 97b] Joachim Baumann und Nikolaos Radouniklis: *Agent Groups in Mobile Agent Systems*. In: Distributed Applications and Interoperable Systems, Chapman & Hall, 1997.

[Baumann 98a] Joachim Baumann, Fritz Hohl, Kurt Rothermel und Markus Straßer: *Mole – Concepts of a Mobile Agent System*, World Wide Web, Vol. 1, Nr. 3, S. 123-137. 1998.

[Baumann 98b] Joachim Baumann und Kurt Rothermel: *The Shadow Approach: An Orphan Detection Protocol for Mobile Agents*. Technischer Bericht TR 1998/08, Fakultät Informatik, Universität Stuttgart, 1998.

[Baumann 99a] Joachim Baumann: *A Comparison of Mechanisms for Locating Mobile Agents*. Technischer Bericht TR 1999/11, Fakultät Informatik, Universität Stuttgart, 1999.

[Baumann 99b] Joachim Baumann: *Control Algorithms for Mobile Agents*. Dissertation, Fakultät Informatik, Universität Stuttgart, 1999.

[Beck 97] Bernhard Beck: *Terminierung und Waisenerkennung in einem System mobiler Software-Agenten*. Diplomarbeit, Universität Stuttgart, Fakultät für Informatik, 1997.

[Bellavista 99] Paolo Bellavista, Antonio Corradi und Cesare Stefanelli: *An Open Secure Mobile Agent Framework for Systems Management*. In: Journal of Network and Systems Management (JNSM), Special Issue on Mobile Agent-based Network and Service Management, Vol. 7, No. 3, S. 323-339, September 1999.

[Berghoff 96] Jürgen Berghoff, Oswald Drobnik, Anselm Lingnau und Christian Mönch: *Agent-based configuration management of distributed applications*. In: Proc. 3rd International Conference on Configurable Distributed Systems. 1996.

[Berkovits 98] Shimshon Berkovits, Joshua D. Guttman und Vipin Swarup: *Authentication for Mobile Agents*. In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Binder 02] Walter Binder and Volker Roth: *Secure mobile agent systems using Java – where are we heading?* In: Proc. 17th ACM Symposium on Applied Computing, Special Track on Agents, Interactions, Mobility, and Systems (SAC/AIMS), Madrid/Spanien, März 2002.

[Bic 98] Lubomir F. Bix: *Mobile Network Objects*. In: Encyclopedia of Electrical and Electronics Engineering, John Wiley & Sons, Inc., 1998.

[Bieszczad 98a] Andrzej Bieszczad, Bernard Pagurek und Tony White: *Mobile Agents for Network Management*. In: IEEE Communication Surveys, Fourth Quarter 1998, Vol. 1, No. 1, 1998.

[Bieszczad 98b] Andrzej Bieszczad und Bernard Pagurek: *Application-Oriented Taxonomy of Mobile Code*. In: Proc. IEEE/IFIP Network Operations and Management Symposium NOMS'98, New Orleans, Louisiana/USA, February 1998.

[Binder 01] Walter Binder, Jarle G. Hulaas und Alex Villazon: *Portable Resource Control in Java: Application to Mobile Agent Security.* In: Security of Mobile Multiagent Systems (SEMAS 2001), Fifth International Conference on Autonomous Agents (Agents 2001), Montreal/Kanada, Mai 2001.

[Bouchenak 99] S. Bouchenak: *Pickling threads state in the Java system*. In: Third European Research Seminar on Advances in Distributed Systems (ERSADS '99), Madeira/Portugal, April 1999.

[Bouchenak 00] S. Bouchenak und D. Hagimont: *Approaches to Capturing Java Threads State.* Middleware 2000 (Poster Session), New York/USA, April 2000.

[Bradshaw 96] Jeffrey Bradshaw: *An Introduction to Software Agents.* In: Software Agents. The MIT Press, S. 3-46, 1996.

[Braun 99] Peter Braun: *Über die Migration bei mobilen Agenten*. Jenaer Schriften zur Mathematik und Informatik Nr. 99/13, 1999.

[Breugst 99] Markus Breugst und Sang Choy: *Management of Mobile Agent Based Services*. In: Han Zuidweg, Mario Campolargo, Jaime Delgado, Alvin P. Mullery (Hg.): Intelligence in Services and Networks – Paving the Way for an Open Service Market, 6th International Conference on Intelligence and Services in Networks, IS&N'99, Barcelona/Spanien, April 1999. Vol. 1597 Lecture Notes in Computer Science, Springer, 1999.

[Bryce 99a] Ciarán Bryce und Jan Vitek: *The JavaSeal Mobile Agent Kernel*. In: Proc. Third International Symposium on Mobile Agents (AM/MSA'99), Palm Springs, Oktober 1999.

[Bursell 97] Michael Bursell und Takanori Ugai: *Comparison of autonomous mobile agent technologies*. Technical Report des FollowMe-Projekts, Oktober 1997.

[Busse 98] Ingo Busse, Stefan Covaci und Markus Nietfeld: *Sicherheitsrisiken bei Mobilen Agenten und deren Lösung in einer Java basierten Plattform*. PIK 21 (1998) 3 S. 144-148. München: K. G. Saur Verlag 1998.

[Cabrera 02] João B. D. Cabrera, Wenke Lee, Lundy Lewis und Xinzhou Qin: *Integrating Intrusion Detection and Network Management*. In: NOMS 2002, 2002 IEEE/ IFIP Network Operations and Management.

[Cabri 98] Giacomo Cabri, Letizia Leonardi und Franco Zambonelli: *Mobile Agent Technology: Current Trends and Perspectives*. In: Congresso annuale AICA ’98, Napoli/Italien, November 1998.

[Caglayan 97] Alper Caglayan und Colin Harrison: *Agent Sourcebook – A Complete Guide to Desktop, Internet, and Intranet Agents*. New York u.a.: John Wiley & Sons, Inc. 1997.

[Carzaniga 97] Antonio Carzaniga, Gian Pietro Picco und Giovanni Vigna: *Designing Distributed Applications with Mobile Code Paradigms*. In: Proc. 19th International Conference on Software Engineering, Boston/USA, April 1997.

[Chan 99] Anthony H. W. Chan und Michael R. Lyu: *Security Modeling and Evaluation for the Mobile Code Paradigm*. In: P. S. Thiagarajan, Roland H. C. Yap (Eds.): Advances in Computing Science – ASIAN’99, 5th Asian Computing Science Conference, Phuket/Thailand, December 10-12, 1999. Vol. 1742 Lecture Notes in Computer Science, Springer, 1999.

[Chess 95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Paris und Gene Tsudik: *Itinerant agents for mobile computing*. IBM Research Report RC 20010, IBM, März 1995.

[Chess 98] David M. Chess: *Security Issues in Mobile Code Systems*. In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Condict 96] Michael Condict, Dejan Milojicic, Franklin Reynolds und Don Bolinger: *Towards a world-wide civilization of objects*. In: Proc. Seventh ACM SIGOPS European Workshop, Connemara, Ireland, September 1996.

[Cugola 97a] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco und Giovanni Vigna: *Analyzing mobile code languages*. In: Jan Vitek and Christian Tschudin (Hg.): Mobile Object Systems: Towards the Programmable Internet (MOS’96), Linz/Österreich, Juli 1996. Vol. 1222 Lecture Notes in Computer Science. Berlin: Springer Verlag 1997.

[Cugola 97b] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco und Giovanni Vigna: *A characterization of mobility and state distribution in mobile code languages*. In: Max Mühlhäuser (Hg.): 2nd ECOOP Workshop on Mobile Object Systems: Agents on the Move, Linz/Österreich, Juli 1996. Dpunkt-Verlag 1997.

[Dean 96] Drew Dean, Ed Felten und Dan Wallach: *Java security: From HotJava to Netscape and beyond*. In: Proc. 1996 IEEE Symposium on Security and Privacy, Oakland, Kalifornien/USA, Mai 1996.

[Dean 97] Drew Dean und Edward Felten: *Secure Mobile Code: Where do we go from here?* Accepted paper to the DARPA Workshop on Foundations for Secure Mobile Code Workshop, 26 - 28 March 1997.

[Denning 82] Dorothy Elizabeth Robling Denning: *Cryptography and Data Security.* Reading, Massachusetts u.a.: Addison-Wesley Publishing Company 1982.

[Dickinson 97] Ian Dickinson: *Agent Standards*. Technical Report, Agent Technology Group, Hewlett-Packard Lab., 1997[DMTF-98]Distributed Management Task Force: *Desktop Management Interface Specification, Version 2.0s*. Juni 1998.

[DMTF-02] Distributed Management Task Force: *Specification for CIM Operations over HTTP, Version 1.1.* Mai 2002.

[Dömel 97] Peter Dömel, Anselm Lingnau und Oswald Drobnik: *Mobile Agent Interaction in Heterogeneous Environments*. In: Mobile Agents, First International Workshop, MA'97. Berlin, Januar 1997.

[Dula 97] Tobias Dula: *Mobile Agenten – Security und Anwendungen*. In: Seminar „Kommunikation in verteilten Systemen", Technische Universität Darmstadt, 1997.

[Edjlali 98] Guy Edjlali, Anurag Acharya und Vipin Chaudhary. *History-based access control for mobile code.* In: Proc. Fifth ACM Conference on Computer and Communications Security, San Francisco, Kalifornien/USA, November 1998.

[Falk 99] Rainer Falk: *Verwaltung von Java-2-Zugriffspolitiken.* In: Proc. of Java Informationstage 1999, JIT'99, Düsseldorf, September 1999. Informatik aktuell, Springer, 1999.

[Farmer 96] William M. Farmer, Joshua D. Guttman und Vipin Swarup: *Security for mobile agents: Issues and requirements.* In Proc. 19th National Information Systems Security Conference, S. 591-597, Baltimore, Md., Oktober 1996.

[Feigenbaum 97] John Feigenbaum und Peter Lee: *Trust Management and Proof-Carrying Code in Secure Mobile-Code Applications*. DARPA Workshop on Foundations for Secure Mobile Code, März 1997.

[Fenet 01] Serge Fenet und Salima Hassas: *A distributed Intrusion Detection and Response System Based on Mobile Autonomous Agents Using Social Insects Communication Paradigm*. In: Security of Mobile Multiagent Systems (SEMAS 2001), Fifth International Conference on Autonomous Agents (Agents 2001), Montreal/Kanada, Mai 2001.

[Fischer 99] Dirk Fischer: *Mobile Agenten in Java*. Unveröffentlichtes Manuskript, 1999.

[Fischmeister 99] Sebastian Fischmeister und Wolfgang Lugmayr: *The Supervisor-Worker Pattern*. Pattern Languages of Programs (PLoP'99), Allerton House, IL, August 1999.

[Fong 98] Philip W. L. Fong: *Viewer's discretion: Host security in mobile code systems*. Technical Report SFU CMPT TR 1998-19, School of Computing Science, Simon Fraser University, Burnaby, BC, November 1998.

[Franklin 96] Stan Franklin und Art Graesser: *Is it an Agent or just a Program?: A Taxonomy for Autonomous Agents*. Technical Report, Institute for Intelligent Systems, University of Memphis, 1996.

[Fuggetta 98] Alfonso Fuggetta, Gian Pietro Picco und Giovanni Vigna: *Understanding Code Mobility.* IEEE Transactions on Software Engineering, 24(5), S. 342-361, Mai 1998.

[Fünfrocken 98a] Stefan Fünfrocken: *Integrating Java-Based Mobile Agents into Web Servers under Security Concerns*. In: Proc. Hawaii International Conference on System Sciences, Hawaii 1998.

[Fünfrocken 98b] Stefan Fünfrocken: *Mobile Agenten im Internet*. In: S. Jähnichen (Hg.): Internet Computing: Kollaboratives Arbeiten im Internet auf der Basis von Java-Applets, Online '98 Kongressband, 1998.

[Fünfrocken 98c] Stefan Fünfrocken: *Transparent Migration of Java-based Mobile Agents*. In: Proc. Second International Workshop on Mobile Agents (MA '98), Stuttgart 1998.

[Fünfrocken 99a] Stefan Fünfrocken und Friedemann Mattern: *Mobile Agents as an Architectural Concept for Internet-based Dirtributed Applications – The WASP Project Approach*. In: Kommunikation in Verteilten Systemen (KiVS'99), Darmstadt, März 1999.

[Fünfrocken 99b] Stefan Fünfrocken: *Protecting Mobile Web-Commerce Agents with Smartcards*. In: Proc. Third Workshop on Mobile Agents (MA '99), Palm Springs, 1999.

[Füssl 01] Thomas Füssl: *Ein dezentrales, selbstorganisierendes Informationsmanagementsystem*. Dissertation, Fakultät für Elektrotechnik und Informationstechnik, Technische Universität München, 2001.

[Ghanea 01] Robert Ghanea-Hercock und Ian Gifford: *Top-Secret Multi-Agent Systems*. In: Security of Mobile Multiagent Systems (SEMAS 2001), Fifth International Conference on Autonomous Agents (Agents 2001), Montreal, Kanada, Mai 2001.

[Ghezzi 97] Carlo Ghezzi und Giovanni Vigna: *Mobile Code Paradigms and Technologies: A Case Study.* In: Kurt Rothermel (Hg.): Proc. First International Workshop on Mobile Agents (MA'97), Berlin, April 1997. Vol. 1219 Lecture Notes in Computer Science, Springer Verlag, Berlin 1997.

[Glissmann 99] Jörg Glissmann: *Einsatz von Mobilen Agenten zur Leistungsmessung in Rechnernetzen – Mobiler Agent für die Durchführung von passiven Leistungsmessungen*. Interdisziplinäres Projekt, Lehrstuhl für Datenverarbeitung, Techn. Univ. München, 1999.

[Goldberg 98] A. Goldberg: *A specification of java loading and bytecode verification*. In: 5th ACM Conference on Computer and Communications Security, pages ?-? ACM Press, San Francisco, California, November 1998.

[Gong 97] Li Gong, Marianne Mueller, Hemma Prafullchandra und Roland Schemers: *Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2*. In: Proc. USENIX Symposium on Internet Technologies and Systems, Monterey, Kalifornien/USA, Dezember 1997.

[Gong 98a] Li Gong und Roland Schemers: *Signing, Sealing, and Guarding Java Objects.* In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Gong 98b] Li Gong: *Secure Java Class Loading*. In: IEEE Internet Computing, Vol. 2, No. 6, November/Dezember 1998.

[Görl 98] Harald Görl: *Realisierung der Sicherheit mobiler Agenten in unterschiedlichen Agentensystemen.* Diplomarbeit am Institut für Informatik, Techn. Univ. München, 1998.

[Görl 99] Harald Görl: *Einsatz von Mobilen Agenten zur Leistungsmessung in Rechnernetzen – Graphische Bedienoberfläche*. Interdisziplinäres Projekt, Lehrstuhl für Datenverarbeitung, Techn. Univ. München, 1999.

[Gray 95a] Robert S. Gray: *Agent TCL: Alpha Release 1.1*. Dokumentation zu D'Agents 1995.

[Gray 96a] Robert Gray: *Security in Mobile-Agent Systems*. Präsentation an der Universität Leiden, 1996.

[Gray 96b] Robert Gray, George Cybenko, David Kotz und Daniela Rus: *Agent Tcl*. In: Itinerant Agents: Explanation and Examples with CD-ROM. Manning Publishing 1996

[Gray 97a] Robert S. Gray, David Kotz, Saurab Nog, Daniela Rus und George Cybenko. *Mobile agents for mobile computing*. In: Proc. Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis, Fukushima/ Japan, März 1997.

[Gray 97b] Robert S. Gray. *Agent Tcl: A flexible and secure mobile-agent system.* Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, Januar 1998. Ph.D. Thesis, Juni 1997.

[Gray 98a] Robert S. Gray, David Kotz, George Cybenko und Daniele Rus: *D'Agents: Security in a Multiple-Language, Mobile-Agent System.* In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998

[Gray 00] Robert S. Gray, George Cybenko, David Kotz und Daniela Rus: *Mobile agents: Motivations and State of the Art*. In Jeffrey Bradshaw (Hg.): Handbook of Agent Technology, AAAI/MIT Press, 2000.

[Green 97] Shaw Green und Fergal Somers: *Software agents: A review.* Technical report, Intelligent Agent Group, Trinity College/Irland, Mai 1997.

[Greenberg 98] Michael S. Greenberg, Jennifer C. Byington und David G. Harper: *Mobile Agents and Security.* In: IEEE Communications Magazine, Vol. 36, No. 7, Juli 1998, S. 76-85

[Gritzalis 98b] Stefanos Gritzalis und George Aggelis: *Security Issues Surrounding Programming Languages for Mobile Code: Java vs. Safe-Tcl*. Operating Systems Review 32(2): 16-32 (1998)

[Gruschke 99] Boris Gruschke, Stephen Heilbronner und Helmut Reiser: *Mobile Agent System Architecture – Eine Plattform für flexibles IT-Management*. Technischer Bericht 9902. Institut für Informatik, Ludwig-Maximilians-Universität München, August 1999.

[Gypsy 00] Gypsy Development Team: *Gypsy User Guide*. Technical University of Vienna, 2000.

[Hagimont 97] Daniel Hagimont und L. Ismail: *A Protection Scheme for Mobile Agents on Java*. In: Proc. Third ACM/IEEE Int. Conf. on Mobile Computing and Networking (MobiCom'97), Budapest, September 1997

[Harrison 95] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. *Mobile agents: Are they a good idea?* IBM Research Report. März 1995.

[Hawblitzel 98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu und Thorsten von Eicken. *Implementing multiple protection domains in Java*. In: Proc. 1998 USENIX Annual Technical Conference, New Orleans, LA., Juni 1998.

[Hegering 99] Heinz-Gerd Hegering, Sebastian Abeck und Bernhard Neumair: *Integriertes Management vernetzter Systeme. Konzepte, Architekturen und deren betrieblicher Einsatz.* Heidelberg: Dpunk-Verlag 1999.

[Heldwein 98] Christian Heldwein: *Einsatz von Mobilen Agenten zur Leistungsmessung in Rechnernetzen – Mobiler Agent für die Durchführung von aktiven Leistungsmessungen*. Interdisziplinäres Projekt, Lehrstuhl für Datenverarbeitung, Techn. Univ. München, 1998.

[Helmer 98] Guy Helmer, Johnny S. K. Wong, Vasant Honavar und Les Miller: *Intelligent Agents for Intrusion Detection*. In: Proc. IEEE Information Technology Conference, Syracuse, NY, September 1998.

[Henne 98] Thomas Henne: *Plattform für mobile Agenten im Konfigurationsmanagement*. Interdisziplinäres Projekt, Lehrstuhl für Datenverarbeitung, Techn. Univ. München, 1999.

[Hohl 97a] Fritz Hohl: *An approach to solve the problem of malicious hosts*. Universität Stuttgart, Fakultät Informatik, Fakultätsbericht Nr. 1997/03.

[Hohl 97b] Fritz Hoh, Peter Klar und Joachim Baumann: *Efficient Code Migration for Modular Mobile Agents*. Universität Stuttgart, Fakultät Informatik, Fakultätsbericht Nr. 1997/06.

[Hohl 98a] Fritz Hohl: *Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts*. In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Hohl 98c] Fritz Hohl: *A Model of Attacks of Malicious Hosts Against Mobile Agents.* In: 4th ECOOP Workshop on Mobile Object Systems (MOS'98): Secure Internet Mobile Computations. 1998.

[Hohl 99] Fritz Hohl und Kurt Rothermel: *A Protocol for Preventing Blackbox Tests of Mobile Agents*. In: Kommunikation in Verteilten Systemen (KiVS'99), Darmstadt, März 1999.

[Hohl 00] Fritz Hohl, Joachim Baumann, Kurt Rothermel, Markus Schwehm, Markus Straßer und Wolfgang Theilmann: *AIDA II – Abschlußbericht*. Bericht Nr. 2000/04, Universität Stuttgart, Fakultät Informatik, März 2000.

[Humphries 00] Jeffrey W. Humphries, Curtis A. Carver Jr. und Udo W. Pooch: *Secure Mobile Agents for Network Vulnerability Scanning.* In: Proc. 2000 IEEE Workshop on Information Assurance and Security, United States Military Academy, West Point, NY/USA, Juni 2000.

[IKV 01] IKV++ GmbH: *Grasshopper Basics And Concepts*. Dokumetation zu Grasshopper 2.2. Berlin 2001.

[ISO IS9596] International Organization for Standardization: *ISO IS9596, Management information protocol specification – Part 2: Common management information protocol*. Januar 1990.

[Jansen 99a] Wayne Jansen und Tom Karygiannis: *Mobile Agent Security*. NIST Special Publication 800-19. National Institute of Standards and Technology, August 1999.

[Jansen 99b] Wayne Jansen, Peter Mell, Tom Karygiannis und Don Marks: *Applying Mobile Agents to Intrusion Detection and Response*. NISTIR 6416, National Institute of Standards and Technology September 1999.

[Jansen 00] Wayne Jansen: *Countermeasures for Mobile Agent Security*. Computer Communications, Special Issue on Advanced Security Techniques for Network Protection, Elsevier Science BV, November 2000.

[Jansen 01] Wayne Jansen: *A Privilege Management Scheme for Mobile Agent Systems*. In: Security of Mobile Multiagent Systems (SEMAS 2001), Fifth International Conference on Autonomous Agents (Agents 2001), Montreal/ Kanada, Mai 2001.

[Jazayeri 00] Mehdi Jazayeri und Wolfgang Lugmayr: *Gypsy: A Component-based Mobile Agent System*. Accepted at the 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000) Rhodos/Griechenland, Januar 2000.

[Johansen 95] Dag Johansen, Robbert van Renesse und Fred B. Schneider: *An introduction to the TACOMA distributed system*. Technical Report 95-23, Department of Computer Science, University of Tromsø, Tromsø/Norwegen, Juni 1995.

[Johansen 98a] Dag Johansen, Fred B. Schneider, and Robbert van Renesse: *What tacoma taught us*. In: Dejan Milojicic, Frederick Douglis und Richard Wheeler (Hg.): Mobility, Mobile Agents and Process Migration – An edited Collection. Addison-Wesley, 1998.

[Karjoth 97] Günter Karjoth, Danny B. Lange und Mitsuru Oshima: *A Security Model for Aglets*. In: IEEE Computing 1089-7801/1997.

[Karnik 98] Neeran M. Karnik: *Security in Mobile Agent Systems*. Ph.D. Dissertation, Department of Computer Science and Engineering, University of Minnesota 1998.

[Karnik 99a] Neeran M. Karnik und Anand R. Tripathi: *Security in the Ajanta Mobile Agent System*. Technical Report, Department of Computer Science and Engineering, University of Minnesota 1999.

[Karnik 99b] Neeran M. Karnik und Anand R. Tripathi: *Design Issues in Mobile Agent Programming Systems*. In: IEEE Concurrency, Juli-September 1998, S. 52-61.

[Karnik 01] Neeran Karnik und Anand Tripathi: *Security in the Ajanta Mobile Agent System*. Software – Practice and Experience, Januar 2001.

[Kassab 98] Lora L. Kassab, und Jeffrey Voas: *Agent Trustworthiness*. In: Proc. ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, S. 121-133, INRIA, Frankreich, 1998.

[Kato 97] K. Kato: *Safe and Secure Execution Mechanisms for Mobile Objects*. In: Jan Vitek und Christian Tschudin (Hg.): Mobile Object Systems: Towards the Programmable Internet, S. 201-211. Springer-Verlag, April 1997. Vol. 1222 Lecture Notes in Computer Science 1997.

[Kiniry 97] Joseph Kiniry und Daniel Zimmermann. *A Hands-On Look at Java Mobile Agents*. In: IEEE Internet Computing, Vol. 1, Nr. 4 (Juli/August 1997), S. 21-30.

[Knoll 01] Gerald Knoll, Niranjan Suri und Jeffrey M. Bradshaw: *Path-based Security for Mobile Agents*. In: Security of Mobile Multiagent Systems (SEMAS 2001), Fifth Int. Conf. on Autonomous Agents (Agents 2001), Montreal/ Kanada, Mai 2001.

[Konopka 95] Robert Konopka und Markus Trommer: *A Multilayer Architecture for SNMP-Based, Distributed and Hierarchical Management of Local Area Networks*. In: Proc. Fourth IEEE International Conference on Computer Communications and Networks, ICCCN'95, Las Vegas, Nevada/USA, September 1995.

[Korba 99] Larry Korba: *Towards Secure Agent Distribution and Communication*. In: Porc. 32nd Hawaii International Conference on System Sciences 1999.

[Kotz 96] David Kotz, Robert S. Gray and Daniela Rus: *Transportable Agents Support Worldwide Applications*. In: Proc. Seventh ACM SIGOPS European Workshop, S. 41-48, September 1996. ACM Press.

[Kotz 99] David Kotz und Robert S. Gray: *Mobile Agents and the Future of the Internet*. In: ACM Operating Systems Review, 33(3), S. 7-13, August 1999

[Kotzanikolaou 00] Panayiotis Kotzanikolaou, Mike Burmester und Vassilios Chrissikopoulos: *Secure Transactions with Mobile Agents in Hostile Environments*. In: E. Dawson, A. Clark und C. Boyd (Hg.): Information Security and Privacy. Proc. 5th Australasian Conference, ACISP 2000. Vol. 1841 Lecture Notes in Computer Science, Springer-Verlag 2000.

[Lange 98a] Danny B. Lange: *Mobile Objects and Mobile Agents: The Future of Distributed Computing?* In: Proc. European Conference on Object-Oriented Programming ’98, 1998.

[Lange 98b] Danny B. Lange und Mitsuru Oshima: *Mobile Agents with Java: The Aglet API*. World Wide Web Journal, 1998.

[Lange 99] Danny B. Lange und Mitsuru Oshima: *Seven Good Reasons for Mobile Agents*. Communications of the ACM, March 1999.

[Lingnau 95] Anselm Lingnau, Oswald Drobnik und Peter Dömel: *An HTTP-based Infrastructure for Mobile Agents*. In: Proc. 4th Int. World Wide Web Conference, Boston, Dezember 1995.

[Lingnau 98] Anselm Lingnau und Oswald Drobnik: *Agent – User Communications: Requests, Results, Interaction.* In: Mobile Agents: Second International Workshop, MA ’98. Stuttgart. September 1998.

[Lugmayr 99] Wolfgang Lugmayr: *Gypsy: A Component-Oriented Mobile Agent System*. Dissertation, Distributed Systems Group, Technical University of Vienna, Austria. October 1999.

[Marques 99] PauloJorge Marques, Luis Moura Silva und João Gabriel Silva: *Security Mechanisms for using Mobile Agents in Electronic Commerce*. In: Proc. Eighteenth Symposium on Reliable Distributed Systems, Oktober 1999, Lausanne/Schweiz. IEEE Computer Society, 1999.

[McManis 96] Chuck McManis: *The basics of Java class loaders*. JavaWorld 1996.

[Meadows 97] Catherine Meadows: *Detecting Attacks on Mobile Agents*. DARPA Workshop on Foundations for Secure Mobile Code, März 1997.

[Milojicic 98a] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran und J. White. *MASIF: The OMG Mobile Agent System Interoperability Facility.* In: Kurt Rothermel (Hg.): Proc. Second International Workshop on Mobile Agents (MA'98), Stuttgart, September 1998. Vol. 1477 Lecture Notes in Computer Science, Springer Verlag, Berlin 1998.

[Milojicic 98b] Dejan Milojicic, Gul Agha, Philippe Bernadat, Deepika Chauhan, Shai Guday, Nadeem Jamali und Dan Lambright: *Case Studies in Security and Resource Management for Mobile Objects*. In: Proc. ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, S. 191-205, INRIA, Frankreich, 1998.

[Milojicic 98c] Dejan Milojicic, William LaForge und Deepika Chauhan: *Mobile Objects and Agents, Design, Implementation and Lessons Learned*. In: Distributed Systems Engineering, IEE, 5 (1988), S. 1-14.

[Misikangas 00] Pauli Misikangas und Kimmo Raatikainen: *Agent Migration Between Incompatible Platforms*. In: 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan/China, April 2000.

[Mitsubishi 98] Mitsubishi: *Mobile Agent Computing*. Whitepaper http://www.meitca.com/HSL/Projects/Concordia/documents.htm

[Moore 98] Jonathan T. Moore: *Mobile Code Security Techniques*. University of Pennsylvania, Department of Computer and Information Science, Technical Report MS-CIS-98-28, Mai 1998.

[Nagaratnam 98] Nataraj Nagaratnam und Doug Lea: *Role-Based Protection and Delegation for Mobile Object Environments*. In: Proc. ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, S. 157-170, INRIA, Frankreich, 1998.

[Necula 98] George C. Necula und Peter Lee: *Safe, Untrusted Agents Using Proof-Carrying Code.* In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Nwana 96] Haycinth S. Nwana: *Software agents: An overview.* Knowledge Engineering Review, 11(3): S. 205-244, 1996.

[OMG 97] The Open Group: *Mobile Agent Facility Specification*.

[OMG 01] Object Management Group: *OMG – Unified Modeling Language, v1.4*. September 2001.

[OMG 02] Object Management Group: *The Common Request Broker: Architecture and Specification, Version 3.0*. Juli 2002.

[Ordille 96] Joann Ordille: *When agents roam, who can you trust?* In: Proc. First Conference on Emerging Technologies and Applications in Communications, Portland, Mai 1996 .

[Oshima 98] Mitsuru Oshima, Guenter Karjoth und Kouichi Ono: *Aglets Specification 1.1 Draft*. IBM 1998. http://www.trl.ibm.co.jp/aglets/spec11.html

[Ousterhout 98] John K. Ousterhout, Jacob Levy und Brent B. Welch: *The Safe-Tcl Security Model.* In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Peine 97a] Holger Peine: *An Introduction to Mobile Agent Programming and the Ara System*. ZRI report 1/97, Dept. of Computer Science, Universität Kaiserslautern, 1997.

[Peine 97b] Holger Peine und Torsten Stolpmann: *The Architecture of the Ara Platform for Mobile Agents*. In: Kurt Rothermel (Hg.): Proc. First International Workshop on Mobile Agents (MA'97), Berlin, April 1997. Vol. 1219 Lecture Notes in Computer Science, Springer Verlag, Berlin 1997.

[Peine 98] Holger Peine: *Security Concetps and Implementation in the Ara Mobile Agent System*. 7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, Stanford University/USA, Juni 1998

[Pietro 99] E. Di Pietro, O. Tomarchio, G. Iannizzotto und M. Villari: *Experiences in the use of Mobile Agents for developing distributed applications*. In: Sistemi distribuiti: Algoritmi, Architetture e Linguaggi (WSDAAL'99), L'Aquila/ Italien, September 1999.

[Pham 98] Vu Anh Pham und Ahmed Karmouch: *Mobile Software Agents: An Overview.* In: IEEE Communications Magazine, Vol. 36, No. 7, Juli 1998, S. 76-85.

[Proell 98] Thomas Pröll: *Einsatz von mobilen Agenten im Konfigurationsmanagement – Mobiler Agent zur dynamischen Messung von Antwortzeiten im Netz.* Interdisziplinäres Projekt, Lehrstuhl für Datenverarbeitung, Techn. Univ. München, 1998.

[Puliafito 97] A. Puliafito, O. Tomarchio und L. Vita. *MAP: Design and Implementation of a Mobile Agents Platform*. Technical Report TR-CT-9712, University of Catania, 1997.

[Puliafito 99] A. Puliafito and O. Tomarchio: *Advanced Network Management Functionalities through the use of Mobile Software Agents*. In: 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA'99), Stockholm/Schweden, August 1999.

[Puliafito 00] A. Puliafito und O. Tomarchio: *Using Mobile Agents to implement flexible Network Management strategies*. Computer Communication Journal. To be published.

[Reiser 00a] Helmut Reiser und Gerald Vogt: *Security Requirements for Management Systems using Mobile Agents*. In: Proc. Fifth IEEE Symposium on Computers and Communications, ISCC 2000, Antibes/Frankreich, Juli 2000.

[Reiser 00b] Helmut Reiser und Gerald Vogt: *Threat Analysis and Security Architecture of Mobile Agent based Management Systems*. In: Proc. NOMS 2000 IEEE/IFIP Network Operations and Management Symposium „The Networked Planet: Management Beyond 2000“, Honolulu, Hawaii/USA, April 2000.

[RFC 1157] J. D. Case, M. Fedor, M. L. Shoffstall und C. Davin: *Simple network management protocol (RFC 1157)*. DDN Network Information Center, SRI International, Mai 1990.

[RFC 1757] S. Waldbusser: *Remote Network Monitoring Management Information Base (RFC 1757).* 1995.

[RFC 2246] T. Dierks und C. Allen: *The TLS Protocol Version 1.0 (RFC 2246)*. 1999.

[RFC 3165] D. Levi und J. Schönwälder: *Definitions of Managed Objects for the Delegation of Management Scripts (RFC 3165)*. 2001.

[Reitzig 99] Jochen Reitzig: *Messung der Netzbelastung beim Einsatz mobiler Agenten im Netzmanagement*. Diplomarbeit, Lehrstuhl für Datenverarbeitung, Techn. Univ. München, 1999.

[Rölle 99] Harald Rölle: *Authentisierung und Autorisierung für das Java/CORBA-Agentensystem MASA*. Diplomarbeit am Institut für Informatik der Techn. Univ. München 199.9

[Roth 98] Volker Roth und Mehrdad Jalali-Sohi: *Access Control and Key Management for Mobile Agents*. In: Computers & Graphics, 1998, 22(3), Special Issue Data Security in Image Communication and Networks, 1998.

[Roth 01a] Volker Roth: *Programming Satan's Agents*. In: Security of Mobile Multiagent Systems (SEMAS 2001), Fifth International Conference on Autonomous Agents (Agents 2001), Montreal, Kanada, Mai 2001.

[Roth 01b] Volker Roth: *On the Robustness of Some Cryptographic Protocols for Mobile Agent Protection*. In Proc. Mobile Agents 2001, Atlanta/USA Dezember 2001. Lecture Notes in Computer Science, Springer-Verlag 2001.

[Roth 01c] Volker Roth und Jan Peters: *A Scalable and Secure Global Tracking Service for Mobile Agents*. In: Proc. Mobile Agents 2001, Atlanta/USA Dezember 2001. Lecture Notes in Computer Science, Springer Verlag 2001.

[Rothermel 97a] Kurt Rothermel, Fritz Hohl und Nikolaos Radouniklis: *Mobile Agent Systems: What is Missing?* In: H. König, K. Geihs and T. Preuß (eds.) Distributed Applications and Interoperable Systems (DAIS'97), Chapman & Hall, S. 111-124. 1997.

[Rothermel 97b] Kurt Rothermel: *Mobile Software-Agenten: Chancen und Risiken*. In: Kommunikation in Verteilten Systemen (KiVS'99), Darmstadt, März 1999.

[Sahai 98] Akhil Sahai und Christine Morin: *Enabling a Mobile Network Manager (MNM) Through Mobile Agents*. In: Kurt Rothermel (Hg.): Proc. Second International Workshop on Mobile Agents (MA'98), Stuttgart, September 1998. Vol. 1477 Lecture Notes in Computer Science, Springer Verlag, Berlin 1998

[Sander 97a] Tomas Sander und Christian F. Tschudin: *Towards Mobile Cryptography.* Technical Report 97-049, International Computer Science Institute, Berkeley. 1997.

[Sander 97b] Tomas Sander: *On the Cryptographic Protection of Mobile Code*. Vortrag am Workshop on Mobile Agents and Security, UMBC, Oktober 1997.

[Sander 97c] Tomas Sander: *Security! or "How to Avoid to Breath Life in Frankensteins Monster"*. Vortrag am ICSI-Workshop über Auto Mobile Code Documentation 1997.

[Sander 98a] Tomas Sander und Christian F. Tschudin: *Protecting Mobile Agents Against Malicious Hosts*. In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Sander 98b] Tomas Sander und Christian F. Tschudin: *On Sofware Protection via Function Hiding*. In: D. Aucsmith (Hg.): Information Hiding II. Proc. Second International Workshop, IH'98. Springer-Verlag 1998.

[Satoh 02] Ichiro Satoh: *A Framework for Building Reusable Mobile Agents for Network Management*. In: NOMS 2002, 2002 IEEE/IFIP Network Operations and Management Symposium. Florenz/Italien, April 2002.[Simões 02]P. Simões, J. Rodrigues, L. Silva und F. Boavida: *Distributed Retrieval of Management Information: Is It About Mobility, Locality or Distribution?* In: NOMS 2002, 2002 IEEE/IFIP Network Operations and Management Symposium. Florenz, Italien, 15.-19. April 2002.

[Schelderup 99] Kristian Schelderup, Jon Ølnes: *Mobile Agent Security – Issues and Directions*. In: Han Zuidweg, Mario Campolargo, Jaime Delgado, Alvin P. Mullery (Hg.): Intelligence in Services and Networks – Paving the Way for an Open Service Market, 6th International Conference on Intelligence and Services in Networks, IS&N'99, Barcelona/Spanien, April 1999. Proceedings. Vol. 1597 Lecture Notes in Computer Science, Springer, 1999.

[Sidor 98] David J. Sidor: *TMN Standards: Satisfying Today's Needs While Preparing for Tomorrow.* IEEE Communications Magazine, Vol. 36, No. 3, S. 54-64, März 1998.

[Silva 98a] Alberto Silva und José Delgado: *The Agent Pattern for Mobile Agent Systems*. Proc. Third European Conference on Pattern Languages of Programming and Computing, EuroPLoP'98, Irsee/Deutschland, Juli 1998.

[Straßer 96] Markus Straßer, Joachim Baumann und Fritz Hohl. *Mole – a Java based mobile agent system.* In: 2nd ECOOP Workshop on Mobile Object Systems, S. 28-35, Linz/Österreich, Juli 1996.

[Sudmann 98] Nils P. Sudmann: *Position Paper: Security in Tacoma*. In: Proc. ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, S. 155-156, INRIA, France, 1998.

[Sugauchi 99] Kiminori Sugauchi, Satoshi Miyazaki, Stefan Covaci und Tianning Zhang: *Efficiency Evaluation of a Mobile Agent Based Network Management System*. In: Han Zuidweg, Mario Campolargo, Jaime Delgado, Alvin P. Mullery (Hg.): Intelligence in Services and Networks – Paving the Way for an Open Service Market, 6th International Conference on Intelligence and Services in Networks, IS&N'99, Barcelona/Spanien, April 1999, Proceedings. Vol. 1597 Lecture Notes in Computer Science, Springer, 1999.

[Susilo 98] G. Susilo, Andrzej Bieszczad und Bernard Pagurek: *Infrastructure for Advanced Network Management based on Mobile Code*. In: Proc. IEEE/IFIP Network Operations and Management Symposium NOMS'98, New Orleans, Louisiana/USA, Februar 1998.

[Swarup 97] Vipin Swarup: *Trust Appraisal and Secure Routing of Mobile Agents*. Accepted paper to the DARPA Workshop on Foundations for Secure Mobile Code Workshop, März 1997.

[Tardo 96] Joseph Tardo und Luis Valenta: *Mobile Agent Security and Telescript*. In: Proc. IEEE COMPCON '96, Februar 1996.

[Theilmann 99] Wolfgang Theilmann und Kurt Rothermel: *Efficient Dissemination of Mobile Agents.* Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart, 1999.

[Thorn 97] Tommy Thorn: *Programming languages for mobile code*. In: ACM Computing Surveys, 29(3), S. 213-239, September 1997.

[Tripathi 01] Anand Tripathi, Tanvir Ahmed und Neeran Karnik: *Experiences and Future Challenges in Mobile Agent Programming*. Microprocessor and Microsystems 2001.

[Tripathi 02a] Anand Tripathi, Tanvir Ahmed, Sumedh Pathak, Megan Carney und Paul Dokas: *Paradigms for Mobile Agent Based Active Monitoring of Network Systems*. In: NOMS 2002, 2002 IEEE/IFIP Network Operations and Management Symposium. Florenz/Italien, April 2002.

[Tripathi 02b] Anand R. Tripathi, Neeran M. Karnik, Tanvir Ahmed, Ram D. Singh, Arvind Prakash, Vineet Kakani, Manish K. Vora und Mukta Pathak: *Design of the Ajanta System for Mobile Agent Programming*. Journal of Systems and Software. Volume 62, Issue 2, S. 123-140, May 2002.

[Trommer 96] Markus Trommer und Robert Konopka: *Verteilung von Netzmanagement-aufgaben mit Hilfe einer dezentralen, hierarchischen Mehrschichten-architektur.* In: Proc. of ITG/GI/GMA-Fachtagung Softwaretechniken in Automation und Kommunikation, STAK'96, München, März 1996.

[Trommer 99] Markus Trommer: *Mobile Agenten im Netzmanagement*. Fachgespräch „Systemmanagement, Java, mobile Agenten“ an der Techn. Univ. München. Unveröffentlicht.

[Utermann 00] Heike Utermann: *Integration heterogener Datenquellen in ein agenten-basiertes Informationsmanagementsystem*. Dissertation, Fakultät für Elektrotechnik und Informationstechnik, Techn. Univ. München, 2001.

[Venners 97] Bill Venners: *Inside the Java Virtual Machine.* McGraw-Hill 1997

[Vigna 97] Giovanni Vigna: *Protecting Mobile Agents through Tracing.* In: Proc. Third Workshop on Mobile Object Systems, Finland, Juni 1997.

[Vigna 98a] Giovanni Vigna: *Cryptographic Traces for Mobile Agents*. In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Vigna 98b] Giovanni Vigna: *Mobile Code Technologies, Paradigms, and Applications*. PhD Thesis, Politecnico di Milano/Italien, Februar 1998.

[Vitek 97a] Jan Vitek: *Secure object spaces*. In: Max Mühlhaüser (Hg.): Special Issues in Object-Oriented Programming. Workshop Reader of the 10th European Conference on Object-Oriented Programming (Ecoop'96). S. 340-347. Dpunkt-Verlag 1997.

[Vitek 97b] Jan Vitek, Manuel Serrano und Dimitri Thanos: *Security and Communication in Mobile Object Systems*. In: Jan Vitek und Christian Tschudin (Hg.): Mobile Object Systems: Towards the Programmable Internet, S. 177-199. Springer-Verlag, April 1997. Vol. 1222 Lecture Notes in Computer Science 1997.

[Volpano 98] Dennis Volpano und Geoffrey Smith: *Language Issues in Mobile Program Security*. In: Giovanni Vigna (Hg.): Mobile Agents and Security. Berlin u.a.: Springer-Verlag 1998.

[Wagner 96] David Wagner und Bruce Schneier: *Analysis of the SSL 3.0 protocol (revised version)*. In: Proc. 2nd USENIX Workshop on Electronic Commerce, November 1996.

[Wallach 97] Dan S. Wallach, Dirk Balfanz, Drew Dean und Edward W. Felten: *Extensible Security Architectures for Java*. In: 16th Symposium on Operating Systems Principles, Saint-Malo/Frankreich, Oktober 1997.

[Wallach 98] Dan S. Wallach und Edward W. Felten: *Understanding Java Stack Inspection*. In: Proc. 1998 IEEE Symposium on Security and Privacy, Oakland, Kalifornien/USA, Mai 1998.

[Walsh 98] Tom Walsh, Noemi Paciorek und David Wong: *Security and Reliability in Concordia*. In: 31st Annual Hawai'i International Conference on System Sciences (HICSS31), Kona, Hawaii/USA, Januar 1998.

[Wang 00] Chenxi Wang, Jonathan Hill, John Knight und Jack Davidson: *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Technical Report CS-2000-12, Department of Computer Science, University of Virginia, 2000.

[White 96] James E. White. *Telescript technology: Mobile agents*. In: Jeffrey Bradshaw (Hg.): Software Agents, AAAI Press/The MIT Press, 1996.

[White 98] Tony White, Andrzej Bieszczad, B. Pagurek, G. Sugar und X. Tran: *Intelligent Network Modeling using Mobile Agents*. In: Proc. IEEE Global Telecommunications Conference GLOBECOM '98, November 1998, S. 1082-1087.

[Wilhem 98] Uwe G. Wilhelm und Sebastian Staamann: *Protecting the Itinerary of Mobile Agents*. In: Proc. ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, pp 135 - 145, INRIA, Frakreich, 1998.

[Wilhelm 99] Uwe Georg Wilhelm: *A Technical Approach to Privacy based on Mobile Agents Protected by Tamper-resistant Hardware*. PhD-Dissertation, École Polytechnique fédérale de Lausanne, 1999.

[Wong 97] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young und Bill Peet: *Concordia: An Infrastructure for Collaborating Mobile Agents*. In: First International Workshop on Mobile Agents 97 (MA'97), Berlin, April 1997.

[Wong 99] David Wong, Noemi Paciorek und Dana Moore: *Java-based Mobile Agents*. In: Comunications of the ACM, Vol 42, Nr. 3, März 1999, S. 92-102.

[Yee 97] Bennet Yee: *A Sanctuary for Mobile Agents.* Accepted paper to the DARPA Workshop on Foundations for Secure Mobile Code Workshop, März 1997.

[Yellin 97] Frank Yellin: *Low Level Security in Java*. In: Proc. Fourth International World Wide Web Conference, Vol. 1, Issue 1, 1996.

[Zapf 99a] Michael Zapf, Klaus Herrmann und Kurt Geihs: *Decentralized SNMP Management with Mobile Agents*. In: Proc. Sixth IFIP/IEEE International Symposium on Integrated Network Management (IM'99), Boston MA/USA, Mai 1999

[Zapf 99b] Michael Zapf, Klaus Herrmann und Kur Geihs: *Netz- und Systemmanagement mit mobilen Agenten und SNMP.* In: Management verteilter IV-Systeme, 13. GI-Fachtagung über den Betrieb von Informations- und Kommunikationssystemen, Seeheim-Jugenheim, April 1999.

Anhang

A Konfigurationsdateien von MobMan

Die Konfiguration ist innerhalb der MobMan-Architektur in einem hierarchischen Objektbaum abgelegt. Die Objekte sind Instanzen der `MonManConfig`-Klasse, diese enthält auch einen Parser für die Textform der Konfiguration. Über verschiedene Methoden kann die Objekthierarchie bequem ausgewertet werden, Konfigurationen lassen sich zur Laufzeit mischen, ineinander einhängen, ergänzen, kopieren oder löschen.

A.1 Format

Die `MobManConfig`-Dateien bestehen aus Parameter-Wert-Paaren, die hierarchisch geordnet sind. Alle Parameter, Werte und Verzweigungen sind Text-Strings. Die Wertangaben können auch fehlen (werden durch einen leeren String ersetzt), womit Aufzählungen möglich sind. Die Reihenfolge bleibt dabei erhalten.

Verzweigungen sind durch eine geschweifte Klammer gekennzeichnet (`{`). Der Name des Asts steht direkt vor der Klammer; der Unterbaum erstreckt sich bis zur passenden schließenden Klammer. Ein Gleichheitszeichen (`=`) verbindet den Parameter und seinen Wert. Weitere Parameter (und optional auch Werte) sind durch Whitespace getrennt, also Leerzeichen, Tabulator oder Zeilenumbruch. Ab einem Kommentarzeichen (`#` oder `//`) wird der Rest der Zeile ignoriert. Bereiche, die in Anführungszeichen gesetzt sind (`"`), gelten als ein Wort. Ein Backslash (`\`) vor einem Zeichen hebt dessen Sonderbedeutung auf.

Es gibt keine globale Regel, welche Einträge in einer Konfiguration möglich sind. Dies entscheidet das jeweilige Werkzeug, das die Konfiguration aus der Datei benutzt. Häufig werden Unterbäume an Agenten oder Plugins weitergegeben, die selbst bestimmen, welche Optionen sie erwarten.

Folgende Pseudo-EBNF-Darstellung spezifiziert die Syntax. Leerzeichen sind dabei absichtlich nicht erwähnt, um die Darstellung zu vereinfachen.

```
Config     ::= {Statement} .
Statement  ::= Assign | Word | Nest | Comment .
Assign     ::= Word Equal Word .
Nest       ::= Word OpenBrace {Statement} CloseBrace .
Comment    ::= "#" {Char} EOL | "//" {Char} EOL .
Equal      ::= "=" .
OpenBrace  ::= "{" .
CloseBrace ::= "}" .
Word       ::= WordChar {WordChar} | Quotation Char {Char} Quotation .
WordChar   ::= <beliebiges Zeichen, das keine Sonderbedeutung hat, auch
               keine Leerzeichen; oder: durch Backslash maskiert>
Char       ::= <beliebiges Zeichen>
EOL        ::= <Ende der Zeile>
Backslash  ::= "\" .
Quotation  ::= """ .
```

A.2 Beispiel

a) launch.conf

Die Datei `launch.conf` konfiguriert den Start von Agenten (siehe Anhang B *Das Launch-Programm*, Seite 152). Launch liest diese Konfigurationsdatei, startet Agenten und migriert sie auf ihre erste Plattform. Das folgende Beispiel ist sehr ausführlich gehalten. Es konfiguriert eine Reihe von Test-Agenten, ebenso den Pia Service Agent, der Pia-Module auf den Plattformen verteilt. Die Namen der Parameter sind weitgehend selbsterklärend gehalten. Wo nötig erläutern Kommentare die Einträge.

```
Destination {
  host = ldvhp47
  port = 3000
}

Owner=testuser # owner of agents for agent property; NOT authenticated!

plugins {
  Authentifier {
    class = MobMan.Security.Utils.AuthClientPasswd
    user = thfu
    password = test
  }
  #Authentifier {
  #  class = MobMan.Security.Utils.AuthClientCert
  #  certfile = certs/user-fjl.p12
  #  passphrase = MobMan
  #}
}

#Socket {
#  class = MobMan.Security.Utils.MobManSocketSSL
#  certfile = certs/user-thfu.p12
#  passphrase = MobMan
#  trustedsigners {
#    certs/ca-LDV.der
```

```
#  }
#}

UserAgents.Demo.TestAgent {
  plugins {
    Wanderer {
      class = MobMan.Utils.Plugins.WandererPlugin
    }
    #Distribute {
    #  class = MobMan.Utils.Plugins.DistributePlugin
    #}
  }
}

UserAgents.Demo.MessageTestAgent {
  Destination {
    host = ldvhp47
    port = 3000
    name = MessageTestAgent
    #sequenceNumber = 8
  }
  plugins {
    Authentifier {
      class = MobMan.Security.Utils.AuthClientPasswd
      user = fjl
      password = test
    }
  }
}

MobMan.Utils.MobManSim {
  Destination {
    host = ldvhp47
    port = 3000
  }
}

MobMan.Pia.PiaClient {
  Destination {
    host = ldvhp47
    port = 3000
  }
  Authentifier {
    class = MobMan.Security.Utils.AuthClientPasswd
    user = thfu
    password = test
  }
  #Authentifier {
  #  class = MobMan.Security.Utils.AuthClientCert
  #  certfile = certs/user-thfu.p12
  #  passphrase = MobMan
  #}
  #Socket {
```

```
  #  class = MobMan.Security.Utils.MobManSocketSSL
  #  certfile = certs/user-thfu.p12
  #  passphrase = MobMan
  #  trustedsigners {
  #    certs/ca-LDV.der
  #  }
  #}
}

MobMan.Pia.PiaServiceAgent {
  plugins {
    Authentifier {
      class = MobMan.Security.Utils.AuthClientPasswd
      user = thfu
      password = test
    }
  }
  # List and Config of User Interfaces
  UserInterfaces {
    # What kind (type) of UI? Currently there are Text and AWT UIs
    Text {
      class = MobMan.Pia.Modules.TextMasterInterface
      # Groups of Modules (they are grouped in a common menu)
      Groups {
        View {
          # List of modules, might as well have subtree with
          # additinal config
          MobMan.Pia.Modules.GetPlatformsTextUI    { rights {} }
          MobMan.Pia.Modules.GetAgentsTextUI       { rights {} }
          MobMan.Pia.Modules.ViewAgentResultTextUI { rights {} }
          MobMan.Pia.Modules.SaveAgentResultTextUI { rights {} }
          MobMan.Pia.Modules.AgentMessageTextUI    { rights {} }
        }
        Admin {
          # o.k., here the names do have a subtree which configures
          # some rights of the PIA modules       { rights {} }
          MobMan.Pia.Modules.ChangeModulesTextUI { rights { PiaConfig } }
          MobMan.Pia.Modules.ChangeUserTextUI    { rights { SecurityConfig } }
        }
        StartStop {
          MobMan.Pia.Modules.StartAgentTextUI     { rights {} }
          MobMan.Pia.Modules.TerminateAgentTextUI { rights { KillAllAgents } }
          MobMan.Pia.Modules.KillAgentTextUI      { rights { Kill } }
          MobMan.Pia.Modules.ShutdownTextUI       { rights { Shutdown } }
        }
      }
    }
    # Another kind of UI, here AWT based. The elements are
    # identical (only class names differ)
    AWT {
      class = MobMan.Pia.Modules.AWTMasterInterface
      Groups {
        View {
```

```
        MobMan.Pia.Modules.GetPlatformsAWTUI    { rights {} }
        MobMan.Pia.Modules.GetAgentsAWTUI       { rights {} }
        MobMan.Pia.Modules.ViewAgentResultAWTUI { rights {} }
        MobMan.Pia.Modules.SaveAgentResultAWTUI { rights {} }
        MobMan.Pia.Modules.AgentMessageAWTUI    { rights {} }
      }
      Admin {
        MobMan.Pia.Modules.ChangeModulesAWTUI { rights { PiaConfig } }
        MobMan.Pia.Modules.ChangeUserAWTUI    { rights { SecurityConfig } }
      }
      StartStop {
        MobMan.Pia.Modules.StartAgentAWTUI     { rights {} }
        MobMan.Pia.Modules.TerminateAgentAWTUI { rights { KillAllAgents } }
        MobMan.Pia.Modules.KillAgentAWTUI      { rights { Kill } }
        MobMan.Pia.Modules.ShutdownAWTUI       { rights { Shutdown } }
      }
    }
  }
}
# List of PIA modules
Modules {
  # name of that module (the name is irrelevant for operation,
  # it is just needed to separate the entries)
  GetPlatforms {
    # Class name of client and server
    Client = MobMan.Pia.Modules.GetPlatformsClient
    Server = MobMan.Pia.Modules.GetPlatformsServer
    # Rights of that module
    rights { PiaConfig }
  }
  GetAgents {
    Client = MobMan.Pia.Modules.GetAgentsClient
    Server = MobMan.Pia.Modules.GetAgentsServer
  }
  ChangeModules {
    Client = MobMan.Pia.Modules.ChangeModulesClient
    Server = MobMan.Pia.Modules.ChangeModulesServer
    rights { PiaConfig }
  }
  ChangeUser {
    Client = MobMan.Pia.Modules.ChangeUserClient
    Server = MobMan.Pia.Modules.ChangeUserServer
    rights { SecurityConfig }
  }
  TerminateAgent {
    Client = MobMan.Pia.Modules.TerminateAgentClient
    Server = MobMan.Pia.Modules.TerminateAgentServer
    rights { KillAllAgents }
  }
  KillAgent {
    Client = MobMan.Pia.Modules.KillAgentClient
    Server = MobMan.Pia.Modules.KillAgentServer
    rights { Kill }
```

```
    }
    Shutdown {
      Client = MobMan.Pia.Modules.ShutdownClient
      Server = MobMan.Pia.Modules.ShutdownServer
      rights { Shutdown }
    }
    StartAgent {
      Client = MobMan.Pia.Modules.StartAgentClient
      Server = MobMan.Pia.Modules.StartAgentServer
    }
    AgentMessage {
      Client = MobMan.Pia.Modules.AgentMessageClient
      Server = MobMan.Pia.Modules.AgentMessageServer
    }
    ViewAgentResult {
      Client = MobMan.Pia.Modules.ViewAgentResultClient
      Server = MobMan.Pia.Modules.ViewAgentResultServer
    }
    SaveAgentResult {
      Client = MobMan.Pia.Modules.SaveAgentResultClient
      Server = MobMan.Pia.Modules.SaveAgentResultServer
    }
  }
}
```

b) mobman-*host*.conf

Das zweite Beispiel `mobman-host.conf` konfiguriert die MobMan-Plattform. Der Dateiname enthält den Hostnamen, sodass mehrere Konfigurationen in einem Verzeichnis liegen können. Damit kann MobMan auch in Netzwerk-Dateisystemen installiert sein und in einem Verzeichnis die Konfiguration aller Plattformen enthalten.

Der aufwändigsten Eintrag ist die Konfiguration des Security Agent, dort speziell die des Security Manager.

```
MobMan.Core.MobManPlatform {
  startAgents {
    MobMan.Security.SecurityAgent
    MobMan.SystemAgents.CommunicationAgent
    MobMan.SystemAgents.MigrationAgent
    MobMan.Pia.PlatformInterfaceAgent
    MobMan.SystemAgents.WebAgent
  }
}

MobMan.SystemAgents.CommunicationAgent {
  port = 3000
  dump {
    console
    error
    police
    debuginfo
  }
  plugins {
```

```
    Proxy {
      class = MobMan.SystemAgents.Utils.ProxyPlugin
    }
  }
}

MobMan.SystemAgents.WebAgent {
  plugins {
    Proxy {
      class = MobMan.SystemAgents.Utils.ProxyPlugin
    }
  }
}

MobMan.SystemAgents.MigrationAgent {
  plugins {
    Authentifier {
      class = MobMan.Security.Utils.AuthClientPasswd
      user = root
      password = all
    }
  }
}

###########
# SECURITY
###########

MobMan.Security.SecurityAgent {
  plugins {
    # plugins inheriting AuthServer are used as authentication server.
    AuthServerPasswd {
      class = MobMan.Security.Utils.AuthServerPasswd
      usersfile = users
    }
    #AuthServerCert {
    #  class = MobMan.Security.Utils.AuthServerCert
    #  usersfile = users
    #  trustedsigners {
    #    certs/ca-LDV.der
    #  }
    #}
  }
  #Socket {
  #  class = MobMan.Security.Utils.MobManSocketSSL
  #  certfile = certs/platform-ldvhp47.p12
  #  passphrase = MobMan
  #  trustedsigners {
  #    certs/ca-LDV.der
  #  }
  #}
  SecurityManager {
    # class of security manager
```

```
class = MobMan.Security.Utils.MobManSecurityManager

# Configure the verbosity of the logging facility
# (list of any of the following):
# Deny:         Log every access that is denied.
# DenyDetail:   Log some details concerning the denied accesses.
# Warning:      Log warning messages too.
# Permit:       Log every access that is permitted.
# PermitDetail: Log some details concerning the permitted accesses.
# Log { Deny DenyDetails Warning Permit PermitDetails }
Log {
  Deny
  DenyDetails
}

# Specify the destination for log messages. Possible values are
# stdout (output to console) or stderr (error stream to console).
# Other options might become available (file, sockets, agents)
LogDest {
  stdout
}

# Each file gets a set of access rights. Only files that are listet
# in this section can be accessed by agents running under control
# of this SecurityManager
FileAccess {
  /usr/lib/java/bin/../lib/awt.properties                = r
  /usr/lib/java/bin/../lib/font.properties.de_DE_8859_1  = r
  /usr/lib/java/bin/../lib/font.properties.de_DE         = r
  /usr/lib/java/bin/../lib/font.properties.de_8859_1     = r
  /usr/lib/java/bin/../lib/font.properties.de            = r
  /usr/lib/java/bin/../lib/font.properties               = r
  certs/ca-LDV.der                                       = r
  certs/platform-ldvhp47.p12                             = r
}

# Only connections from hosts within the set 'PermitAcceptFrom'
# and outside the set 'DenyAcceptFrom' are permitted. Connections to
# foreign hosts are only permitted if the foreign host is part of the
# set 'PermitConnectTo' and not part of 'DenyConnectTo'.
# Each definition can consist of a single IP-Address or a range of
# IP-Addresses.
PermitAcceptFrom {
  129.187.105.0-129.187.105.255
  127.0.0.1
  129.187.240.80-129.187.240.89
}
PermitConnectTo {
  129.187.105.0-129.187.105.255
  129.187.240.80-129.187.240.89
  127.0.0.1
}
```

```
# Classes that are permitted to create ServerSockets (several possible).
CreateServerSocketClass {
  MobMan.Core.MobManSocket
  iaik.security.ssl.SSLServerSocket
  #MobMan.SystemAgents.Utils.MobManSocketSSL
}

# Range of ports that can be used for ServerSockets (several possible).
ServerSocketPort {
  3000-3010
  4000
}

# Only bytecode that is part of the specified class is permitted
# to terminate the JVM (several possible).
# Class that calls java.lang.System -> java.lang.Runtime
ExitClass {
  MobMan.Core.MobManPlatform
}

# Only bytecode that is part of the specified class is permitted
# to load native code libraries (several possible).
# Class that calls java.lang.System -> java.lang.Runtime
LinkNativeClass {
  java.net.PlainSocketImpl
  java.net.PlainDatagramSocketImpl
  sun.awt.motif.MToolkit
  java.math.BigInteger
}

# Only bytecode that is part of the specified class is permitted
# to open a top level window (several possible).
# This restriction applies only to dynamically loaded classes.
# Stationary classes may open top level windows without any restriction.
OpenTopLevelWindow {
  MobMan.SystemAgents.GUIAgent
}

# Threadgroup that can be manipulated by dynamically loaded classes
# (several possible)
# Other Threadgroups can only be altered by stationary classes.
AlterableThreadGroup {
  AgentThreads
}

# Only bytecode that is part of the specified class is permitted
# to access the members and fields of another class by using the
# Reflection-API (several possible).
MemberAccessClass {
  MobMan.Core.Utilities
}

# Only bytecode that is part of the specified class is permitted
```

```
    # to create a classloader (several possible).
    CreateClassLoader {
      MobMan.SystemAgents.MigrationAgent
      MobMan.Pia.PlatformInterfaceAgent
    }

    # Set this flag to true in order not to restrict the access to the local
    # filesystem from stationary classes. If enabled, the section
    # FileAccess can be reduced to list files that should be accessible
    # by dynamically loaded classes (e.g. agents), else it has to list
    # much more files.
    UnrestrictedFileAccessSC = true
  }
}
```

c) users

Die `users`-Datei enthält die Namen der Benutzer, die sich auf der Plattform authentifizieren können. Zu jedem User sind dessen Rechte angegeben. Je nach Verfahren ist auch noch das kodierte Passwort nötig, beim Zertifikatsverfahren muss sich nur der Name mit den Angaben im Benutzerzertifikat decken. Dieses Zertifikat muss von einer bekannten und vertrauenswürdigen CA ausgestellt sein.

```
# Known users for AuthServerUserPasswd
UserPasswd {
  root {
    # passwd needs quotes in order to protect the equal signs
    # (equal signs are here due to base64 encoding rules)
    passwd = "oYGmA3acH5itkn5zZ8eqUQ=="    # all
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
      PiaConfig
      KillAllAgents
    }
  }
  thfu {
    passwd = "CY9rzUYh03PK3k6DJie09g=="    # test
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
      PiaConfig
      PiaUser
    }
  }
  fjl {
    passwd = "CY9rzUYh03PK3k6DJie09g=="    # achim
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
```

```
      PiaConfig
      PiaUser
    }
  }
  killer {
    passwd = "U0c1iE00EHF2Lt568BxT6A=="    # kill
    rights {
      KillAllAgents
    }
  }
}

# Known users for AuthServerCert
Certificate {
  root {
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
      PiaConfig
      KillAllAgents
    }
  }
  thfu {
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
      PiaConfig
      PiaUser
    }
  }
  fjl {
    rights {
      StartAgent
      StartSystemAgent
      SecurityConfig
      PiaConfig
      PiaUser
    }
  }
  killer {
    rights {
      KillAllAgents
    }
  }
}
```

B Das Launch-Programm

Um den Start der Plattform, der Agenten und der Hilfstools zu vereinfachen, enthält MobMan ein Startskript mit Namen `launch`. Dieses Skript setzt die nötigen Umgebungsvariablen (vor allem `CLASSPATH`) und vereinfacht die Übergabe von Parametern. Statt etwa den vollständigen Klassennamen mit seinem Package anzugeben, genügt der Name des Agenten. Den Rest sucht das Skript im Dateisystem.

Zusätzliche Parameter übergibt Launch direkt an die aufgerufene Klasse. Nachfolgend sind die drei wichtigsten Startmöglichkeiten aufgeführt. Weitere Details sind der Dokumentation zu entnehmen.

```
launch mobman
```

Startet die Plattform. Die Konfiguration steht in der Datei `mobman-`*`hostname`*`.conf`; alternativ kann der Name der Konfigurationsdatei als weiterer Parameter angegeben werden.

```
launch agentenname [-platform host:port] [-owner name]
```

Das Launch-Programm benutzt die Konfiguration in `launch.conf` im aktuellen Verzeichnis. Es sucht die Agentenklasse im Agentenverzeichnis und, soweit vorhanden, in `Classes` unter dem aktuellen Verzeichnis. Der Name kann daher abgekürzt angegeben sein.

```
launch minica
```

Die Mini-CA führt interaktiv durch die Aufgaben einer CA. Sie ist als Testwerkzeug zu verstehen. Die erzeugten Zertifikate entsprechen nicht notwendigerweise den Sicherheitsanforderungen, da sie keine qualitativ hochwertigen Zufallszahlen erzeugt. Es generiert direkt die benötigten Schlüsselpaare und Zertifikate, kann aber keine Fremdschlüssel signieren. Für höhere Anforderungen kann jede X.509v3-CA-Software dienen.

C TLS: Transport Layer Security

TLS [RFC 2246] basiert auf dem von Netscape entwickelten Secure Socket Layer Protocol SSL Version 3. Es fügt eine weitere Protokollschicht auf der Basis eines verlässlichen Transportprotokolls (etwa TCP) ein und bietet den darüber liegenden Schichten den identischen Dienst, erweitert um:

- Kryptografische Authentifikation (einseitig oder zweiseitig).
- Sicherstellen der Datenintegrität.
- Sicherstellen der Vertraulichkeit von Daten (Verschlüsselung).

TLS besteht aus mehreren Teilprotokollen:

- Das TLS Record Protocol stellt die Basis dar. Es bietet Verschlüsselung, Integritätssicherung, Komprimierung und Fragmentierung. Es kann beliebig große Datenblöcke transportieren.

- Das TLS Handshake Protocol besteht selbst aus drei Teilprotokollen:
  - Handshake Protocol: Authentifikation und sicheres Aushandeln eines gemeinsamen Geheimnisses (shared secret, vor allem der Sitzungsschlüssel).
  - Alert Protocol: Fehlermeldungen innerhalb der TLS-Schicht.
  - Change Cipher Spec Protocol: Ändern der Sicherheitsparameter während einer Verbindung.
- Application Data Protocol: Transport der Nutzerdaten.

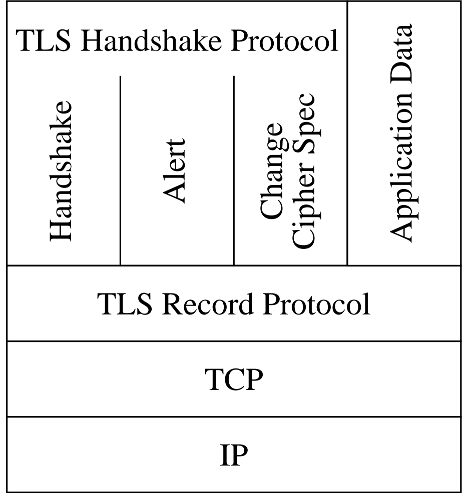

Abbildung C.1 *Aufbau des TLS-Protokolls*

Das TLS Handshake Protocol ist für Zustandsänderungen zuständig. Es handelt die Sicherheitsparameter aus (Algorithmen sowie Zufallswerte zum Generieren der Sitzungsschlüssel), überträgt die X.509-Zertifikate und sorgt für die Authentifizierung.

Beide Endsysteme führen für beide Kommunikationsrichtungen je zwei Zustände: Current und Pending. Mit dem Handshake Protocol wird der nächste Zustand (Pending Connection State) ausgehandelt. Eine Change-Cipher-Spec-Nachricht erklärt Pending zum neuen Current State.

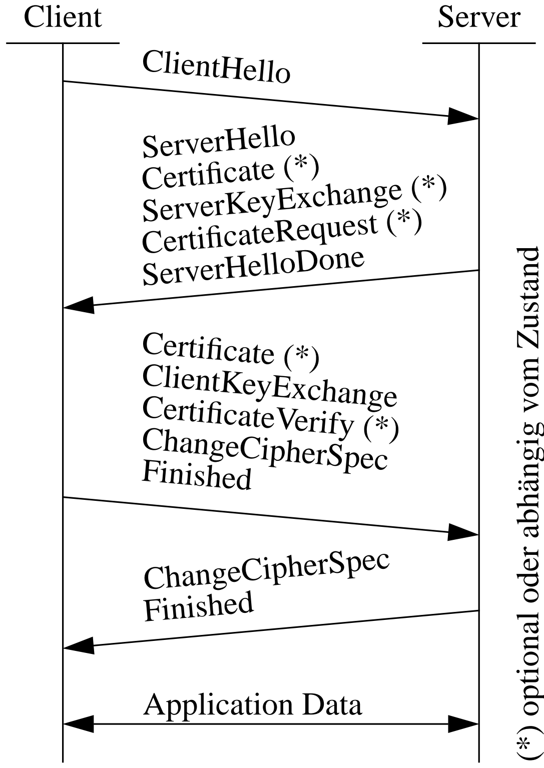

Abbildung C.2 *Verbindungsaufbau bei TLS*

Die Applikationsdaten werden in der Record-Protocol-Schicht fragmentiert und abhängig vom Current State komprimiert, verschlüsselt und integritätsgesichert übertragen.

D Schlüssel und Zertifikate in Browser importieren

Mit den Benutzerzertifikaten, die etwa die Mini-CA ausgestellt hat, kann sich ein Anwender beim Web Agent authentifizieren. Der Browser muss dazu das Benutzerzertifikat und das Zertifikat der CA kennen. Beide können in den Browser importiert werden, als Beispiel dient hier der Netscape Navigator in einer 4.x-Version.

Als erster Schritt muss die Root Level CA als „trusted root" anerkannt werden, da die Mini-CA eigene Zertifikate ausstellt und entsprechend nicht in der vorkonfigurierten CA-Liste im Browser enthalten ist.

- Die Mini-CA überträgt ihr Zertifikat (DER-codiert) über eine HTTP-Verbindung an den Browser. Durch den MIME-Typ im HTTP-Header erfährt der Browser, dass es sich hier um ein CA-Zertifikat handelt. Als Server kann die Mini-CA selbst dienen.
- Der Benutzer spricht über einige Dialogfenster diesem Zertifikat sein Vertrauen aus. Als Verwendungszweck ist „Certifying Network Sites" für Plattformzertifikate und „Certifying E-Mail Users" für Benutzerzertifikate anzugeben. Die Mini-CA stellt zwar keine Zertifikate für E-Mail aus, diese Option gilt aber auch für HTTPS-Client-Zertifikate (bei Netscape sind auch andere Funktionen eigenartig benannt).

Bei der Mini-CA sind keine Certificate Requests nötig. Die CA stellt direkt das fertige Zertifikat aus, inklusive dem geheimen Schlüssel. Der Browser importiert den Schlüssel zusammen mit dem Zertifikat aus einer PKCS#12-Datei:

- Öffnen des Fensters `Security Info`.
- Dort `Certificates -> Yours` wählen.
- Dann `Import a Certificate...`
- PKCS#12-Datei wählen und importieren.
- Nun noch die Passphrase eingeben. Mit dieser hat die Mini-CA den geheimen Schlüssel in der PKCS#12-Datei geschützt.

Bei der nächsten HTTPS-Verbindung kann der Browser dieses Zertifikat nutzen, wenn der Server eine Benutzerauthentifizierung wünscht. Durch ein Dialogfenster wählt der Benutzer aus den importierten Zertifikaten das für diese Verbindung gewünschte aus oder bricht die Authentifizierung ab.

E Weitere Agentensysteme

Abschnitt 2.3 *Bestehende Agentensysteme*, Seite 22 gab bereits einen Überblick über wichtige bestehnede Agentensysteme. Die dort nur kurz vorgestellten Plattformen werden nachfolgendend näher beschrieben.

E.1 Kommerzielle Agentensysteme

a) Aglets

Das Aglets-System von IBM implementiert schwache Migration, um mit einer Standard-JVM (Java Virtual Machine) arbeiten zu können. Für starke Migration wären Änderungen an der JVM erforderlich. Aglets enthält einen Persistenzmechanismus, bei dem der Agenten-Code und -Zustand gesichert und der Agent für bestimmte Zeit deaktiviert wird. Agenten-Proxies sorgen dafür, dass Agenten unabhängig von ihrem Aufenthaltsort ansprechbar sind. Das System unterstützt verschiedene Kommunikationsmechanismen. Das Laden der Klassendefinition kann auf verschiedenen Wegen stattfinden, etwa von einem zentralen Codeserver oder über das Agent Transfer Protocol ATP während der Migration. Dabei sind die Dateien im Java-eigenen Archivformat Jar zusammengefasst und verpackt.

Das Sicherheitsmodell unterscheidet zwischen Prinzipalen (identifizierbare Entitäten) und Identitäten (Name und eventuell weitere Attribute). Aglets besteht aus Agenten (Aglets), Kontexten (Plattformen) und Domänen (Gruppen von Plattformen). Daraus ergeben sich folgende Prinzipale:

- Aglet (Agent)
- AgletManufacturer (Hersteller/Programmierer)
- AgletOwner (Absender/Eigentümer)
- Context (Plattform)
- ContextManufacturer (Hersteller/Programmierer)
- ContextMaster (Eigentümer/Administrator der Plattform)
- Domain (Gruppe von Plattformen)
- DomainAuthority (Eigentümer/Administrator der Domäne)

Aus diesen Prinzipalen leiten sich die jeweiligen Rechte (Privileges) ab. Die dazu nötige Sicherheits-Policy wird in einer eigenen Konfigurationssprache definiert. Sowohl die Plattform als auch der Agent selbst verfügen über eine eigene Policy. Ein eigener Security Manager sorgt für die Einhaltung der Rechte. [Karjoth 97] [Lange 98b] [Oshima 98]

b) Concordia

Das von Mitsubishi entwickelte Agenten-Framework Concordia basiert ebenfalls auf Java. Bei Concordia kümmert sich der Security Manager um die Authentifizierung und Verschlüsselung von Agenten. Agenten werden anhand ihres Eigentümers authentifiziert. Der Persistence Manager sorgt für das Einfangen des Zustands eines Agenten; der Zustand wird für die Migration und für das Checkmarking benötigt. Nach einem Systemausfall kann der Agent mit dem Zustand neu gestartet werden, den er beim letzten Checkpoint hatte. [Mitsubishi 98]

Concordia unterscheidet zwei Arten von Agenten-Kommunikation: Neben der Verteilung von Events können Agenten auch eng miteinander kooperieren [Wong 97]. Die Kommunikation wird mit SSL gesichert. Agenten werden auf jeder Plattform authentifiziert, dazu dient eine einfache Passwort-Datenbank. Agenten sind auch dann vor fremdem Zugriff geschützt, wenn sie sich im Persistenzspeicher befinden: Concordia verschlüsselt die Dateien [Wong 99]. Die Plattform bietet auch ein transaktionsbasiertes, verlässliches Message Queuing für asynchrone Kommunikation [Walsh 98].

c) Grasshopper

Das vom deutschen Unternehmen IKV++ entwickelte Java-basierte Mobile-Agenten-System Grasshopper soll für Mobile-Commerce-Anwendungen dienen – der Hersteller arbeitet im Telekommunikationsumfeld, vor allem an UMTS-Anwendungen (Universal Mobile Telecommunications System). Grasshopper ist für nichtkommerzielle Anwendungen kostenlos. Die Plattform wurde kompatibel zu MASIF entwickelt, für die MASIF-Unterstützung ist allerdings eine zusätzliche Softwarekomponente zu installieren. Interessant ist, dass für Grasshopper ebenfalls eine FIPA-Komponente existiert.

Die Plattform wird bei Grasshopper als Agency bezeichnet. Die Core Agency bietet eine Reihe von Diensten an:

- Kommunikation über CORBA-IIOP (Internet Inter-ORB Protocol), Java-RMI (Remote Method Invocation) oder gewöhnliche Sockets. RMI und Sockets können mit SSL gesichert werden.
- Registrierung aller Agenten und Orte innerhalb der Agency. Orte sind Gruppierungen von Agenten, sie unterteilen eine Agency weiter.
- Management der Plattform
- Migration der Agenten
- Security
- Persistenz der Agenten

Bei den Sicherheitsdiensten unterscheidet Grasshopper zwischen externer und interner Sicherheit. Für externe Sicherheit, also den Schutz der Agenten und Daten während der Übertragung, kommt SSL zum Einsatz. Die interne Sicherheit schützt die Agenten voreinander und bietet Authentifizierung und Autorisierung zusammen mit Zugriffspolitiken. Die internen Sicherheitsmechanismen stützen sich vor allem auf die Dienste des JDK 1.2 und einer Codesource-basierten Identifikation von Agenten. [IKV 01a]

d) Jumping Beans

Die Java-basierten Jumping Beans transferieren angeblich neben Code und Daten auch den Zustand; unklar ist, ob dies den Thread-Zustand einschließt: [Gray 00] vermutet, dass letztlich doch die bei Java-Agentensystemen übliche schwache Migration implementiert ist.

Jede Jumping-Beans-Domäne besitzt einen zentralen Server, der Plattformen und Agenten in der Domäne authentifiziert und verwaltet. Agenten können derzeit [Gray 00] allerdings nicht zwischen verschiedenen Domänen migrieren.

e) Odyssey

Odyssey ist General Magics Telescript-Nachfolger. Es ist eine neue Implementierung, vollständig in Java 1.1 gehalten. Das Design orientiert sich stark an Telescript. Die Plattform implementiert schwache Migration.

Die zentralen Objekte in Odyssey sind Agenten und Orte, Orte benötigen als Ausführungsumgebung lediglich eine RMI-Registry. Das Sicherheitsmodell basiert auf dem Java-Sicherheitsmodell und einem Security Manager. Odyssey erreicht in Version 1.0 allerdings nicht den Funktionsumfang von Telescript. [Bursell 97]

f) Telescript

Telescript ist der Klassiker unter den Systemen mobiler Agenten. Dieses kommerzielle Agentensystem von General Magic bietet starke Migration. Es verwendet dazu eine eigene Programmiersprache, die an Java und C++ angelehnt ist. Sicherheit gilt als wichtiger Designfaktor: Die Rechte der Agenten sind anhand von Permits festgelegt (vergleichbar zu Capabilities), Permits regeln auch die Resource Limits [Gray 96a]. Telescript-Plattformen authentifizieren den Besitzer des Agenten. Die Kommunikation zwischen den Agenten kann über `meet` auf der gleichen Plattform oder mit `connect` über Plattformgrenzen hinweg stattfinden. [Gray 00] [White 96]

Das Sicherheitsmodell von Telescript besteht aus vier Ebenen [Tardo 96]:

- *Object Runtime Safety*
  Ähnliche Schutzmechanismen wie in Java: Garbage Collection, keine Zeiger, Typ-Überprüfung zur Laufzeit.
- *Process Safety and Security*
  Jeder Prozess gehört einer authentifizierten Identität. Jedes Objekt gehört eindeutig zu einem solchen Prozess. Permits beschränken den Zugriff auf Ressourcen. Telescript-Orte können entscheiden, ob sie einen Agenten entgegennehmen; wenn sie das verneinen, bleibt der Agent am alten Ort. Spezielle Mix-in-Klassen können einer Klasse besondere Eigenschaften verleihen, etwa dass ihre Objekte nicht migrieren dürfen.
- *System Safety*
  Gewährt nur kontrollierten Zugriff auf das Hostsystem.
- *Network Security*
  Regionen enthalten ihre eigenen Policies. Die Kommunikation kann über kryptografisch gesicherte Kanäle stattfinden.

E.2 Forschungsprojekte

a) AgentTcl

Am Dartmouth College (Hanover, USA) wurde eine der ersten Agentenplattformen entwickelt, die auch für die Forschung frei zugänglich und nicht wie der Vorreiter Telescript von kommerziellen Interessen geprägt war. Die ursprüngliche Version AgentTcl (später D'Agents) benutzt Tcl als Programmiersprache für Agenten. Der Tcl-Interpreter wurde dazu um die Fähigkeit erweitert, den Zustand eines Agenten zu speichern und wieder herzustellen. Ein einziges Kommando `agent_jump` erledigt aus Programmierersicht die Migration. Agenten können sich untereinander mit `agent_meet` verständigen, dieses Kommando stellt eine transparente Verbindung her. Durch die Änderungen am Interpreter ist AgentTcl in der Lage, seinen Agenten starke Migration zu ermöglichen.

Die Agenten laufen bei diesem System in eigenen Prozessen. Das vereinfacht nicht nur die Verwaltung gleichzeitig laufender Agenten, die Architektur kann dadurch auch verschiedene Interpreter-Sprachen für Agenten unterstützen. [Gray 96b]

AgentTcl nutzt SafeTcl, um die Plattform vor dem Agenten zu schützen. Ähnlich einer Sandbox stellt SafeTcl dem Interpreter, der den Agenten ausführt, nur eine eingeschränkte Menge an Kommandos zur Verfügung. Durch Aliasing kann ein Master-Interpreter potenziell gefährliche, aber notwendige Kommandos auch dem SafeTcl-Interpreter zur Verfügung stellen. Der Master bestimmt, ob das einzelne Kommando dann tatsächlich ausgeführt wird. Dieses Konzept ähnelt zwar dem Security Manager von Java, das SafeTcl-Konzept kann aber auf beliebige Kommandos angewendet werden, ohne dass diese von der Existenz von SafeTcl wissen.

Die Authentifizierung des Agenten-Eigentümers und der beteiligten Plattformen findet mit RSA-Signaturen statt, die mit PGP erstellt werden [Gray 96a]. Resource-Manager-Agenten überprüfen die Autorisierung. Hierbei handelt es sich um stationäre Agenten, die eine Access Control List (ACL) führen [Gray 97b].

b) Ajanta

Ajanta ist Java-basiert und implementiert schwache Migration. Agenten haben einen Eigentümer und meist auch einen so genannten Guardian, ein stationäres Objekt, das Ausnahmesituationen behandelt. Tritt eine solche ein, überträgt Ajanta den Agenten zu seinem Guardian. Dort kann der Agent die `report`-Methode des Guardians aufrufen. Ajanta enthält auch einen eigenen Namensdienst, der einen sicheren Namensraum aufspannt, um Agenten und Plattformen eindeutig identifizieren zu können.

Agenten laufen in ihrer eigenen Thread Group und werden über ihren eigenen Classloader geladen. Dadurch stellt Ajanta jedem Agenten seine eigene Protection Domain zur Verfügung, die Agenten voneinander isoliert. Der Zugang zu Systemressourcen ist durch Proxy-Objekte abgesichert. Durch diesen Ansatz kann das Zugriffsrecht einem Agenten auch nachträglich wieder entzogen werden.

Ajanta benutzt ein eigenes Protokoll, um auch die Agentenkommunikation mit RMI (Java Remote Method Invocation) zu authentifizieren. Die Migration kann authentifiziert und verschlüsselt stattfinden (siehe aber auch [Roth 01b]). Jeder Agent hat ein Credentials-Objekt, das seine Rechte beschreibt. Eine Signatur über dieses Objekt authentifiziert den Agenten (-Programmierer oder -Eigentümer).

Ajanta enthält auch einige Vorkehrungen, die den Agenten vor der Plattform schützen sollen [Karnik 98], [Karnik 99a]:

- Nur lesbarer Zustand (Read-Only): Eine Signatur stellt sicher, dass Änderungen erkannt werden.
- Nur anfügbare Logs (Append-Only): Änderungen an älteren Einträgen werden erkannt.
- Nur für bestimmte Plattformen lesbare Daten: Verschlüsselt mit dem Public Key der jeweiligen Plattform.

c) Ara

Ara (Agents for Remote Action) bietet starke Migration. Es unterstützt einige verschiedene Sprachen: Tcl, Java und sogar C/C++. Letztere wird aber ähnlich wie Java in eine Bytecode-Zwischensprache übersetzt und nicht direkt in Maschinensprache. Ungewöhnlich ist, dass Ara trotzt der verschiedenen unterstützten Sprachen mit Multi-Threading arbeitet und nicht wie D'Agents mit einzelnen Prozessen. Die Implementierung der Plattform selbst ist durch diese Architektur recht schwierig, dafür ist die Performance wesentlich besser als bei Multi-Prozess-Ansätzen [Gray 00]. Rechte werden durch Allowances festgelegt (vergleichbar zu Capabilities) [Peine 97b].

Ara unterscheidet bei der Authentifizierung zwischen dem Hersteller (Programmierer), der den Code signiert, und dem Besitzer, der die Parameter und die Allowances signiert. Ein Ausweis (Passport) enthält die Allowances eines Agenten, seine global eindeutige ID sowie die Zertifikate und Signaturen des Herstellers oder Programmierers und die des Besitzers oder Absenders. Die Migration findet per SSL gesichert statt [Peine 98].

Ara-Agenten können auch aus nativem Code bestehen, wenn die Sicherheitsanforderungen dies zulassen, etwa weil der Agent stationär auf einer Plattform verweilt und diese um zusätzliche Funktionen ergänzt [Peine 97a].

d) D'Agents

Mit der Namensänderung in D'Agents erfüllt AgentTcl das Versprechen, mehr als eine Programmiersprache für Agenten bereitzustellen. Neben Tcl sind nun auch Scheme und Java als Sprache möglich. Mithilfe von PGP signiert und verschlüsselt D'Agents seine Agenten. Der Agent kann selbst entscheiden, ob er diesen Schutz wünscht und wenn ja, ob er nur signiert oder signiert und verschlüsselt übertragen werden soll. Auch die Kommunikation zwischen Agenten kann mit diesen Mitteln geschützt werden. Da D'Agents für jede kryptografische Operation ein externes PGP-Kommando aufrufen muss (als eigenen Prozess), ist dieses Verfahren allerdings relativ langsam.

Agenten tragen nie selbst ihre geheimen Schlüssel mit sich. Der Agent (beziehungsweise sein Absender) kann sich daher nur gegenüber der ersten Plattform mit einer PGP-Signatur authentifizieren. Alle folgenden Plattformen müssen ihren Vorgängern vertrauen. Der Agent wird bei jeder Migration mit dem Schlüssel der Absender-Plattform signiert. Verlässt der Agent eine Gruppe von Plattformen, die sich gegenseitig vertrauen, wird er zu einem anonymen Agenten, der sich nicht mehr authentifizieren kann.

Resource Manager sind für die Umsetzung der Security Policy zuständig. Je nach Ressource und Autorisierung des jeweiligen Agenten sorgen sie für die Einhaltung der Policy. Die Einhaltung einzelner Regeln stellen so genannte Enforcement-Module sicher, die für jede der unterstützten Programmiersprachen getrennt implementiert sind. [Gray 98a]

Um auch für Java starke Migration zu ermöglichen, verwendet D'Agents eine modifizierte JVM [Gray 00].

e) ffMAIN

Die in Frankfurt am Main entwickelte Agentenplattform ffMAIN benutzt HTTP als Migrationsprotokoll. Sie verwendet einen eigenen MIME-Typ (Multipurpose Internet Mail Extensions) `application/agent`, der in Untertypen untergliedert ist (`application/agent-attributes`, `.../agent-code`, `.../agent-state`) [Lingnau 95]. Agenten können in Tcl oder Perl geschrieben sein. Die Plattform unterstützt die Kommunikation mit Anwendern über HTTP und HTML, also mithilfe eines Web-Browsers. Sie benötigt daher keine spezialisierte Management-Software. [Lingnau 98].

f) Gypsy

Gypsy ist wie viele andere Systeme auch Java-basiert. Es benutzt aber Java Beans für die Agenten und überträgt sie als Jar-Archiv. Das System enthält eine zentrale Registry, die mithilfe von RMI umgesetzt ist. Es bietet mehrere Verbreitungstechniken (RMI, E-Mail+PGP und SSL-gesicherte Kommunikation) in Form eigener Communicator-Agenten. Die Communicator-Agenten kapseln die Kommunikation in eigenen Komponenten. Die Plattform verwendet einen eigenen Security Manager sowie Java 2 Security Policies. [Gypsy 00]

Gypsy enthält das Konzept von kooperierenden Supervisor- und Worker-Agenten. Der Supervisor-Agent migriert und startet bei Bedarf seine Worker-Agenten. Gypsy unterstützt mehrere Sprachen: Die Interpreter für die zusätzlichen Sprachen sind selbst in Java programmiert und werden daher innerhalb der JVM ausgeführt. [Jazayeri 00]

Gypsy legt besonderen Wert darauf, dass für die meisten Aufgaben des Systems Agenten eingesetzt werden, also das Agenten-Paradigma möglichst universell genutzt wird. [Lugmayr 99]

g) MAMAS

MAMAS [Bellavista 99] wurde am Dipartimento di Elettronica, Informatica e Sistemistica an der Università di Bologna in Italien entwickelt. Der Schwerpunkt liegt hier in der Eignung für das Systemmanagement, wobei Interoperabilität und Sicherheit im Vordergrund stehen.

Für die Administration werden zwei Abstraktionen eingeführt: Die Network Locality abstrahiert lokale Netze (physikalische Ebene), während die Administration Locality die Domäne bezeichnet, für die ein Administrator zuständig ist (logische Ebene). Parallel zu diesen beiden Abstraktionen arbeitet MAMAS auch mit den gewohnten Orten, die aber zu Domänen zusammengefasst sind. Die Domänen sind nicht hierarchisch gegliedert. Die administrativen Domänen kommen vor allem bei den Berechtigungen und Zugangskontrollen zum Einsatz. Dadurch wird ein Vertrauensmodell für Managementaufgaben gebildet.

Sicherheit ist bei MAMAS in unterschiedlichen Schichten implementiert. Die Instanz, deren Berechtigungen ein Agent annimmt, wird durch eine digitale Signatur des Agenten-Codes (zur Authentifizierung) festgestellt. Agenten können verschlüsselt und vor Manipulation geschützt übertragen werden. Auf den Plattformen findet eine Autorisierung statt, die auf der Basis der Rechte des jeweiligen Administrators die Zugriffsrechte zuteilt.

Die Interoperabilität soll durch den MASIF-Standard sichergestellt werden. MASIF basiert auf CORBA; dadurch ergibt sich die Möglichkeit, CORBA auch zur Integration klassischer Management-Anwendungen zu nutzen. MAMAS ist in Java 1.2 implementiert. Die Integration mit CORBA und MASIF wird durch eine CORBA-Bridge und eine MASIF-Bridge erleichtert, die als zusätzliches Modul auf der Plattform installiert sind. Da die MASIF-Bridge allerdings recht komplex ist, empfehlen die Autoren von MAMAS, sie nur an zentralen Stellen einzusetzen und ansonsten den einfacheren CORBA-Client-Server-Ansatz zu nutzen.

h) MAP

Die Java-basierte Agentenplattform MAP implementiert schwache Migration und wurde konform zum MASIF-Standard entwickelt. Die Klassen werden von einer Liste möglicher Code-Server geladen. MAP wurde unter anderem für den Einsatz im Netzmanagement entwickelt. [Pietro 99] [Puliafito 97]

i) MASA

Auch MASA ist Java-basiert und kompatibel zu MASIF. Es nutzt CORBA als Kommunikationsprotokoll. MASA erweitert den MASIF-konformen MAFfinder durch einen Verzeichnis- und Lokalisierungsdienst, der das Auffinden von Agenten erleichtert. MASA enthält ein Gateway, mit dessen Hilfe es auch Agenten des Voyager-Systems ausführen kann. [Gruschke 99]

Die MASA-Plattform nutzt SSL zur Sicherung der Kommunikation, X.509-Zertifikate zur Authentifizierung und Permissions zur Autorisierung. [Rölle 99]

j) MOA

Mobile Objects and Agents (MOA) benutzt Java (JDK 1.1). In der ersten Version wurden viele Sicherheitsfeatures ausgelassen, keine Authentifizierung oder Autorisierung, nicht einmal ein Security Manager ist vorhanden. Der Zugang zu Ressourcen wird jedoch sehr detailliert geregelt [Milojicic 98b]. MOA enthält unter anderem einen eigenen Namensdienst und einen Locator-Service, der den aktuellen Aufenthaltsort von Agenten ermittelt. Es bietet auch ein eigenes Komponentenmodell für Agenten. Eine Besonderheit ist, dass MOA die von den Agenten geöffneten Kommunikationskanäle über die Migration hinweg offen hält. [Milojicic 98c]

k) Mole

Mole wurde am Institut für parallele und verteilte Höchstleistungsrechner der Fakultät Informatik an der Universität Stuttgart entwickelt. [Baumann 98a] und [Straßer 96] fassen die wesentlichen Konzepte zusammen: Mole nutzt Java für die Plattform und als Sprache für die Implementierung der Agenten. Das System basiert auf dem Konzept von Agenten und Orten: Agenten sind wie üblich aktive Entitäten, die autonom zwischen den Orten wandern können. Sie können Multi-Threaded sein, Code und Zustand bleiben bei der Migration erhalten (schwache Migration). Ein Agentensystem besteht aus mehreren Orten, die eine sichere Ausführungsumgebung darstellen sowie weitere Dienste bereitstellen.

Jeder Agent ist durch eine globale, eindeutige ID gekennzeichnet. Diese ID wird ihm bei der Erzeugung von der erzeugenden Plattform zugewiesen. Die ID bleibt während der Lebenszeit des Agenten unverändert. Sie besteht aus folgenden Teilen:

- Dynamischer Zähler, der für jeden neuen Agenten inkrementiert wird.
- Crash-Counter, der bei jedem Neustart der Plattform (und bei jedem Überlauf des dynamischen Zählers) inkrementiert wird.
- IPv6-Adresse des Hosts.
- Portnummer der Plattform.

Ein Ort liegt immer auf einem einzelnen Rechner, ein Rechner kann aber mehrere Orte beherbergen. Orte werden je nach Netzanbindung ihres Rechners in zwei Klassen unterteilt:

- *Connected*
  Der Rechner ist ständig mit dem Netzwerk verbunden.
- *Associated*
  Der Rechner ist nur zeitweise am Netz (etwa bei PDAs oder Notebooks).

Mole nutzt schwache Migration, um ohne Modifikationen der JVM auszukommen. Die Kommunikation findet in Sessions statt, innerhalb derer über RMI kommuniziert wird. Alternativ können die Agenten in der Session asynchrone Nachrichten über Messaging-Objekte übertragen. Eine Session lässt sich innerhalb der Plattform oder über Plattformgrenzen hinweg aufbauen. Die Kommunikation kann dabei nicht nur direkt von Agent zu Agent, sondern auch anonym zwischen Gruppen von Agenten stattfinden. Diese Art der Kommunikation ist über Events realisiert.

Bei der Sicherheit verlässt sich Mole auf das Sandbox-Modell. Normale User-Agenten erhalten keinerlei Zugang zum Rechner. Nur spezielle Service-Agenten dürfen auf den Rechner zugreifen; sie sind aber nicht mobil und stellen daher kein grundsätzliches Sicherheitsproblem dar. Orte können auch Zugangsbeschränkungen (im Sinne einer Access Control List, ACL) aufstellen und nur bestimmte Agenten zulassen.

Mole benutzt einen eigenen Scheduler für Java-Threads und kann damit auch ein Accounting von CPU-Zeit, Netzwerk-Kommunikation (lokal/remote), Anzahl erzeugter Agenten und Gesamtzeit im Ort vornehmen.

l) SAE, WASP

SAE steht für Server Agent Environment for Web Agents, WASP für Web Agent-based Service Providing [Fünfrocken 99a]. Es handelt sich hierbei um eine Agentenerweiterung für Webserver, die damit mobile Agenten (hier Web Agents genannt) beherbergen können. Die Agenten erhalten Zugang zu den lokalen Daten des Servers.

SAE basiert auf Java. Die Rechte der Agenten sind durch Capabilities festgelegt, die User, denen die Rechte zugeordnet sind, werden über die HTTP-Mechanismen authentifiziert. Die Übertragung der Agenten findet über HTTP statt [Fünfrocken 98a]. SAE unterscheidet zwischen dem Agent Owner (hat den Agenten installiert) und dem Agent User (hat ihn gestartet). Die tatsächlichen Rechte ergeben sich aus den Rechten von Owner und User sowie den Default-Vorgaben des Server-Administrators.

m) TACOMA

TACOMA unterstützt schwache Migration, die Spezialität dieser Plattform sind die vielen unterstützten Programmiersprachen (C, C++, ML, Perl, Python, ...). Jeder Agent läuft in diesem System in einem eigenen Unix-Prozess. Ungewöhnlich an TACOMA ist, dass die Agenten selbst dafür verantwortlich sind, ihren Code und ihren Zustand in einem Folder zu speichern. Diesen Folder überträgt das Framework dann bei der Migration zum Zielhost. Durch diese Architektur wird das Programmieren von Agenten zwar erschwert, andererseits zwingt dieses Verfahren dem Agentenentwickler kein spezielles Programmiermodell auf [Johansen 98a] und es vereinfacht die Unterstützung weiterer Programmiersprachen. [Johansen 95] [Gray 00]

Da in TACOMA alle Agenten als eigener Prozess laufen, kann es auf die Sicherheitsfähigkeiten des Betriebssystems zurückgreifen. Das System nutzt unter Unix etwa die Standardfunktionen `setrlimit()`, um Ressourcen-Limits festzulegen und `chroot()`, um den Zugriff auf das Dateisystem einzuschränken. Eine lokale Firewall regelt die Kommunikation zwischen den Agenten. [Sudmann 98]

F Weitere Lösungen für Caller-ID

In 5.6 *Caller-ID*, Seite 83 wurden einige mögliche Lösungen des Caller-ID-Problems vorgestellt. Einige Zwischenstufen wurden dabei nur kurz erklärt. Hier folgt nun eine detailliertere Darstellung dieser Varianten.

F.1 Variante „Cookie“

Ähnlich der X11-Client-Server-Authentfizierung könnte ein Cookie (beliebige Zufallszahl) zum Nachweis der behaupteten Identität des Absenders dienen. Statt den Message Body intern im Agenten zu halten, steht dort ein Cookie (etwa im Konstruktor gesetzt). Beim Anlegen eines Message-Objekts muss dieses Cookie der Klasse explizit übergeben werden. Die Message-Klasse vergleicht das Cookie dann mit dem Cookie des Absenders. Die Abfrage des Cookies könnte über Package Visibility auf Klassen aus `MobMan.Core` eingeschränkt werden. Solche Klassen darf ein Agent selbst nicht mitbringen (der Classloader verhindert dies), das Cookie wäre so ausreichend geschützt.

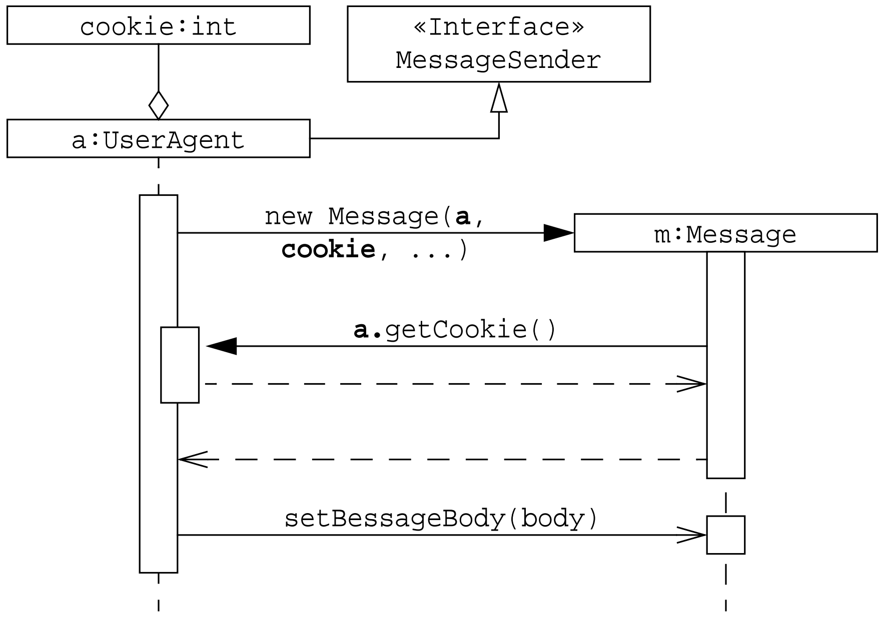

Abbildung F.1 *Caller-ID-Variante „Cookie“*

Vorteile:

- Nachvollziehbares Verfahren.
- Keine Race Condition, der Absender muss wirklich sich selbst angeben, er kann keinem anderen Agenten die Nachricht stehlen (vergleiche 5.6 *Caller-ID*, Seite 83 bis d) *Variante „Rückgriff“*, Seite 86).

Nachteile:

- Aufwand: Die Zufallszahl muss generiert werden.
- Es kann immer nur eine Nachricht gleichzeitig angelegt und ausgefüllt werden. Für mehrere Messages wären auch mehrere Cookies nötig.

Fazit:

- Erfüllt im Wesentlichen die Anforderungen.
- Komplex und aufwändig.

F.2 Variante „Private ID“

Jeder Agent erhält beim Start eine zweite, private ID, die er geheim hält. Die Plattform speichert die Zuordnung der privaten ID zur öffentlich bekannten ID des Agenten. Beim Versenden einer Nachricht muss der Absender neben seiner (angeblichen) ID auch die private ID angeben. Im Prinzip genügt die private ID allein, wenn sie bei der Zustellung durch die öffentliche ID ersetzt wird, entweder vom Message Dispatcher, oder besser von der Message-Klasse selbst.

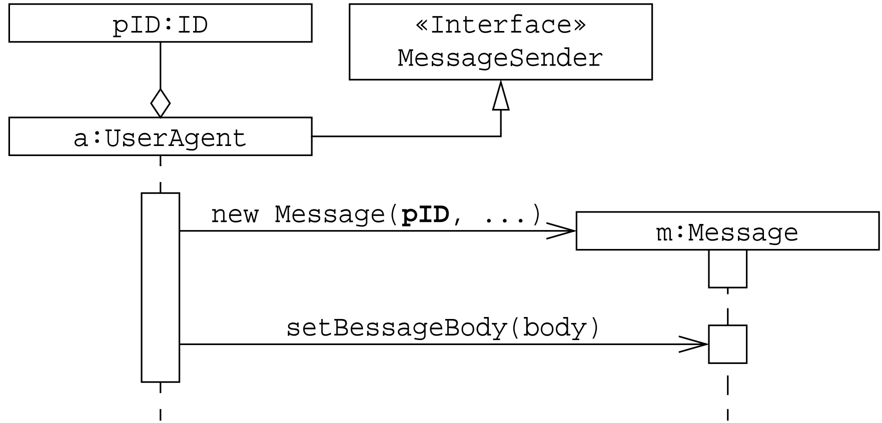

Abbildung F.2 *Caller-ID-Variante „Private ID“*

Vorteile:

- Klarer Aufbau, keine Kunstgriffe wie bei „Rückgriff“.
- Sicher, so lange die private ID geheim bleibt.

Nachteile:

- Komplex durch eine weitere Liste, die es zu verwalten gilt.

F.3 Variante „geschützte Methode und Forwarder-Referenz“

Bei den nun folgenden Ansätzen wird das gemeinsame Interface `MessageSender` durch eine gemeinsame Basisklasse `MobManObject` abgelöst. Für die Absender von Nachrichten ergibt sich dadurch zwar eine Einschränkung, schließlich können sie nicht mehr von beliebigen Klassen erben (Java kennt keine Mehrfachvererbung). In der Praxis ist diese Einschränkung aber kaum relevant, da die Absender in der Regel von MobMan-Klassen erben (Agent, Plugin, Plattform). Im Klassenmodell von MobMan (siehe 4.3 *Klassenmodell*, Seite 40) ist `MobManObject` bereits die Vaterklasse.

In `MobManObject` ist eine als `final` deklarierte Methode enthalten, die zum Senden von Messages benutzt wird. Nur diese Methode kann Nachrichten absenden, weil nur sie die Referenz auf die für das Versenden zuständige Klasse besitzt (als `private` deklariert). Die Methode übergibt die `this`-Referenz des Absenders.

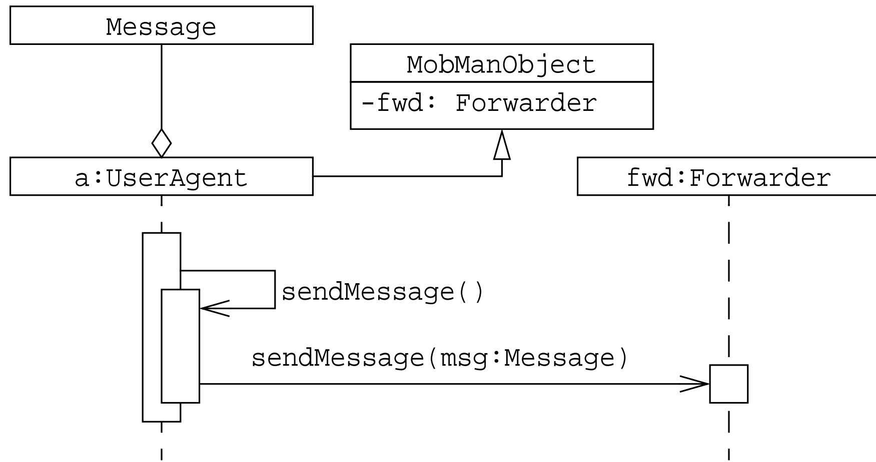

Abbildung F.3 *Caller-ID-Variante „geschützte Methode und Forwarder-Referenz“*

Vorteile:

- Die Basisklasse kann Methoden vorgegeben, die die Kindklassen nicht verändern können. Dadurch andere Prinzipien möglich.
- Durch private Felder in der Vaterklasse können Agenten Daten besitzen, auf die sie selbst keinen Zugriff erhalten.

Nachteile:

- Bislang kümmert sich die `Message`-Klasse selbst um das Versenden. Jetzt müsste es einen zusätzlichen zentralen Forwarder geben, der hierfür zuständig ist.
- Noch ein weiterer Schritt beim Zustellen:
  - Absender
  - Neu: zentraler Forwarder
  - Alle `MessageDispatcher`-Objekte
  - Empfänger

Fazit:

- Die zusätzlichen Forwarder-Klassen stören die Eleganz der Basisklasse.

F.4 Variante „geschützte Methode und Shared Secret“

Statt der geheimen Referenz auf einen Dispatcher bei der vorherigen Variante kann die Sicherung auch über ein Shared Secret stattfinden. Dieses Geheimnis erfährt der Agent beim Start, es ist aber nur in seiner Basisklasse bekannt (wegen der `private`-Deklaration). Jeder Dispatcher vergleicht nun dieses Shared Secret mit dem auch ihm mitgeteilten Wert. Das Verfahren ähnelt also dem Zugriff von privilegierten Systemagenten auf die Plattform.

Entscheidend ist, dass nur die Methode in der Basisklasse dieses Shared Secret kennt, und es auch diese Methode ist, die den `this`-Zeiger angibt. Dadurch ist sichergestellt, dass nicht einfach eine beliebige Referenz eingetragen werden kann.

Wird ein `Message`-Objekt angelegt und soll es direkt aus einem Agenten verschickt werden (ohne eine Methode aus `MobManObject` zu nutzen), ist der Absender anonym. Offen ist noch, ob sich die Nachricht selbst verschickt (aktuell so gelöst), oder ob das Versenden in `MobManObject` implementiert ist, und Message nur als Container für die eigentliche Nachricht dient.

Die `MessageDispatcher`-Implementierungen müssen auch das Shared Secret kennen und es verifizieren. Eventuell könnte die Methode zum Verifizieren auch in der `MobManObject`-Basisklasse implementiert sein.

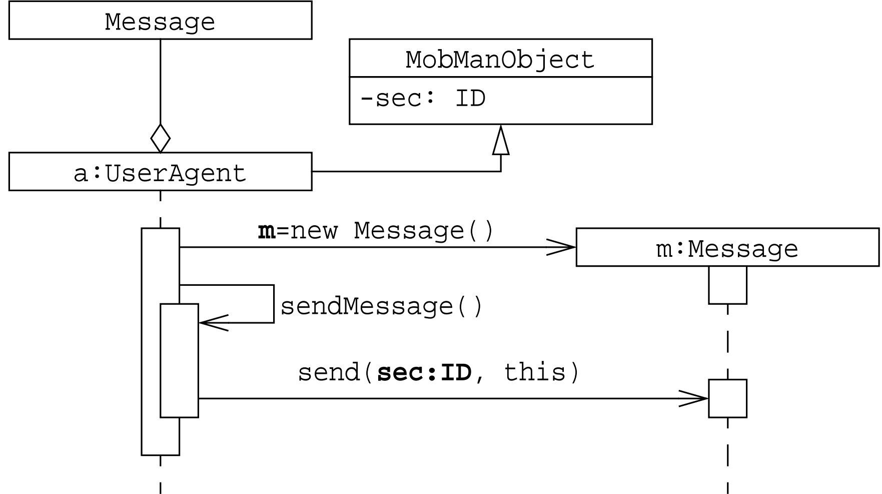

Abbildung F.4 *Caller-ID-Variante „geschützte Methode und Shared Secret“*

Vorteile:

- Kein Forwarder-Objekte mehr nötig.
- Message-Klasse kann sich selbst um das Versenden kümmern.

Nachteile:

- Weiteres Shared Secret.

Fazit:

- Nicht besser als die vorherige Lösung, statt eines Forwarders ist jetzt ein Shared Secret nötig.

G Zuordnungstabellen

An mehreren Stellen in der MobMan-Architektur sind Zuordnungstabellen nötig. Zur besseren Übersicht sind nachfolgend einige wichtige Tabellen aufgeführt.

G.1 Plattform

- `Vector MobManPlatform.agentlist`
  Referenzen aller Agenten der Plattform in der Reihenfolge des Eintreffens.

- `Hashtable MobManPlatform.namelist`
  Cache für die Abbildung von Agentennamen auf ihre IDs. Schlüssel: Name, Wert: Vektor von IDs, ein Name kann mehrfach vorhanden sein kann. Die Abbildung wäre auch über `agentlist` möglich, aber nur durch vollständiges Durchsuchen.
- `Vector MobManPlatform.dispatcher`
  Alle Agenten, die das Interface `MessageDispatcher` implementieren und mit der Plattform gestartet wurden. Ebenfalls ein Cache, als Vektor von Referenzen implementiert.

G.2 Security Agent

- `Hashtable SecurityAgent.agentNames`
  Abbildung von IDs auf authentifizierte Namen. Schlüssel: ID, Wert: Vektor von Namen, unter denen sich ein Agent authentifiziert hat.
- `Hashtable SecurityAgent.agentRights`
  Bildet IDs auf die Rechte der Agenten ab. Schlüssel: ID, Wert: Vektor von Rechten.

G.3 Authentifizierungs-Server

- `TrustedSigners AuthServerCert.signers`
  Liste aller CAs, denen dieses Modul vertraut.
- `MobManConfig AuthServerCert.users`
  `MobManConfig AuthServerPasswd.users`
  Teil des Konfigurationsbaums, der die Benutzer mit ihren Rechten enthält, passend für das jeweilige Authentifizierungsverfahren.

G.4 Pia

- `Hashtable PlatformInterfaceAgent.serviceModules`
  Alle Module, die an diesem Platform Interface Agent angemeldet sind. Unter den Kategorien (Schlüssel der Hashtable) `ServiceServers`, `ServiceClients`, `MasterUIs` und `ServiceUIs` sind Vektoren enthalten (als Werte der Hashtable), in denen die jeweiligen Module aufgeführt sind. Die Elemente dieser Listen enthalten die Informationen der einzelnen Module (wiederum als Hashtable), sie stammen vom Pia Service Agent PISA.
  - `ServiceClassData`: Bytecode des Moduls.
  - `ClassName`: Klassenname dieses Moduls.
  - `GroupName`: Gruppe des Moduls, etwa `View`, `Change`, `Kill` oder `Start`. Dient vor allem der Organisation der Benutzeroberfläche.
  - `SecurityRights`: Aufzählung aller Rechte, die ein Benutzer haben muss, um dieses Modul benutzen zu dürfen. Relevant vor allem bei den Server-Modulen, wird aber auch für Client und UI angewendet.
  - `ServiceClass`: Vom Platform Interface Agent selbst hinzugefügt (Caching); enthält ein Objekt der Klasse `MobManClassStorage` (Bytecode und `Class`-Objekt).
- `Vector PlatformInterfaceAgent.runningServers`
  Liste der aktiven Server-Module.

G.5 Migration Agent

- `Hashtable MigrationAgent.rejectAgents`
  Enthält Agenten, die derzeit nicht auf die Plattform migrieren dürfen. Wird etwa vom Pia-Modul `KillAgentServer` benutzt. Verhindert unter anderem, dass Agenten während des Killvorgangs neu eingespielt werden. Der Name des Agenten steht im Schlüssel der Hashtable, der Wert ist ein Benutzungszähler. Er sorgt dafür, dass Mehrfach-Einträge in der Reject-Liste möglich sind. Nur wenn der Eintrag ebenso häufig entfernt wie vorher angelegt wurde, löscht ihn der Migration Agent aus dieser Liste.

H UML-Diagramme

Die Unified Modeling Language UML [OMG 01] ist ein Industriestandard zur Beschreibung von Softwaresystemen. Verbreitet ist vor allem ihre grafische Notation, die übersichtlich Klassen, Objekte und ihre Interaktionen darstellen. UML definiert eine Reihe von Diagrammtypen. Ihre grafischen Primitive sind möglichst umfassend und semantisch eindeutig definiert, um verschiedenste Aspekte darstellen zu können. Der Standard schreibt aber nicht vor, welche dieser Elemente in einer Beschreibung tatsächlich genutzt werden. Dies hängt vom konkreten Ziel des Diagramms ab.

Diese Arbeit nutzt Klassendiagramme und Interaktionsdiagramme jeweils mit einem Subset ihrer Ausdrucksmöglichten, ergänzt durch kleine Zusätze. Ziel ist es, die Diagramme übersichtlich zu gestalten. Nachfolgend sind die benutzten Symbole zusammengefasst.

H.1 Klassendiagramme

Klassendiagramme beschreiben die statische Struktur von Objekten im Softwaresystem. Zentrales Element sind die Klassen. Sie können in verschiedenen Detail-Stufen enthalten sein, etwa nur der Klassenname oder der Klassenname zusammen mit den Feldern und Methoden. Der Standard schreibt nicht vor, alle tatsächlich vorhandenen Elemente aufzuführen. Auch bei den Bezeichnungen der Methoden sind verschiedene Detail-Grade möglich: Ihre Parameter können ausgeblendet, nur mit Namen aufgeführt oder zusammen mit ihrem Typ dargestellt sein. Ein Diagramm und seine Bestandteile sollten gerade so ausführlich sein, dass sie die beabsichtigte Aussage verdeutlichen.

Die Gruppierung der Klassen in Paketen stellen laut UML eigene Package-Diagramme dar. Die Beziehungen der Pakete sind durch Pfeile angedeutet. Die vorliegende Arbeit integriert die Pakkage-Diagramme an einigen Stellen direkt in die Klassendiagramme. Packages dienen hier nicht nur der logischen Gruppierungen, sie erfüllen auch Sicherheitsaufgaben. Daher ist eine losgelöste Darstellung wenig sinnvoll. Um die Diagramme nicht zu unübersichtlich zu gestalten, sind die Beziehungen der Packages nicht durch Pfeile, sondern durch Einbetten der Pakkage-Grafiken ineinander dargestellt.

Symbol	Bedeutung
Agent	Die Klasse `Agent`, hier ohne innere Details.
«Interface» MessageDispatcher	Das Interface `MessageDispatcher`. Interfaces sind eine Java-Besonderheit: Sie sind rein abstrakte Basisklassen und ersetzen die Mehrfachvererbung anderer Sprachen.
MobManObject / Felder / Methoden	Die Klasse `MobManObject` mit ihren Feldern und Methoden. Wie diese beschrieben werden, fasst Tabelle H.3 *UML: Felder und Methoden*, Seite 171 zusammen.
MobMan.Core / Agent	Das Java-Package `MobMan.Core` enthält die Klasse `Agent`.
eigener Thread	Kommentar.

Tabelle H.1 *UML: Klassen und Pakete*

Striche und Pfeile symbolisieren die Beziehungen der Klassen untereinander. Verschiedene Stricharten und Pfeilspitzen bezeichnen die Art der Beziehung, etwa Vererbung, Aggregation oder eine allgemeine Assoziation. Beschriftungen an den Pfeilen spezifizieren die Beziehung genauer, siehe Tabelle H.2 *UML: Beziehungen der Klassen*, Seite 170.

Symbol	Bedeutung
MobManObject ◁— Agent	Die Pfeilspitze entscheidet über die Art der Beziehung beider Klassen. Hier erbt `Agent` von `MobManObject`. Der Pfeil zeigt in Richtung der Vaterklasse, die Spitze ist nicht ausgefüllt.
ID 1 —◇ 2 MobManPlatform	Auch die Aggregation ist gerichtet: `MobManPlatform` enthält `ID`. Die Kardinalität der Aggregation kann durch Zahlen und Symbole an den Enden des Pfeils dargestellt sein: • Eine exakte Anzahl ist direkt als Zahl enthalten: `2` • Beliebig oft, auch Null: `*` • Bereiche: `1..3` Auch das Ende, an dem die Zahl steht, enthält eine Aussage: `MobManObject` enthält hier zwei einzelne Referenzen auf je ein `ID`-Objekt. Bei einem Array wäre an der Raute die Zahl 1, am anderen Ende die Zahl der Elemente aufgeführt.
Message — < sendet — Agent	Eine allgemeine Assoziation zweier Klassen ist durch eine Linie ohne Pfeilspitzen dargestellt. Die Art der Beziehung kann beschreibender Text näher erläutern. Vor der Beschreibung kann ein Größer- oder Kleiner-Zeichen (als stilisierter Pfeil) zeigen, auf welches Ende der Linie sie sich bezieht.

Tabelle H.2 *UML: Beziehungen der Klassen*

In der Beschreibung einer Klasse können die einzelnen Felder und Methoden spezifiziert sein. Hierfür benutzt UML eine eigene, sprachunabhängige Syntax. Sie stellt die Namen, Parameter, Typen und Zugriffsregeln dar, siehe Tabelle H.3 *UML: Felder und Methoden*, Seite 171.

Feld oder Methode	Bedeutung
`agentlist`	Das Feld (die Variable) `agentlist`. Typ und Zugriffsrechte sind nicht näher spezifiziert.
`agentlist: Vector`	Das Feld `agentlist` ist vom Typ (Klasse) `Vector`.
`findAgents()`	Die Methode heißt `findAgents`. Rückgabetyp, Parameter und Zugriffsrechte sind nicht näher spezifiziert.
`findAgents(name)`	Die Methode `findAgents` hat einen Parameter namens `name`.
`findAgents(name: String)`	Der Typ des Parameters `name` ist `String`.
`findAgents(name: String): Vector`	Die Methode `findAgents` liefert als Rückgabewert ein Objekt der Klasse `Vector`.
`findAgents(name: String, sn: int)`	Die Methode `findAgents` hat zwei Parameter: `name` ist von Typ `String`, der Parameter `sn` ist ein `int`-Wert. Der Rückgabewert ist nicht näher spezifiziert.

Tabelle H.3 *UML: Felder und Methoden*

Die Zugriffsregeln auf eine Methode oder ein Feld (Sichtbarkeit) stehen als Präfix unmittelbar vor dem Namen. UML definiert für die klassischen Varianten `private`, `protected` und `public` eigene Zeichen (siehe Tabelle H.4 *UML: Sichtbarkeit von Feldern und Methoden*, Seite 171). Seit Version 1.4 ist auch `package` im Standard enthalten. Die Bedeutung der Sichtbarkeitsregeln stellt Tabelle 5.1 *Kapselung*, Seite 77 dar.

Symbol	Bedeutung
+	`public`
#	`protected`
~	`package`
-	`private`
*	`final` (eigene Ergänzung)

Tabelle H.4 *UML: Sichtbarkeit von Feldern und Methoden*

Leider kennt der UML-Standard kein Symbol, um als `final` deklarierte Methoden hervorzuheben. Es ist nur möglich, *non-final*-Methoden darzustellen, und zwar durch kursive Schrift. Da überschreibbare Methoden der Regelfall sind und `final` die Ausnahme, erscheint diese Darstellung als wenig geeignet. Zudem ist Kursivschrift häufig kaum von gerader Schrift zu unterscheiden. Da `final` eine wichtige Rolle für die Sicherheit von MobMan spielt, benutzen die Diagramme in dieser Arbeit hierfür ein eigenes Symbol: „`*`“. Dieser Stern ist hinter dem Symbol für die Sichtbarkeit angeordnet.

H.2 Sequenzendiagramme

Sequenzendiagramme sind eines der Kollaborationsdiagramme des UML-Standards. Diese Arbeit benutzt sie nur in ihrer Ausprägung mit Focus of Control, in der der Kontrollfluss deutlich zu sehen ist.

Sequenzendiagramme stellen ein Fallbeispiel im zeitlichen Ablauf dar. Die Zeitachse ist senkrecht von oben nach unten angeordnet. Nebeneinander sind die Instanzen (Objekte) aufgeführt, die miteinander interagieren. Ihre Kommunikation ist durch Pfeile dargestellt. Da es sich hierbei um Methodenaufrufe handelt, muss zwischen Aufruf und Rückgabe unterschieden werden. Zur Vereinfachung sind auch asynchrone Aufrufe darstellbar, bei diesen verbleibt der Kontrollfluss beim Aufrufer. Siehe Tabelle H.5 *UML: Sequenzendiagramme*, Seite 173.

In Ergänzung zu UML zeigen manche Grafiken in dieser Arbeit eine Kombination aus Sequenzen- und Klassendiagramm. Die Objekte am Kopf der einzelnen Ablaufachsen sind teils zusammen mit ihren Feldern und Methoden dargestellt, teils sind auch die Vererbungsbeziehungen in die Abbildungen integriert.

Ein weitere Ergänzung sind die MobMan-Nachrichten (Messages), die auch zu einer Kommunikation der Objekte führen. Messages lassen sich auf Methodenaufrufe zurückführen; die Details dieses Ablaufs zu verbergen ist eines der Ziele der MobMan-Architektur. Dem tragen die Darstellungen Rechnung und zeigen Nachrichten als Methodenaufruf von Absender zu Empfänger, ergänzt durch einen stilisierten Brief in der Mitte des Pfeils.

Symbol	Bedeutung
a:UserAgent	Die gestrichelte senkrechte Linie stellt den zeitlichen Ablauf des Objekts `a` vom Typ `UserAgent` dar. Das Objekt existiert bis zum Ende der Linie. Das langgezogene Rechteck symbolisiert den Kontrollfluss. Während dieser Zeit ist das Objekt aktiv. Als Aktivität gilt hier auch das Warten, bis eine (synchron) aufgerufene Funktion zurückkehrt.
call()	Synchrone Aufrufe sind mit einer horizontalen durchgezogenen Linie mit ausgefüllter Pfeilspitze dargestellt. Der Name der Methode `call()` steht über dem Pfeil. Der Rücksprung ist als gestrichelte Linie mit offenem Pfeilende gezeichnet. Der Rückgabewert könnte über diesem Pfeil genannt sein. Während der Arbeit in der Methode `call()` ist auch das rechte Objekt aktiv, zu sehen am vertikalen Rechteck.
call()	Interessiert der Rückgabewert nicht und springt die Methode unmittelbar zum Aufrufer zurück, ist die Darstellung als asynchroner Aufruf einfacher. Der horizontale Pfeil ist durchgezogen, die Pfeilspitze offen.
call()	Kehrt der Kontrollfluss rekursiv zur ursprünglichen Funktion zurück, dann symbolisiert ein weiteres Rechteck diese zweite Aufrufebene.
s:SystemAgent	Durch Nebenläufigkeiten im Code (Threads) kann es vorkommen, dass ein Objekt mehrere Kontrollflüsse gleichzeitig ausführt. In diesem Fall gabelt sich die vertikale, gestrichelte Linie und jeder Kontrollfluss erhält sein eigenes Aktivitäts-Rechteck.
a new() b	Objekt `a` erzeugt Objekt `b` durch einen `new()`-Aufruf. Ist der Ablauf im Konstruktor interessant, kann die Darstellung von synchronen Aufrufen übernommen werden, das Aktivitätsrechteck beginnt dann unmittelbar unter dem Objekt, auch der Rücksprung ist dargestellt. Ein fett gezeichnetes Kreuz kennzeichnet das Ende der Lebenszeit eines Objekts.

Tabelle H.5 *UML: Sequenzendiagramme*