INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Authentisierung und Autorisierung für das Java/CORBA-Agentensystem MASA

Harald Rölle

INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Authentisierung und Autorisierung für das Java/CORBA-Agentensystem MASA

Bearbeiter:	Harald Rölle
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Boris Gruschke Helmut Reiser

Abgabetermin: 15. August 1999

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

München, den 15. August 1999

. . . .
(Unterschrift des Kandidaten)

Zusammenfassung

Mit der wachsenden Komplexität von IT-Infrastrukturen rückte in der Vergangenheit das Problem des Managements solcher Strukturen immer weiter in den Vordergrund. Mit der Anwendung von Konzepten von flexiblen und autonomen Agenten, in Kombination mit dem Management-by-Delegation Paradigma, ist eine solide Basis zur Berherrschung dieser Komplexität gefunden worden. Dabei war die Sicherheit der Managementsysteme immer ein wichtiges Moment. Kommen Systeme mobiler Agenten, auf obiger Basis aufbauend, im Management zum Einsatz, wird die Sicherheit zu einem kritischen Faktor.

Diese Arbeit definiert für das CORBA/Java basierte mobile Agentensystem MASA ein Sicherheitsmodell.

Beginnend mit einer prinzipiellen Analyse der Sicherheitsanforderungen an ein System mobiler Agenten, gefolgt von einer konkreten Untersuchung von MASA in Konzeption und Implementierung, wird ein detailliertes Sicherheitsmodell für MASA entwickelt. Dabei wird nicht nur den gestellten konzeptionellen Anforderung Rechnung getragen, es wird auch die Konformität zu den MASA zugrundeliegenden Standards betrachtet.

Die Realisierbarkeit der in dem Sicherheitsmodell definierten Mechanismen zur Authentisierung und Autorisierung wird dann durch eine Erweiterung der bestehenden MASA-Implementierung unter Beweis gestellt. Dabei kommen etablierte Techniken wie X.509-Zertifizierungshierarchien, SSL und Java 1.2 zum Einsatz.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Das Management von IT-Infrastrukturen gewann in den letzten Jahren stetig an Bedeutung. Ursachen sind die permanent wachsende Komplexität von Netzen, Systemen und Anwendungen, hervorgerufen durch die Zusammenfügung heterogener Hardware- und Software-Komponenten, sowie Dienstleistungen, zu komplexen Wertschöpfungsketten. Katalysierend für das Komplexitätswachstum wirkt dabei die enorme Geschwindigkeit der technischen Entwicklung.

Die folgende sehr kurze Einführung in die Problematik des Managements solcher komplexen IT-Systeme diene als Einstieg in die Materie. Eine ausführliche Darstellung der gesamten Managementproblematik findet man in [HAN 99a].

Klassische Managementansätze erweisen sich als zunehmend inadäquat. Sie basierten auf einem starren Konzept, welches aufbauend auf einer Managementplattform dedizierte Managementanwendungen realisiert. Die Managementplattform stellt dabei u. a. die Kommunikationsplattform dar, über die Managementanwendungen auf die durch einen Managementagenten repräsentierte Managed Resource zugreifen. Die Funktionalität der Agenten (und damit die Managementfunktionalität) ist dabei fest definiert und nach der Installation nicht mehr änderbar, wodurch während der Systemlaufzeit keine Anpassungen mehr vorgenommen werden können.

Die dadurch entstandenen starren Strukturen konnten aber nicht mehr mit dem schnellen Fortschreiten der Technologie schritthalten, die ständig neue Funktionalität mit sich brachte und somit immer neue Anforderungen an ein Managementsystem stellte.

Nachdem klassische Managementansätze den notwendigen Bedürfnissen nicht mehr gerecht werden konnten und andere (statische) Ansätze nicht sehr erfolgreich waren, wurde in [Moun 97] der neue Ansatz des integrierten Managements mittels des Management-by-Delegation-Paradigmas durch "flexible Agenten" vorgestellt. "Flexible Agenten" zeichnen sich dadurch aus, daß sie autonom handlungsfähig sind und selbständig mit anderen Agenten kommunizieren und kooperieren können. Mit der Erweiterung der flexiblen Agenten um die Fähigkeit sich frei im Netz zu bewegen, entstand der Begriff des "mobilen Agenten". Ihre Kommunikationsfähigkeit und Mobilität ermöglicht die Erstellung eines verteilten Managementsystems, in dem die Managementfunktionalität während der Laufzeit dynamisch verändert werden kann.

Mit [Kemp 98] wurde die Mobile Agent System Architecture (MASA), eine Plattform für mobile Agenten, konzeptionell entwickelt und deren Implementierung realisiert. Dabei fokussiert

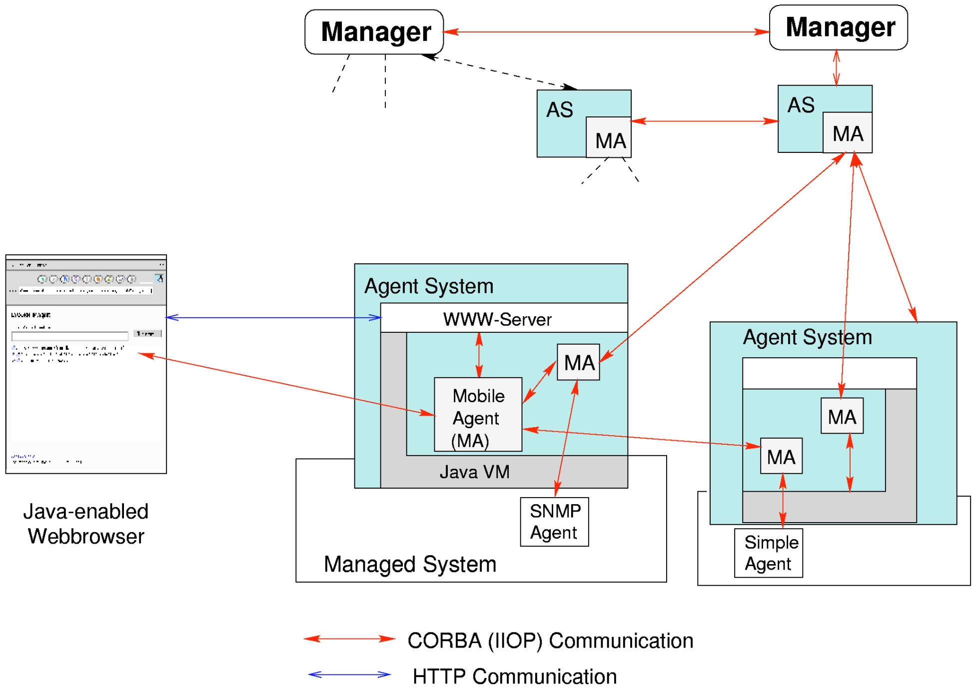

Abbildung 1.1: MASA Architektur (aus [GHR 99])

[Kemp 98] Fragen der prinzipiellen Architektur, die Implementierung stellt eine Machbarkeitsstudie dar.

Für die Akzeptanz von Managementsystemen ist die Zusicherung und Erfüllung von Sicherheitseigenschaften Voraussetzung. Durch den verteilten, dynamischen Ansatz mobiler Agenten werden jedoch neue Fragen der Sicherheit aufgeworfen, die eine Entwicklung eines eigenen Sicherheitskonzepts nötig machen. Mit dieser Arbeit wird ein Sicherheitsmodell für MASA entwickelt. Dabei werden nicht nur die notwendigen konzeptionellen Arbeiten für die Erstellung eines solchen Modells durchgeführt, auch dessen Realisierbarkeit wird durch eine Erweiterung der MASA-Implementierung belegt.

Zur Einführung in die Aufgabenstellung wird im Folgenden zunächst ein kurzer Überblick über MASA gegeben und anschließend Notwendigkeit und Problematik der Sicherheit an einem Beispiel motiviert.

1.1 Überblick über die "Mobile Agent System Architecture" (MASA)

MASA ist eine plattformunabhängige Laufzeit- und Kommunikationsarchitektur für mobile Agenten. In diesem Kapitel wird nur ein knapper Überblick über MASA gegeben. Die in Abbildung 1.1 veranschaulichte MASA-Architektur und ihre im Folgenden kurz beschriebenen Komponenten werden in [GHR 99] ausführlich dargestellt.

Auf einem zu managenden Endsystem (managed system) wird ein Agentensystem (agent system) ausgeführt, das die Laufzeitumgebung für alle Agenten (mobile agents) auf dem

Endsystem darstellt.

Agentensystem und Agenten sind in der Programmiersprache Java ([JLS]) implementiert. Basis des Agentensystems und Abstraktionsschicht von plattformabhängigen Teilen des Endsystems ist die *Java Virtual Machine*, die Java-eigene Laufzeitumgebung. In ihr eingebettet werden die Agentensysteme ausgeführt.

Agenten können mit anderen Agenten kommunizieren und selbständig zu anderen Agentensystemen migrieren. Dabei erfolgt die gesamte Kommunikation über die *Common Object Request Broker Architecture (CORBA)* ([CORBA 2.2]). Weiterhin ist ein eigener Webserver Teil eines jeden Agentensystems. Dieser wird sowohl in der Kommunikation zur Steuerung der Agenten, als auch des Agentensystems selbst, eingesetzt.

Die Benutzeroberfläche von Agentensystemen und Agenten wird durch Applets realisiert, die ebenfalls in Java implementiert sind. Will ein Benutzer auf einen Agenten zugreifen, fordert er vom Webserver des Agentensystems über das HTTP-Protokoll das Applet des Agenten an. Dieses wird dann auf seinen Webbrowser übertragen und dort ausgeführt. Im Folgenden kommuniziert dann das Applet via CORBA direkt mit dem zu steuernden Agenten.

1.2 Die Sicherheitsproblematik anhand eines Beispiels

Um die Sicherheitsprobleme aufzeigen zu können, die mit einem System mobiler Agenten im Managementeinsatz einhergehen, wird das Beispiel einer Managementanwendung nach [HaRe 99] in Abb. 1.2 betrachtet.

Ein Automobilhersteller arbeitet mit einer (großen) Zahl weitverstreuter, unabhängiger Händler zusammen. Der Hersteller möchte nun seinen Händlern Dienste, die durch sein Firmennetz realisiert sind, zur Verfügung stellen, z.B. die Möglichkeit zur Online-Bestellung in seinem Bestell-Center.

Die Realisierung der hierfür notwendigen Vernetzung übernimmt ein Provider. Über das Netz des Providers, zu dem die Händler über Wählleitungen angeschlossen werden, wird die Verbindung zum Firmennetz des Herstellers realisiert. Im Netz des Providers, das gleichzeitig auch von anderen seiner Kunden genutzt wird, bilden Hersteller und Händler dann ein Virtual Private Network (VPN).

Mit dem Provider werden Vereinbarungen getroffen, welche die Güte der Netzverbindungen betreffen. Hierzu werden in einem sog. Service Level Agreement (SLA) dezidierte Quality of Service (QoS) Parameter festgelegt, z.B. der Netzdurchsatz vom Händler zum Bestell-Center des Herstellers.

Um die Einhaltung der geforderten QoS-Parameter gewährleisten zu können, muß das Management-Center des Providers die Möglichkeit haben, die relevanten QoS-Parameter zu messen. Häufig ist das nur vom Händler aus sinnvoll möglich. So kann auch für das angegebene Beispiel der Durchsatz für die *komplette* Strecke vom Händler zum Bestell-Center am einfachsten direkt vom Händler aus gemessen werden. Weiterhin muß das Management Center flexibel auf Veränderungen reagieren können, wenn beispielsweise neue QoS-Parameter in die Vereinbarung aufgenommen werden.

Klassische Managementplattformen sind, wegen ihres statischen Ansatzes, zu unflexibel um völlig neue QoS-Parameter schnell und einfach handhaben zu können. Sie scheiden somit

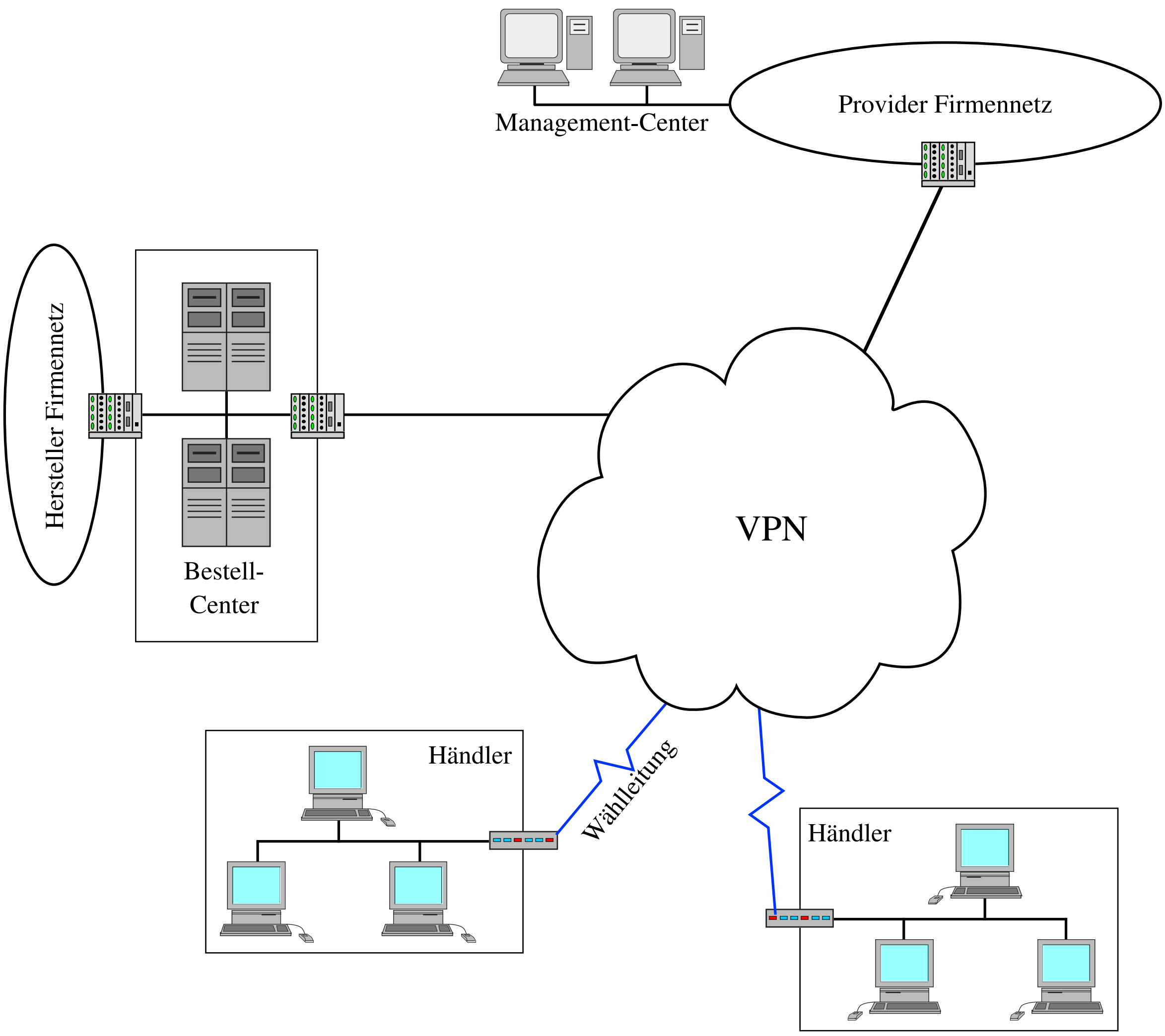

Abbildung 1.2: Skizze des Einführungsbeispiels

für die Realisierung aus. Mobile Agenten dagegen bieten genau die geforderten Möglichkeiten. Für die Durchsatzermittlung beispielsweise, kann das Management-Center des Providers einen mobilen Agenten aussenden, der händlerseitig die gewünschten Messungen durchführt. Sollen neue QoS-Parameter gemessen werden, wird einfach ein passender Agent zusätzlich zum Händler geschickt.

Anhand des geschilderten Szenarios lassen sich charakteristische Sicherheitsprobleme aufzeigen. Hierzu ist zunächst festzustellen, daß die Beteiligten für unterschiedliche Bereiche verantwortlich sind:

- Der Automobil-Hersteller für sein Firmennetz und die Anwendung "Online-Bestellung" im Bestell-Center.
- Der Provider für die Netzverbindung von den Händlern zum Hersteller und die Einhaltung der im zugehörigen SLA festgelegten QoS-Parameter.
- Der Händler für seine internes LAN.

Korrespondierend zu diesen Veranwortlichkeiten gibt es entsprechende Personen:

- Der Anwendungs-Administrator für die Anwendung "Online-Bestellung" beim Herstel-

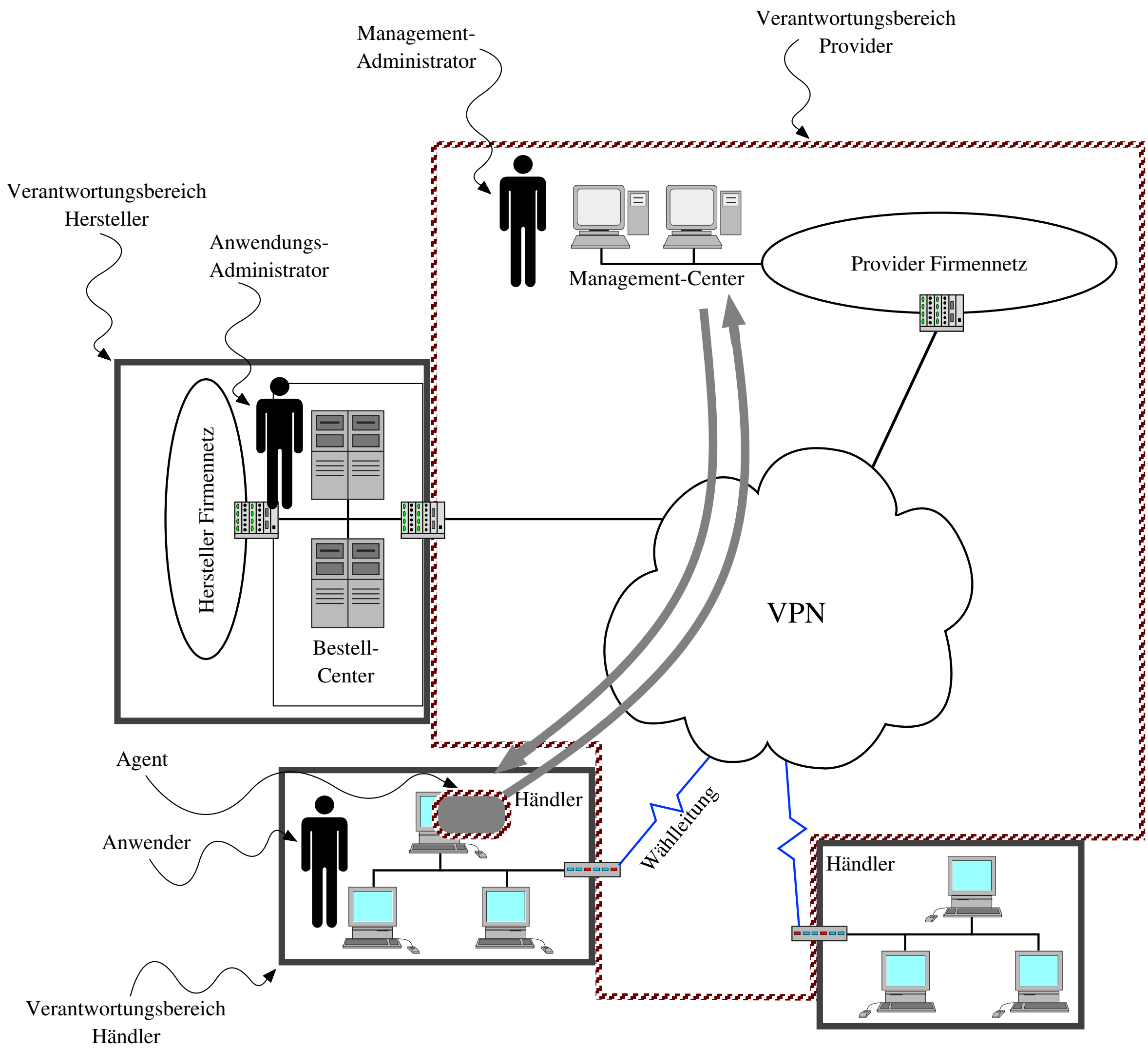

Abbildung 1.3: Einführungsbeispiel mit Verantwortungsbereichen

ler.

- Der Management-Administrator beim Provider.
- Der Anwender (z.B. ein Verkäufer) beim Händler.

Wird nun der Agent zur Durchsatzmessung durch einen Management-Administrator zu einem Händler geschickt (Abb. 1.3), so "betritt" damit eine Komponente den Verantwortungsbereich eines "Fremden": Der Durchsatz-Meß-Agent, der zum Provider gehört, betritt ein Endsystem des Händlers.

Hieraus resultieren potentielle Sicherheitskonflikte:

- Der Händler möchte nicht, daß der Agent interne Daten ausspähen kann und die Arbeit an seinen Endsystemen behindert.
- Der Management-Administrator möchte, daß der Agent ungestört seine Messungen durchführen und mit den ermittelten Daten zum Management-Center zurückkehren kann.

- Der Management-Administrator will sich auf die gelieferten Daten verlassen können. Denn falls die Daten fälschlicherweise eine Verletzung des einzuhaltenden Schwellwerts anzeigen, könnte das z.B. bedeuten, daß der Händler einen (unberechtigten) Preisnachlaß bekommt.

Zur Lösung dieser Konflikte müssen folgende typische Fragen beantwortet werden können:

- Aus der Sicht des Händlers:
  - Wurde der Agent wirklich von einem Management-Administrator des Providers ausgesandt, und kann ihm deshalb die Messung erlaubt werden, oder ist es ein Agent eines anderen Kunden des Providers?
  - Handelt es sich tatsächlich um einen Agenten zur Durchsatzermittlung oder um einen Virus?
  - Kann der Agent tatsächlich nur den Durchsatz messen und hat er ansonsten keinen Zugriff auf interne Firmendaten?
- Aus der Sicht des Providers:
  - Ist der benutzte Agent tatsächlich vom angegebenen Hersteller implementiert worden und erfüllt er prinzipiell nur seine angegebene Aufgabe?
  - Wird die Messung auch tatsächlich vom Endsystem des richtigen Händlers aus durchgeführt und nicht etwa von einem anderen Endsystem aus?
  - Sind die Meßdaten, die der Agent zurückbringt, auch tatsächlich die am Endsystem gemessenen, oder wurden sie beim Händler oder auf dem Weg zurück zum Management-Center verändert?

Die grundsätzliche Ursache für alle genannten Fragen liegt in der Tatsache, daß die beteiligten Komponenten (Firmen, Personen, Agenten, Endsysteme) sich nicht bedingungslos vertrauen können. Um aber ein, zumindest partiell gültiges, Vertrauensverhältnis zwischen den beteiligten Komponenten herzustellen und somit die Fragen beantworten zu können, sind im wesentlichen die im Titel dieser Arbeit genannten Mechanismen notwendig:

- Authentisierung: Die Identität beteiligter Komponenten muß nachgewiesen werden können.
  - Der Agent muß belegen können, daß er im Auftrag des Management-Administrators des Providers tätig ist.
  - Der Agent muß seinen Hersteller, der ihn implementiert hat, belegen können.
  - Das Endsystem beim Händler muß dem Agenten seine Identität belegen können.
  - Die gemessenen Daten müssen nachweislich vom Agenten erhoben worden und unverändert geblieben sein.
- Autorisierung: Die Nutzung bestimmter Funktionalitäten muß gestattet und überwacht werden.
  - Der Agent darf nur Durchsatzmessungen ausführen, ansonsten sind keine anderen Aktionen zu gestatten.

Für das Management der Agenten selbst wiederum bedeutet diese Struktur, daß die durch Netztopologien bzw. die örtliche Anordnung der Komponenten (wie Endsystem beim Händler oder Host im Management-Center) gegebenen Domänen im Hinblick auf die Sicherheit unbrauchbar sind: Der Agent zur Durchsatzmessung befindet sich zwar zeitweise im LAN des Händlers und zeitweise im Firmennetz des Providers, dennoch gehört der Agent aus sicherheitstechnischer Sicht immer zum Provider. Somit ist ein Sicherheitsdomänenbegriff zu definieren, der sich an den Organisationsstrukturen der beteiligten Komponenten orientiert.

Mit der Entwicklung eines Sicherheitsmodells soll in dieser Arbeit für und durch das Agentensystem MASA eine sichere "Umgebung" geschaffen werden, in der die oben aufgeworfenen Fragen zuverlässig beantwortet werden können.

1.3 Vorgehensweise in dieser Arbeit

Im Folgenden wird der Aufbau dieser Arbeit kurz beschrieben. Die Struktur der Kapitel ist in Abbildung 1.4 skizziert.

Zu Beginn der Arbeit steht eine allgemeine Analyse der Anforderungen an ein Sicherheitsmodell für Systeme mobiler Agenten. Zur Schaffung einer einheitlichen Betrachtungsweise und Nomenklatur wird anfangs ein einfaches Modell von Systemen mobiler Agenten vorgestellt. Durch die Untersuchung möglicher Gefahren, der jede verteilte Anwendung ausgesetzt ist, wird das generelle Gefahrenpotential aufgezeigt und anhand der Darstellung typischer Angriffstechniken deren Relevanz belegt. Im Anschluß werden die allgemeinen Gefahren in verteilten Anwendungen durch Betrachtung der speziellen Probleme, wie sie durch mobile Agenten aufgeworfen werden, ergänzt. Bereits hieraus leitet sich unmittelbar die Notwendigkeit von Authentisierungs- und Autorisierungsmechanismen ab. Anforderungen, die diese Mechanismen in einem System mobiler Agenten erfüllen müssen, markieren das Ende von Kapitel 2.

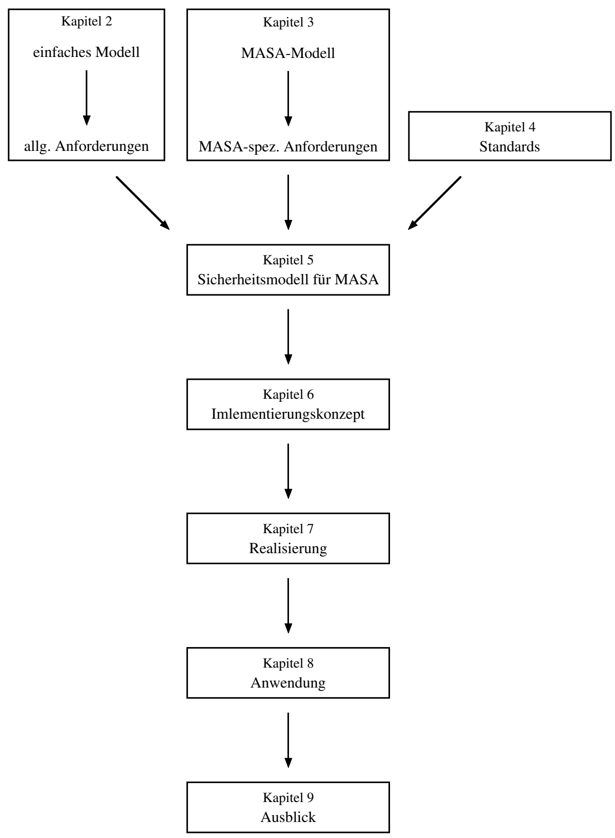

Abbildung 1.4: Skizze der Vorgehensweise

In Kapitel 3 werden dann, durch Erweiterung des einfachen Modells aus der Anfordungsanalyse zum MASA-Modell, die konkreten Risiken für MASA analysiert. Hierzu werden sowohl konzeptionelle Eigenschaften, die sich im MASA-Modell widerspiegeln, als auch konkrete Details aus der Implementierung betrachtet. Unter Einbeziehung der aus Kapitel 2 gewonnenen Erkenntnisse, ergeben sich, durch Betrachtungen aus verschiedenen Sichtweisen, die konkreten Risiken in MASA sowie die MASA-spezifischen Anforderungen an ein Sicherheitsmodell.

Da auch seitens der Standards, auf denen MASA basiert, weitere Rahmenbedingungen gesetzt werden, werden diese in Kapitel 4 aufgezeigt, und durch einen Überblick über bestehende Systeme, die diese Standards nutzen, ergänzt.

Aus den in Kapitel 2 gewonnen allgemeinen Anforderungen, den konkreten Anforderung in Kapitel 3 und den Rahmenbedingungen durch die Standards wird in Kapitel 5 ein Sicherheitsmodell für MASA entwickelt. Zu diesem Zweck werden u. a. die während der Risikoanalyse von MASA gewonnenen Sichten aufgegriffen, anhand derer die einzelnen Teile des Modells entwickelt werden. Dabei werden konkrete Mechanismen zur Authentisierung und Autorisierung entwickelt.

Ein Konzept zur Umsetzung des entwickelten Sicherheitsmodells unter Verwendung konkret verfügbarer und im Rahmen der bestehenden MASA-Implementierung einsetzbarer Techniken, wird in Kapitel 6 angegeben.

Die im Rahmen dieser Arbeit entstandene Implementierung des Sicherheitsmodells wird in Kapitel 7 beschrieben.

Anschließend werden in Kapitel 8 den Entwicklern von Agenten konkrete Richtlinien an die Hand gegeben, die zur Implementierung sicherer Agenten zu beachten sind.

Schließlich wird die Arbeit in Kapitel 9 durch eine Zusammenfassung abgerundet und mit einem Ausblick auf mögliche zukünftige Arbeiten abgeschlossen.

Grundlagen zum Verständnis

Zum Verständnis dieser Arbeit werden Grundkenntnisse aus den Bereichen

- Kryptologie – speziell Public-Key-Verfahren
- CORBA
- Java

vorausgesetzt.

Kapitel 2

Anforderungsanalyse

Um ein Sicherheitskonzept für MASA entwickeln zu können, sollen zunächst einmal allgemeine Anforderungen an ein solches Konzept im Kontext von mobilen Agenten betrachtet werden. Hierzu wird zunächst ein einfaches Modell vorgestellt, anhand dessen mögliche Gefährdungen und die dazu nutzbaren Angriffstechniken aufgezeigt werden, woraus sich anschließend notwendige Forderungen an ein Sicherheitsmodell ergeben.

2.1 Einfaches Modell von Systemen mobiler Agenten (SmA)

Um darstellen zu können, welchen Gefahren ein System mobiler Agenten (SmA) ausgesetzt ist, sei hier ein abstraktes Modell in Anlehnung an [OMG 98-03-09] betrachtet, welches in

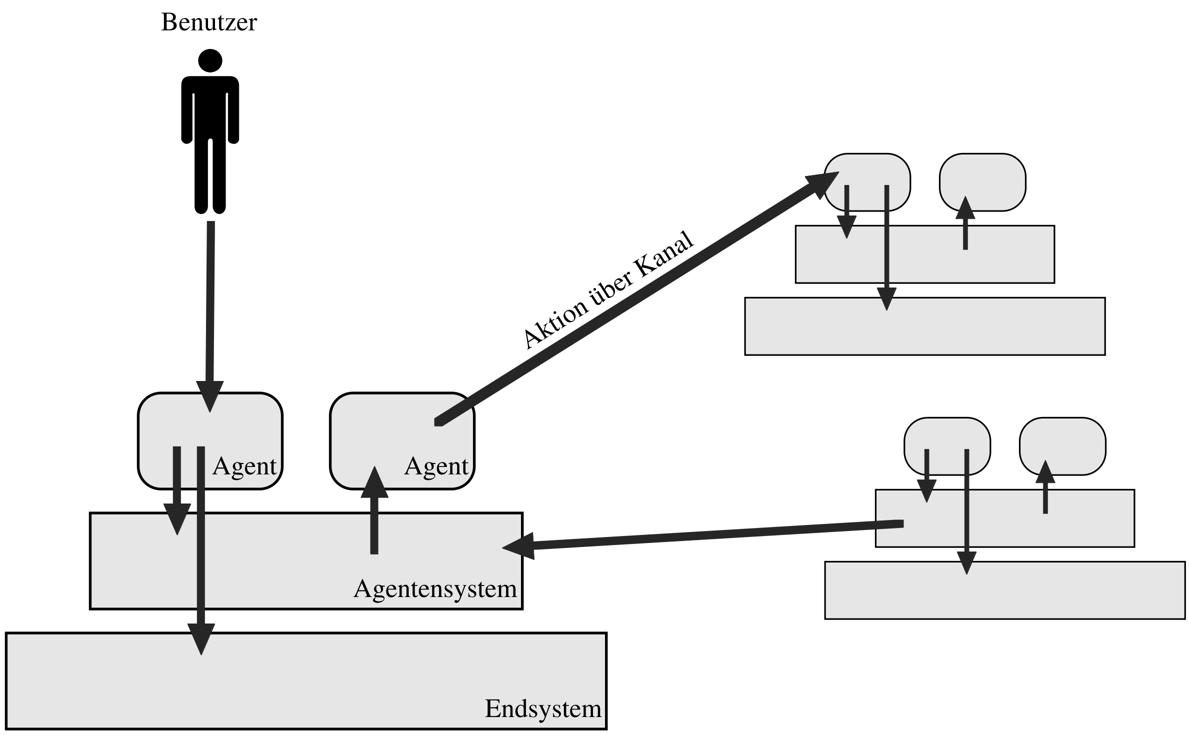

Abbildung 2.1: Skizze des einfachen Agentensystem Modells

Abbildung 2.1 schematisch dargestellt wird. In diesem Modell sind

- Benutzer
- Agenten (Programmcode und dynamische Daten)
- Agentensysteme
- Endsysteme

die handelnden Entitäten, die

- Aktionen

auf einer anderen Entität ausführen. Dabei sind Agentensysteme die Laufzeitumgebungen für einen oder mehrere Agenten; Endsysteme sind die Laufzeitumgebung von Agentensystemen.

Die für die Durchführung einer Aktion notwendige Kommunikation wird über

- Kanäle

ausgeführt. Agenten, Agentensysteme und Endsysteme sind jene Entitäten, die Schnittstellen besitzen, die von anderen Entitäten über Kanäle angesprochen werden können, um dort Aktionen auszuführen.

2.2 Mögliche Gefahren

Bei der Betrachtung der Bedrohungen, mit denen Entitäten eines SmA konfrontiert sind, können drei prinzipielle Typen unterschieden werden. Dabei können diese Bedrohungen sowohl von Entitäten des Modells als auch von außenstehenden Dritten ausgehen (nach [OMG 98-03-09]):

Nichtautorisierter Zugriff oder Benutzung: Ein Benutzer, ein Agent oder ein Agentensystem führt unerlaubt eine Aktion auf einem Agenten, Agentensystem oder Endsystem aus.

Nichtautorisiertes Modifizieren oder Zerstören von Daten: Interne und öffentliche Daten eines Agenten, Agentensystems oder Endsystems werden unberechtigt verändert, zerstört oder es werden unberechtigt neue Daten hinzugefügt.

Denial of Service (DoS): Die Verfügbarkeit eines Endsystems, Agentensystems, Agenten oder eines Teils dieser Entitäten wird unberechtigt reduziert.

Aus diesen Bedrohungen leiten sich nun einige Aspekte des Begriffes “Sicherheit” im Kontext von SmA ab ([OMG 98-12-17], Kap. 15.1.2). Zur Verdeutlichung dieser Aspekte wird ein Agent zum Management von Routing-Tabellen betrachtet:

Vertraulichkeit: Informationen werden nur berechtigten Entitäten preisgegeben.

Beispiel: Der Agent kann nur Daten der Routing-Tabelle einsehen. Er kann beispielsweise nicht die Daten eines auf dem gleichen Endsystem ablaufenden DHCP-Servers einsehen.

Vollständigkeit/Integrität: Informationen werden ausschließlich von berechtigten Entitäten verändert. Veränderungen erfolgen nur in dem Rahmen, der durch die Semantik der Infomation gesteckt wird.

Beispiel: Der Agent kann die Routing-Tabelle nur so modifizieren, daß diese gültig bleibt, er kann keine "unsinnigen" Einträge in die Tabelle vornehmen.

Verfügbarkeit: Die Verfügbarkeit eines Systems kann berechtigten Entitäten nicht durch bösartige Handlungen verwehrt oder eingeschränkt werden.

Beispiel: Der Agent kann nur soviel Festplattenplatz belegen, daß andere Agenten und Prozesse des Endsystems weiter handlungsfähig bleiben.

Verantwortlichkeit/Zurechenbarkeit: Benutzer sind verantwortlich für ihre Handlungen. Jegliche Aktion muß der handelnden Entität zugeordnet werden können, auch nachdem die Aktion abgeschlossen ist. Diese Zuordnung muß nicht nur den an der Aktion unmittelbar beteiligten Entitäten möglich sein, sondern auch anderen Entitäten des Modells.

Beispiel: Der Administrator des Endsystems muß, nachdem der Agent die Routing-Tabelle verändert hat, später nachvollziehen können, welcher Agent die Veränderung vorgenommen hat und auf wessen Anforderung dieser Agent tätig geworden ist.

2.3 Mögliche Angriffstechniken

Techniken, mit denen Angriffe auf einen SmA durchgeführt werden können, sind beispielsweise in [OMG 98-03-09] und [KLM 97] verzeichnet. Die aufgeführten Punkte stehen dabei exemplarisch für eine Klasse von Angriffstechniken. In der Praxis wenden konkrete Anriffe häufig eine Kombination der beschriebenen Einzeltechniken an.

1. **Überbeanspruchung von Ressourcen**: Eine Ressource wird so stark beansprucht, daß ihre Verfügbarkeit für andere Entitäten eingeschränkt wird.

   Beispiel: Ein Agent belegt soviel Festplattenplatz, daß andere Agenten, das Agentensystem und andere Prozesse auf dem Endsystem nur noch eingeschränkt oder gar nicht mehr ausgeführt werden können.

   Ein spezieller Fall hiervon ist **Spamming**: Ein Service wird mit Anfragen (legal oder illegal) überschwemmt, so daß seine Verfügbarkeit eingeschränkt wird.

2. **Maskerade**: Eine Entität fälscht ihre Identität, um Zugriff auf Schnittstellen oder Daten zu erlangen.

   Beispiel: Ein Agent zum Sammeln von Netzstatistiken gibt vor, ein Agent zum Netz-Interface-Management zu sein, um unberechtigt ein Netz-Interface zu manipulieren.

3. **Trojanisches Pferd**: Eine Entität täuscht eine bestimmte, als zulässig akzeptierte Funktionalität vor, um tatsächlich aber illegale Funktionen ausführen zu können um damit an private Informationen zu gelangen.

   Beispiel: Ein Agent zum Sammeln von Netzstatistiken enthält neben dem hierfür notwendigen Programmcode zusätzlich noch weiteren Code, um illegal ein Netz-Interface zu manipulieren.

4. **Abhören**: Ein Kanal (vgl. Abschnitt 2.1), und damit die Kommunikation zwischen Entitäten wird belauscht, um geheime Informationen zu erlangen.

   Beispiel: Ein Angreifer belauscht einen Kanal, über den das Paßwort eines Benutzers übermittelt wird. Mit dem illegal erlangten Paßwort kann er sich später als dieser

Benutzer ausgeben.

5. **Umänderung/Man-in-the-Middle**: Eine über einen Kanal versandte Information wird abgefangen um diese, bevor sie an den Empfänger weitergeleitet wird, zu verändern oder durch eine andere Information ersetzt.

   Beispiel: Ein Angreifer fängt die Daten einer Geldüberweisung ab, die von einer Bankfiliale an die Zentrale versandt wird. Er ersetzt das ursprüngliche Empfängerkonto mit seinem eigenen und leitet die Daten weiter an die Zentrale, wodurch der Angreifer den Betrag gutgeschrieben bekommt.

6. **Wiedereinspielung**: Die Kommunikation über einen Kanal wird von einem Angreifer aufgezeichnet, um die (evtl. modifizierte) Aufzeichnung zu einem späteren Zeitpunkt erneut einspielen zu können.

   Beispiel: Ein Angreifer fertigt einen Mitschnitt einer Kommunikation an, die durchgeführt wird, um eine Überweisung auf sein Konto durchzuführen. Später spielt er den Mitschnitt wieder ein, wodurch die Überweisung ein zweites mal auf sein Konto durchgeführt wird.

7. **Leugnung/Abstreiten**: Eine Entität, die eine Aktion durchgeführt hat, bestreitet später die Durchführung dieser Aktion.

   Beispiel: Ein Angreifer führt eine Lastschrift auf ein fremdes Konto aus. Der Eigentümer des belasteten Kontos fordert den Betrag zurück, aber der Angreifer behauptet die Transaktion nicht durchgeführt und keine Gutschrift erhalten zu haben.

Die oben aufgezählten Techniken nutzen dabei im wesentlichen entweder Schnittstellen anderer Entitäten oder Kanäle als Angriffspunkte: 1, 2 und 3 greifen an Schnittstellen an; 4, 5 und 6 greifen an Kanälen an.

2.4 Spezielle Problematik von Systemen mobiler Agenten

Die in den vorangehenden Abschnitten dargestellte Problematik ist noch nicht spezifisch für SmA. Klassische verteilte Anwendungen, sind mit den gleichen Problemen konfrontiert. Im Fall von SmA werden aber einige Probleme durch die charakteristischen Eigenschaften von SmA noch verschärft bzw. ihre Komplexität drastisch erhöht.

Generelle Ursache für die höhere Komplexität der Sicherheitsproblematik von SmA im Vergleich zu klassischen verteilten Anwendungen ist der unterschiedliche konzeptionelle Ansatz. Verteilte Anwendungen werden zumeist spezifisch für konkrete Anwendungen spezifiziert und implementiert, SmA dagegen verstehen sich als universelle Plattform zur Implementierung konkreter Anwendungen.

Im Folgenden soll stellvertretend für klassische verteilte Anwendungen die Ausprägung als Client/Server Architektur (CSA) betrachtet werden. An einem Beispiel könnte das bedeuten, daß man für eine Banking-Anwendung unter Verwendung von CSA ein genau darauf zugeschnittenes System erstellt, während man anderenfalls spezielle Banking-Agenten auf einem generischen SmA erstellen würde.

2.4.1 Erweiterung der Funktionalität zu Laufzeit

Somit ist in CSA die Funktionalität aller beteiligter Komponenten von vornherein festgelegt (und damit eingeschränkt) und verändert sich auch während der Laufzeit des Gesamtsystems nicht. Dies bewirkt, daß die möglichen Bedrohungen, denen solch ein System ausgesetzt ist, in vergleichsweise engen Grenzen bereits zum Zeitpunkt der Implementierung kalkulierbar sind und ihnen bei Erstellung des Systems spezifisch und vor allem statisch Rechnung getragen werden kann.

Am Beispiel der Banking-Anwendung bedeutet dies vereinfacht, daß die Funktionalität des CSA nur Aktionen wie "Überweisen", "Kontoabfrage", "Dauerauftrag einrichten" umfaßt. Die Schnittstelle des Servers ist somit sehr schmal und leicht zu überwachen. Mögliche Bedrohungen eines solchen Systems wären "Illegale Überweisung/Kontoüberziehung", "Kontoabfrage durch andere Personen als den Kontoinhaber", etc.

In SmA dagegen ist nur die konkrete Funktionalität der Agentensysteme, nicht aber die von Agenten vorherbestimmt. Vielmehr wird die Wirkung von Agenten auf andere Entitäten bewußt offengehalten, um die Mächtigkeit und Flexibilität des SmA nicht einzuschränken. Im Fall von SmA für Managementanwendungen gilt dies insbesondere auch für die Wirkung auf die Endsysteme. Die Folge davon ist allerdings, daß der Umfang der konkreten Bedrohungen ebenso unvorhersehbar wie unüberschaubar ist. Vielmehr muß dynamisch, und von Fall zu Fall entschieden werden, ob eine Aktion nun legitim ist oder nicht.

Wiederum am Beispiel der Banking-Anwendung bedeutet dies, daß entschieden werden muß, ob der Agent berechtigt ist, z.B. bestimmte Einträge einer Datenbank auszulesen, diese zu verändern oder neue hinzuzufügen. Dabei bleibt dem Agentensystem die übergreifende Semantik der Einzelaktionen ("Überweisung") jedoch verborgen. Gefahren, die dabei durch die direkte Schnittstelle zum Datenbanksystem ausgehen, sind deutlich vielfältiger als im Fall der CSA. So können neben Bedrohungen wie sie für die CSA bestehen, jetzt auch Angriffe erfolgen, die z.B. die Datenbank in einen inkonsistenten Zustand bringen.

2.4.2 Mobilität der Agenten

Eine zusätzliche Erschwernis stellt auch die Mobilität der Agenten dar. Stellt in einer CSA ein Client beispielsweise eine Anfrage zur Ausführung einer Aktion auf dem Server, genügt es sich davon zu überzeugen, daß der direkte Kommunikationspartner (der Client) vertrauenswürdig ist, da Daten nur auf ihm oder dem Server selbst bearbeitet wurden und werden.

So genügt es in einer CSA zur Kontenabfrage, daß der Server sich davon überzeugt, daß der anfragende Client von einem bestimmten Typ ist, welcher dann bereits die Überprüfung des Bankkunden vorgenommen hat, ob dieser zur Abfrage der fraglichen Konten berechtigt ist.

Fordert in einem SmA ein Agent die Ausführung einer Aktion an, muß dagegen prinzipiell auch die "Vorgeschichte" des Agenten betrachtet werden. Selbst wenn der Agent als vertrauenswürdig eingestuft wird, könnten nämlich Daten, die für die Ausführung der Aktion benutzt werden, vorher auf anderen Agentensystem unberechtigt (in feindlicher Absicht) manipuliert worden sein.

Wird ein mobiler Agent zu Kontenabfrage "ausgesandt", muß auf jedem Agentensystem erneut übeprüft werden, ob dieser Agent zur Abfrage berechtigt ist, und ob die Identität des Bankkunden auf seiner "Reise" unverändert geblieben ist.

2.4.3 Organisationsstruktur und Verantwortlichkeit

In einer CSA sind die Eigentumsverhältnisse der Komponenten typischerweise identisch zu denen der Endsysteme. Ein Server, der auf einem Endsystem ausgeführt wird, gehört der gleichen Organisation wie das Endsystem selbst, und wird ausschließlich für diese Organisation tätig. Äquivalent gehört der Client jener Organisation/Person, der das Endsystem gehört auf dem dieser ausgeführt wird. Somit ist die Organisationsstruktur in einer CSA äquivalent zu Organisationsstruktur der Endsysteme, auf denen die Komponenten ausgeführt werden.

Für die Zurechenbarkeit von Aktionen bedeutet das, daß eine Aktion des Servers unmittelbar im Verantwortungsbereich des Server-Eigentümers liegt, da der Server die Aktion tatsächlich ausführt, und nur mittelbar dem Eigentümer des Clients zuzuschreiben ist, da dieser die Aktion nur ausgelöst hat.

In SmA dagegen diversifizieren sich die Eigentumsverhältnisse aus der Sicht der Endsysteme. Während die Agentensysteme den Eigentümern der Endsysteme gehören, ist der Eigentümer eines Agenten sein Auftraggeber. Somit befinden sich auf einem Endsystem aus dessen Sicht in der Regel Entitäten mit unterschiedlichen Eigentümern. Daher ist die Organisationsstruktur der Entitäten in einem SmA keineswegs identisch zu jener der beiteiligten Endsysteme.

Für die Verantwortlichkeit einer Aktion heißt das, daß die Aktionen eines Agenten unmittelbar dem Eigentümer des Agenten zuzuschreiben sind und nicht der Verantwortung des Eigentümers des Agentensystems unterliegen. Daraus folgt, daß für SmA ein eigenständiges Domänenkonzept gefunden werden muß, das die Organisationsstruktur im SmA abbildet.

2.4.4 Kooperation von Agenten und Delegation von Rechten

Eine CSA zeichnet sich vorallem durch die fast ausschließlich paarweise Beziehungen zwischen Client und Server aus. Vertrauensbeziehungen bestehen dabei immer nur wechselseitig zwischen Client und Server.

Dagegen sind Anwendungen auf SmA häufig so gestaltet, daß zur Erfüllung einer Aufgabe mehrere verschieden Agenten, jeder spezialisiert auf eine bestimmte Teilaufgabe, kooperieren und dadurch die Gesamtaufgabe gelöst wird. Hierbei sind aber Vertrauensbeziehungen im gesamten Netz der kooperierenden Agenten notwendig. Häufig müssen dabei auch Berechtigungen, die ein Agent A besitzt an einen anderen Agent B weitergegeben werden, damit B eine Teilaufgabe für A übernehmen kann. Dieses Verfahren wird als Delegation von Rechten bezeichnet.

So muß beispielsweise ein Kontenauszugs-Agent, der für einen bestimmten Kunden tätig ist und sich dafür der Hilfe eines Datenbankzugriffs-Agenten bedient, diesem die Berechtigungen "mitgeben", die für diesen Kunden relevanten Teile aus der Datenbank zu lesen.

2.5 Zusammenfassung der Anforderungen

Aus den in Abschnitt 2.2 aufgezählten Bedrohungen ergibt sich unmittelbar, daß

- ein wechselseitiger Schutz aller Entitäten des Modells voreinander, und
- ein Schutz gegen außenstehende Dritte

notwendig ist. Aus diesen Forderungen legitimiert sich das Thema dieser Arbeit.

Um zwischen erlaubten und unbefugten Aktionen unterscheiden zu können ist zunächst eine sichere Identifizierung der beteiligten Entitäten erforderlich: *Authentisierung.*

Anschließend muß für jede Aktion eine Entscheidung getroffen werden, ob die beteiligten Entitäten berechtigt bzw. willens sind, die Aktion durchzuführen: *Autorisierung*

Dabei sind nach Abschnitt 2.3 die zu sichernden Teile vor allem die Schnittstellen und die Kanäle.

Um den speziellen Bedürfnissen von SmA Rechnung zu tragen, sind weitere Anforderungen zu stellen:

- Integrität: Bedingt durch die Mobilität der Agenten sind Daten, die sie mit sich tragen, der Gefahr der Veränderung ausgesetzt. Solche Veränderungen sollten durch einen Integritätsnachweis erkennbar sein.
- Dynamik: Den sich zur Laufzeit eines SmA ändernden Strukturen sollte ein zur Laufzeit dynamisch skalierender Schutz gegenüberstehen.
- Feingranularität: Eine Sicherheitsarchitektur für SmA sollte trotz des generischen Ansatzes der SmA eine Möglichekeit bieten, Sicherheitsaspekte feingranular und auf die Semantik der Agenten angepaßt zu formulieren.
- Domänen: Um den unterschiedlichen Eigentumsverhältnissen und Verantwortlichkeiten Rechnung zu tragen muß ein eigenes Domänenkonzept realisiert werden.
- Koorperation und Delegation muß unterstützt werden.

Konträr zu all diesen Anforderungen steht der Wunsch nach möglichst einfacher Bedienung und Wartung eines SmA.

Die konkrete Anforderung dieser Arbeit besteht nun darin, eine Lösung für ein Sicherheitsmodell für MASA in diesem Spannungsfeld zu finden. Hierfür wird MASA im nächsten Kapitel zunächst auf die konkret bestehenden Risiken untersucht.

Kapitel 3

Risikoanalyse für MASA

In diesem Kapitel wird die Ausgangsversion des MASA-Systems [1] unter dem Aspekt der Sicherheit betrachtet. Alle Betrachtungen in dieser Arbeit beziehen sich auf genau diese Version. Im Folgenden soll sie nur kurz MASA oder MASA-Implementierung genannt werden.

3.1 Modell von MASA

Für die Risikoanalyse von MASA sollen zunächst die hierfür relevanten Komponenten vorgestellt und ihr Zusammenspiel kurz erläutert werden.

Erweitert man das Modell aus Kap. 2.1 um MASA-spezifische Eigenschaften, zerfällt zunächst der Begriff des Agenten in zwei Teile:

- Agentengattung
- Agenteninstanz

Diese Unterscheidung spiegelt die Trennung zwischen den statischen (Agentengattung) und dynamischen Teilen (Agenteninstanz) eines Agenten wider. Eine solche Unterscheidung wird in [OMG 98-03-09] nicht und in [Kemp 98] nicht explizit getroffen. Vereinfacht gesagt kann diese Unterscheidung als ein Äquivalent zu jener von Klassen und Objekten in der Welt der objektorientierten Programmierung gesehen werden:

Unter Agentengattung versteht man alle statischen Teile von Agenten, die allen Agenteninstanzen einer Gattung gemein sind. Dies ist in erster Linie der Programmcode, der für alle ablaufenden Agenteninstanzen einer Agentengattung identisch ist.

Agenteninstanz meint dagegen die konkrete Ausprägung eines ablaufenden Agenten, darunter die aktuelle Belegung seiner Attribute, seiner Historie, etc.

Am Beispiel eines Agenten namens FOO erklärt bedeutet dies, daß zum Start eines Agenten zuerst die Agentengattung gewählt wird: FOO. Unter Angabe von Startparametern wird eine neue Agenteninstanz erzeugt, die dann auf dem Agentensystem ausgeführt wird. Wird mit anderen Parametern eine zweite FOO Instanz gestartet, so gehören beide Agenten einer gemeinsamen Gattung an, stellen aber unterschiedliche Agenteninstanzen dar.

[1] Konkret ist dies die Version des CVS-Repositories vom 19. Februar 1999.

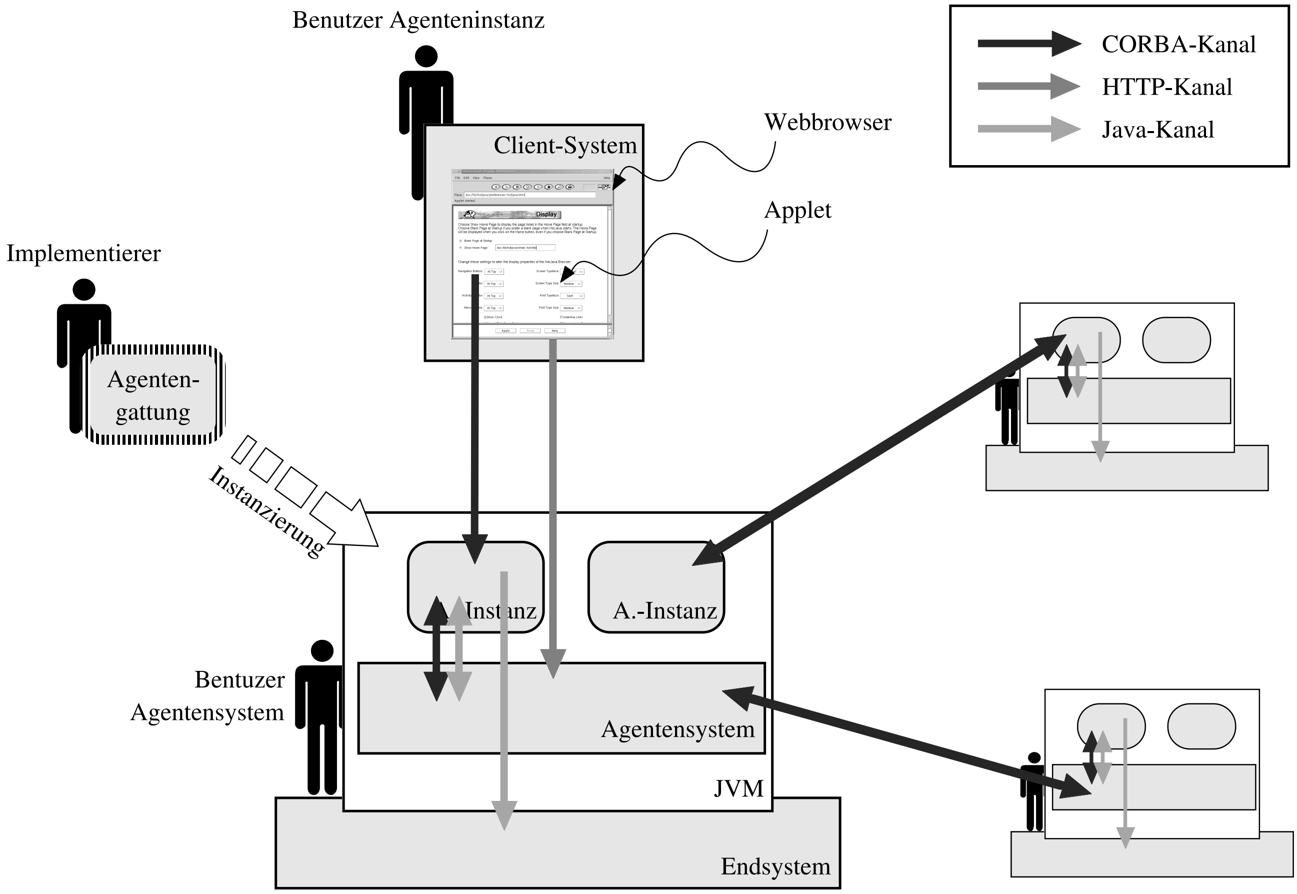

Abbildung 3.1: Skizze des MASA-Modells

Zur Verdeutlichung der Unterscheidung kann auch der Java-eigene Serialisierungsmechanismus von Objekten herangezogen werden. Wird in Java ein durch das Interface `java.io.Serializable` gekennzeichnetes Objekt (beispielsweise auf einen Strom) serialisiert, so enthalten diese Daten nur die aktuelle Belegung der Attribute. Ein Empfänger solcher Daten muß, um das serialisierte Objekt wieder erzeugen zu können, zusätzlich zu den Daten, über den zum Objekt gehörigen Code verfügen.

Die Entitäten im verfeinerten Modell, skizziert in Abb. 3.1, sind nun:

- Agentengattungen
- Applets
- Agenteninstanzen
- Agentensysteme – die Laufzeitumgebungen für Agenteninstanzen
- Endsysteme – auf diesen laufen die Agentensysteme ab – meist Hosts mit eigenem Betriebssystem
- Client-Systeme mit Webbrowser
- Benutzer von Agentensystemen
- Benutzer von Agenteninstanzen

- Implementierer von Agentengattungen

Dabei wird die Kommunikation zwischen allen Komponenten mittels einem der folgenden Kanäle abgewickelt:

- CORBA/IIOP
- HTTP
- Java[2]

Die Unterscheidung der verschiedenen Arten von Benutzern wird unmittelbar durch die verschiedenen Rollen vorgegeben, die sie im MASA einnehmen.

Die Trennung zwischen Agenteninstanz und Agentengattung ruft dabei die Unterscheidung von Implementierern und Benutzern (im Sinne von Anwendern) hervor, konkret: Der Implementierer der Agentengattung ist in der Regel nicht identisch mit dem späteren Anwender des Agenten. Da beide Rollen unterschiedliche Teile des Lebenszyklusses eines Agenten berühren, sind diese für Sicherheitsbetrachtungen zu unterscheiden.

Benutzer von Agenteninstanzen und Agentensystemen sind wegen ihrer unterschiedlichen Aufgabenbereiche, die sie abdecken, zu unterscheiden. Während der Benutzer einer Agenteninstanz nur für diese zuständig ist, übernimmt der Benutzer eines Agentensystems weiterreichende administrative Aufgaben, die entsprechend auch die Agenteninstanzen auf einem Agentensystem berühren können. Somit sind diese beiden Benutzertypen aus der Sicherheitsperspektive zu unterscheiden.

Betrachtet man die Entitäten des Modells, gliedern sich diese in solche, die selbst Aktionen (mittelbar oder unmittelbar) durchführen (Subjekte) und jene auf die eine Aktion wirkt (Objekte):

- Subjekte: Applets, Agenteninstanzen, Agentensysteme, Client-Systeme und Benutzer.
- Objekte: Agentengattungen, Agenteninstanzen, Agentensysteme, Endsysteme

Auffallend ist, daß Agenteninstanzen und Agentensysteme sowohl Subjekte, als auch Objekte darstellen.

Anforderungen zur Ausführung von Aktionen werden von Subjekten über die unterschiedlichen Kanäle übertragen, dabei bieten Agenteninstanzen, Agentensysteme und Endsysteme Schnittstellen an, wo die Aktionen dann ausgeführt werden:

- Benutzer bedienen Applets, die über den HTTP-Kanal auf den Webbrowser geladen wurden und dort ausgeführt werden.
- Über CORBA/IIOP leiten Applets die Anforderungen an die Agenteninstanzen weiter.
- Agenteninstanzen wiederum kommunizieren über den Java-Kanal mit ihrem Agentensystem und führen in Form von Java-API Aufrufen Aktionen auf dem Endsystem aus. Zur Kommunikation mit dem Agentensystem wird in MASA aktuell auch der CORBA-Kanal genutzt. Weiterhin können, ebenfalls über den CORBA-Kanal, andere Agenteninstanzen zur Durchführung von Aktionen veranlaßt werden.
- Agentensysteme verständigen sich ebenfalls über CORBA/IIOP.

[2] Methodenaufrufe über Java werden hier abstrakt als Kanal betrachtet.

In den nachfolgenden Abschnitten wird das Modell nun nacheinander aus der Sicht der Entitäten, Kanäle und Schnittstellen betrachtet, um Möglichkeiten zur Authentisierung der Entitäten, sowie die Gefahr von Angriffen, ansetzend an Kanälen bzw. Schnittstellen, zu beleuchten.

3.2 Entitäten-Sicht

Für die in Kap. 2.5 geforderte Authentisierung ist es zunächst erforderlich, daß die einzelnen Entitäten identifiziert werden können. Die nachfolgende Tabelle 3.1 gibt eine Übersicht darüber, ob und wie dies in der MASA-Implementierung möglich ist.

	Möglichkeit zur	
Entität	Identifizierung	Authentisierung
Agentengattungen	Java-Package & Name der Hauptklasse	-
Applets	Java-Package & Name der Hauptklasse	-
Agenteninstanzen	Datenstruktur `CfMAF.Name`	-
Agentensysteme	Datenstruktur `CfMAF.Name`	-
Endsysteme	indirekt über Agentensystem	-
Client-Systeme	IP-Adresse	-
Benutzer von A.-Systemen	-	-
Benutzer von A.-Instanzen	-	-
Implementierer von A.-Gattungen	-	-

Tabelle 3.1: Übersicht über Möglichkeiten zu Identifizierung und Authentisierung in MASA

3.2.1 Identifizierung

Agentengattungen und Applets

Agentengattungen können über den hierarchisch aufgebauten Namensraum der Java-Klassen eindeutig identifiziert werden. Nach Konvention wird die Basis des Package-Namens einer Java-Klasse durch den rückwärts geschriebenen DNS-Namen der erstellenden Organisation gebildet, woran sich die weitere organisatorische und implementierungstechnische Gliederung anschließt. Eine weitere und MASA-spezifische Konvention besagt dann noch, daß der Name der Hauptklasse eines Agenten aus der letzten Hierarchie-Ebene des Package-Namens und dem Agententypen (`MobileAgent` oder `StationaryAgent`) gebildet wird.

Beispielsweise trägt die Hauptklasse der Gattung des FOO-Agenten den Namen `de.unimuenchen.informatik.mnm.FOO.FOOMobileAgent`, gebildet aus dem DNS-Namen `informatik.uni-muenchen.de`, der Bezeichnung der Suborganisation "MNM", der Gattung "FOO" und dem Typen "mobiler Agent".

Äquivalent zu den Agentengattungen können Applets am Package-Namen ihrer Hauptklasse identifiziert werden.

Da davon ausgegangen werden kann, daß jede Organisation, die Agenten implementiert, im DNS-Namensraum vertreten ist und vom DNS-Namen eindeutig auf die Organisation geschlossen werden kann, ist der geschilderte Mechanismus zur Bildung von Gattungsnamen als ausreichend für deren Identifizierung anzusehen.

Agenteninstanzen und Agentensysteme

Die für Agenteninstanzen und Agentensysteme in [OMG 98-03-09], Kap. 1.2 und Kap. 3.5 definierten und nach [Kemp 98], Kap. 4.5.1 in MASA verwendeten Datenstrukturen erfüllen voll die für eine eindeutige Identifizierung notwendigen Eigenschaften. Dabei bezieht sich die Eindeutigkeit nicht nur lokal auf ein Agentensystem, sie ist prinzipiell vielmehr auch global über die Grenzen der Agentensysteme hinweg im gesamten SmA gegeben.

Endsysteme

Endsysteme können in MASA nach der in [Kemp 98], Kap. 4.5.1 definierten Konvention über den Identifikator des Agentensystems eindeutig identifiziert werden. Diese besagt, daß in der Datenstruktur `CfMAF.Name` das Strukturelement `Identity` auf den Host-Namen des Endsystems zu setzen ist. Konkret wird dabei der DNS-Name als Host-Name, und damit als Identifikator des Endsystems, verwendet, womit ein Endsystem im DNS-Namensraum eindeutig identifiziert werden kann (siehe `AgentSystem.createAgentSystemName(...)`).

Client-Systeme

Client-Systeme können in MASA z. Zt. nur im Namensraum der IP-Adressen identifiziert werden. Genau betrachtet ist es nur dem Webserver-Agenten möglich eine solche Identifizierung vorzunehmen, da dieser die HTTP-Verbindung zum Client-System verwirklicht. Das Agentensystem selbst hat keine Kenntnis über die Identität des Client-Systems.

Benutzer

Eine Identifizierung von Benutzern jeglichen Typs ist in MASA nicht implementiert und auch nicht vorgesehen. Tatsächlich werden Benutzer weder in [OMG 98-03-09] noch in [Kemp 98] betrachtet.

3.2.2 Schwächen der bisherigen Implementierung

Zwar wären einige Entitäten des Modells nach dem vorangegangenen Abschnitt eindeutig identifizierbar, allerdings werden die vorgestellten Identifikatoren in der vorliegenden MASA-Version noch nicht konsequent und korrekt an allen Stellen eingesetzt.

So wird durchgängig für den Identifikator eines Agenten nur das `Identity` Strukturelement betrachtet. Entsprechend wird in der Hilfsklasse `NameWrapper` für die Bildung einer String-Darstellung von `CfMAF.Name` in der Methode `toString()` ausschließlich die `Identity` Komponente benutzt. Folglich ist eine eindeutige Rückabbildung aus der String-Darstellung zum `CfMAF.Name` nicht möglich.

Korrekt wäre, daß das gesamte 3-Tupel von `CfMAF.Name` als Identifikator betrachtet wird und diese für die Abbildung auf eine String-Darstellung herangezogen wird.

Daneben finden sich an verschiedenen Stellen auch noch individuelle Abbildungen zur String-Darstellung, z.B. in der Klasse `AgentSystem` wird das Attribut `agent_system_name` deklariert.

Es wird dann in der `main()`-Methode mit einer String-Darstellung des `CfMAF.Name` des Agentensystems belegt, ohne die Methode `toString()` zu benutzen.

Als weiterer Kritikpunkt an MASA ist die nahezu völlig fehlende Unterscheidung zwischen Agentengattungen und Agenteninstanzen zu nennen. Diese konzeptionelle Schwäche hat weitreichende Folgen, die sich über die gesamte Implementierung erstrecken.

Am deutlichsten zeigt sich das Fehlen dieser Unterscheidung in der Tatsache, daß es nicht möglich ist, auf einem Agentensystem von einer Agentengattung mehrere Instanzen zu starten. Aber auch die URL-Adressierung von Applets einer Agenteninstanz in der Form

`http://`<*agentensystem*>`:`<*port*>`/`<*gattungsname*>

zeigt die Folgen. Um mehrere Agenteninstanzen der gleichen Gattung unterscheiden zu können müßten die URL beispielsweise lauten:

`http://`<*agentensystem*>`:`<*port*>`/`<*cfmafName_string*>

3.2.3 Authentisierung

Wie aus Tabelle 3.1 ersichtlich, wird eine Authentisierung momentan für keine der Entitäten unterstützt, noch ist diese vorgesehen.

3.3 Kanal-Sicht

Da durch mögliche Angriffe, die sich gegen Kanäle richten, eine Bedrohung aller Entitäten gegeben ist, werden in diesem Abschnitt die Kanal-Typen des MASA-Modells betrachtet. Tabelle 3.2 gibt eine Übersicht, welche Kanal-Typen zwischen den verschiedenen Entitäten benutzt werden. Dabei wird zwischen den Entitäten unterschieden, die eine Kommunikation initieren, und jenen, die als Partner vom Initiator angesprochen werden.

Kommunikations-initiator	Kommunikationspartner: Endsystem	Agenten-system	Agenten-instanz	Client-System+ Webbrowser	Applet
Endsystem	-	-	-	-	-
Agentensystem	Java	CORBA	CORBA Java	- -	- -
Agenteninstanz	Java	CORBA Java	CORBA	- -	- -
Client-System+ Webbrowser	-	HTTP	-	-	-
Applet	-	-	CORBA	-	-

Tabelle 3.2: Benutzte Kanäle zwischen den Entitäten

3.3.1 CORBA-Kanal

Mit der in der bisherigen MASA-Implementierung benutzten CORBA-Laufzeitumgebung Visibroker for Java Version 3.0 der Firma Inprise wird ein *Object Request Broker* (ORB) verwendet, der die CORBA-Kommunikation über TCP/IP-Verbindungen implementiert. Dabei werden alle CORBA-Daten unverschlüsselt über einen TCP-Strom versandt. Damit sind alle Angriffe durchführbar, die in Kap. 2.3 für Kanäle aufgezählt wurden. Technisch gesehen führt jeder Angriff, der auf Ebene des ISO/OSI-Modells Schicht 4 und darunter denkbar ist, zu einer Gefährdung.

Als Beispiel hierzu seien zwei Agenten betrachtet, dabei übermittle der eine Agent dem anderen in einem CORBA-Aufruf, für eine SNMP-Operation, einen *SNMPv1 Community String*. Nun kann ein Angreifer durch Abhören des TCP/IP-Datenstroms diesen *Community String* erhalten. Damit erlangt er (aus sicherheitstechnischer Sicht) vollen Zugriff auf die durch SNMP gemanagte Komponente (siehe auch [HAN 99a], Kap. 6.3.1).

Somit ist der CORBA-Kanal als völlig ungesichert zu bezeichnen, eine Sicherung des Kanals ist weder im verwendeten ORB noch in MASA selbst vorgesehen.

3.3.2 HTTP-Kanal

Für den HTTP-Kanal zeigt sich das äquivalente Bild zum CORBA-Kanal. Da wiederum Daten unverschlüsselt über einen TCP-Strom versandt werden, bleibt auch hier das gleich Fazit: Der Kanal ist völlig ungesichert.

Ebenso ist hier keine Sicherung in MASA, speziell in dem für die HTTP-Kommunikation verantwortlichen Agenten WebserverAgent, vorgesehen.

Eine Gefährdung der Sicherheit von MASA resultiert an diesem Punkt z.B. durch eine Umänderungsattacke. Ein Angreifer könnte ein, über den HTTP-Kanal übertragenes, Applet abfangen und eine modifizierte Fassung wiedereinspielen, die dann z.B. zusätzliche Schnittstellen für weitere Angriffe bereitstellt.

3.3.3 Java-Kanal

Da Methodenaufrufe über den Java-Kanal nur innerhalb einer JVM auftreten, ist der Java-Kanal durch den Speicherschutz der JVM (vgl. Kap. 6.1.1 und [JLS]) geschützt. Im Gegensatz zu anderen gängigen Programmiersprachen (z.B. C++) sind Abhör-, oder Umänderungsattacken durch direkte Speicherinspektion oder -manipulationen ausgeschlossen.

3.4 Schnittstellen-Sicht

In diesem Abschnitt sollen nun die Schnittstellen jener Entitäten untersucht werden, an denen Aktionen von anderen Subjekten des Modells ausgeführt werden können. Diese entsprechen den "Kommunikationspartnern" aus Tabelle. 3.2. Hierzu soll u.a. dargestellt werden, wie die Schnittstellen definiert sind und ob ihre Benutzung autorisiert wird bzw. dies vorgesehen ist.

Die Begriffe CORBA-Schnittstelle bzw. Java-Schnittstelle meinen dabei Funktionalität, die über den CORBA- bzw. Java-Kanal nutzbar sind.

3.4.1 Agenteninstanz

Die an einer Agenteninstanz von anderen Entitäten nutzbaren Schnittstellen werden ausschließlich durch die CORBA-Schnittstelle der Agentengattung definiert. Diese wird in IDL beschrieben.

In der Implementierung sind diese CORBA-Schnittstellen völlig ungesichert, was bedeutet, daß jedes beliebige lokale oder entfernte CORBA-Objekt (d.h. damit auch Agenteninstanzen, Applets, Client-Programme und indirekt Benutzer) diese ungehindert benutzen können. Eine Autorisierung findet weder statt, noch ist sie vorgesehen (weder in MASA selbst, noch im verwendeten ORB).

Zusätzlich zu den CORBA-Schnittstellen existieren jedoch auch noch Methoden, die über den Java-Kanal erreichbar sind. Diese werden in den Klassen `Agent`, `StationaryAgent` und `MobileAgent` definiert und werden vom Agentensystem für Verwaltungsaufgaben (z.B. vor und nach einer Migration) an der Agenteninstanz benötigt:

- `init(...)`: Initialisiert interne Attribute des Agentensystems bei Erstellung einer neuen Agenteninstanz.
- `initTransient(...)`: Initialisiert interne Attribute des Agentensystems nach Migration.
- `setThread(...)`: Setzt internes Attribut, das auf den Haupt-Thread der Agenteninstanz verweist.

Dabei sind sowohl die genannten Basisklassen für Agenten `public` deklariert, als auch diese Methoden selbst. Die Verwendung der Methoden wird dabei nicht autorisiert. Eine Autorisierung ist auch nicht vorgesehen. Weiterhin sind teilweise aber auch Attribute der jeweiligen Klassen als `public` deklariert.

Beispielsweise durch die Methode `initTransient(...)` entsteht dabei für eine Agenteninstanz die Möglichkeit interne Attribute des Agentensystems zu manipulieren, womit das Agentensystem massiv beeinträchtigt werden kann.

Durch die Art der Deklaration und die fehlende Autorisierung sind neben den CORBA-Schnittstellen nun auch jegliche Java-Schnittstellen ungeschützt, und können, zumindest von Komponenten auf dem gleichen Agentensystem (bzw. der gleichen JVM) ungehindert genutzt werden. Die Problematik des offenen Java-Kanals gilt dabei auch gleichzeitig für die Methoden, die die CORBA-Schnittstelle implementieren. Diese müssen nach dem IDL-to-Java Mapping ([OMG 98-04-03]) als `public` deklariert werden. Damit können aber, wiederum innerhalb des gleichen Agentensystems, Methoden der CORBA-Schnittstelle (selbst wenn diese gesichert wäre) über den Java-Kanal ungehindert erreicht werden.

Im Falle der Attribute der Klasse `Agent`, besteht aber noch ein weiteres Risiko. Selbst wenn die Attribute durch die Deklaration `protected` nicht von fremden Klassen benutzt werden können, sind diese vor Manipulation durch Tochterklassen nicht geschützt. Das bedeutet, daß jede Agenteninstanz, da sie immer durch eine Tochterklasse von `Agent` implementiert ist, diese Attribute frei verändern kann. Kritisch ist dies bei solchen Attributen, die der Agenteninstanz zwar lesend zur Verfügung stehen sollen, aber nicht verändert werden dürfen. MASA verwaltet einige solcher Attribute direkt in der Klasse `Agent`. Nachfolgend zwei Beispiele für eine mögliche Gefährdung durch solche ungesicherten Attribute.

Das Attribut `_name`, durch das die Identität der Agenteninstanz festgelegt wird, darf nur vom startenden Agentensystem einmalig gesetzt werden. In der bisherigen Implementierung kann eine Instanz dieses Attribut jedoch beliebig manipulieren, und damit ihre Identität verändern.

Ein weiteres Beispiel für einen konkreten Angriff liefert das Attribut `_code_base`. Durch diesen Wert wird bestimmt, wo der ausführbare Code der Agentengattung zu finden ist. Manipuliert eine Agenteninstanz diesen Wert, so kann sie ein "Trojanisches Pferd" implementieren, in dem sie einen neuen Ort des zu ladenden Codes angibt. Später wird von Agentensystemen dann zur Ausführung der Instanz der Code vom neuen Ort benutzt. Implementiert der dort zu findende Code dann eine neue Funktionalität, so ist der Angriff gelungen.

Neben den vor der Laufzeit bekannten Schnittstellen kann eine Agenteninstanz beliebig neue Schnittstellen zur Laufzeit nach außen einrichten. Da die Erstellung von IP-Sockets nicht überwacht wird, ist es einer Agenteninstanz möglich über einen IP-Socket eine neue Schnittstelle zu schaffen, um damit beispielsweise unbemerkt Daten zu versenden und zu empfangen.

3.4.2 Agentensystem

Wie im Fall von Agenten werden die über den CORBA-Kanal legal erreichbaren Schnittstellen durch eine IDL-Beschreibung dargestellt. Diese sind damit zwar exakt definiert, in der Implementierung aber dennoch ungesichert. Wiederum findet eine Autorisierung nicht statt, ebenfalls ist diese auch nicht vorgesehen.

Auch im Fall der Java-Schnittstelle zeigt sich ein zu den Agenten identisches Bild. Besonders kritisch sind dabei interne Java-Schnittstellen des Agentensystems. In der Implementierung ist die Funktionalität des Agentensystem auf zwei Hauptklassen aufgeteilt: `AgentSystem` und `AgentManager`. Zur internen Kommunikation der beiden Klassen wird der Java-Kanal benutzt. Die zugehörigen Schnittstellen sind aber ebenfalls von allen Java-Objekten des gleichen Agentensystems wegen ihrer `public` Deklaration erreichbar. Exemplarisch sei hier die Methode `registerAgentManager(...)` erwähnt. Sie dient zur Anmeldung weiterer `AgentManager`-Objekte, die benötigt werden, um Agenten anderer Agentensysteme auf MASA ausführen zu können (vgl. [Bran 99]).

Zur Bewertung kann man anmerken, daß die Problematik der Java-Schnittstelle von Agentensystemen zwar strukturell identisch zu jener der Agenten ist, allerdings sind die Auswirkungen bezüglich der Sicherheit noch kritischer. Während durch ungesicherte Agenten sich diese zwar gegenseitig beeinflussen können, wird in diesem Fall aber die Sicherheit des gesamten Agentensystems beeinträchtigt.

3.4.3 Endsystem

Die Schnittstellen des Endsystems, dargestellt durch die Methoden der Java-API-Klassen , welche ausschließlich durch den Java-Kanal erreichbar sind, sind ungeschützt. Jegliches Java-Objekt (und damit jeder ausgeführte Agent) kann diese Schnittstellen unkontrolliert nutzen. Die durch die Java-Architektur in der Version 1.1.x vorgesehene Kontrollinstanz, der *Security Manager*, ist zwar in Form der Klasse `AgentSecurityManager` vorhanden, wird aber

1. im System nicht wirksam eingesetzt: Ausschließlich in der Methode `AgentManager.create_agent(...)` wird versucht bei der Instanziierung eines neuen Agenten, jeweils einen eigenen `AgentSecurityManager` zu installieren. Nach

[JDK1.1-SDK] kann dieser jedoch nur einmalig pro JVM installiert werden. Außerdem ist deshalb bis zum Zeitpunkt der Instanziierung des ersten Agenten der *SecurityManager* auch nicht aktiviert, und

2. übt faktisch keine Kontrolle aus, da alle Aktionen unter allen Umständen erlaubt werden.

Somit kann jeder Agent alle durch das Java-API angebotenen Aktionen auf dem Endsystem ausführen. Beispielsweise können beliebige Operationen auf Dateien (lesen/schreiben/löschen) vorgenommen werden oder neue Prozesse auf dem Endsystem gestartet werden.

Da für Managementanwendungen häufig privilegierte Rechte im Endsystem notwendig sind, werden Agentensysteme in vielen Fällen unter einer privilegierten Benutzerkennung des Endsystems ausgeführt. Gepaart mit dem ungeschützten Java-API öffnet sich für Agenten damit die “Büchse der Pandora” an Angriffsmöglichkeiten denen das Endsystem schutzlos ausgeliefert ist.

3.5 Zusätzliche Dienste

Neben dem Agentensystem selbst, sind noch zwei CORBA-Dienste für den Betrieb von MASA notwendig. Deren Sicherheitseigenschaften sollten deshalb ebenfalls betrachtet werden.

Beiden Diensten ist gemein, daß jeder in einer eigenen JVM ausgeführt wird. Sicherheitsrelevante Probleme, die aus einem Angriff über den Java-Kanal erfolgen könnten, sind somit ausgeschlossen.

3.5.1 Naming Service

Der *CORBA Naming Service* wird in MASA durch die CORBA-Laufzeitumgebung Visibroker for Java Version 3.0 bereitgestellt. Ebenso wie für den ORB selbst sind darin für den *Naming Service* keinerlei Sicherungsmaßnahmen vorgesehen.

Da durch den *Naming Service* aber der globale Verzeichnisdienst des SmA realisiert ist ([Kemp 98], Kap. 4.5.1) stellt dies eine empfindliche Sicherheitslücke für das gesamte SmA dar. So kann ein beliebiges CORBA-Objekt (und damit jeder Agent) neue Eintragungen im *Naming Service* vornehmen, bestehende Einträge verändern oder diese entfernen.

Mittels der Veränderung eines Eintrags im *Naming Service* läßt sich beispielsweise sehr einfach ein Spoofing-Angriff realisieren, indem ein Agent X die Objektreferenz eines anderen Agenten A durch die eigene ersetzt. Erfragt nun eine beliebige Entität im SmA die Objektreferenz von A und kommuniziert dann mit dem hierdurch referenziertem Objekt, im Glauben es handle sich um A. Tatsächlich ist die Entität aber mit X verbunden, womit der Angriff gelungen ist.

Durch Entfernen eines Eintrags aus dem *Naming Service* können sehr einfach DoS-Attacken durchgeführt werden, indem Agentensysteme und Agenten nach Entfernung ihres Eintrags für andere Entitäten als nicht existent erscheinen, obwohl diese weiterhin ausgeführt werden.

Dem *Naming Service* selbst ist durch MASA noch ein sehr einfacher HTTP-Server für die Vereinfachung des Bootstrapping-Prozesses zur Seite gestellt, mit dessen Hilfe Agentensysteme initial die Objektreferenz des *Naming Service* erhalten. Für ihn sind keine Sicherungsmaßnahmen implementiert oder vorgesehen. Alle in Abschnitt 3.3.2 gemachten Aussagen zum HTTP-Kanal gelten hier somit äquivalent.

3.5.2 Event Channel Service

Ebenfalls mittels einer Implementierung durch Visibroker for Java Version 3.0 wird ein *CORBA Event Channel Service* bereitgestellt, der ungeschützt implementiert ist.

Der *Event Channel Service* wird in MASA beispielsweise genutzt, um Broadcast-Nachrichten über den Start oder die Terminierung von Agenten und Agentensystemen an andere Entitäten zu versenden.

Vor diesem Hintergrund läßt sich der ungeschützte *Event Channel Service* für DoS-Attacken nutzen. Man könnte beispielsweise einen Agenten glaubend machen, daß ein anderer Agent, mit dem dieser kooperiert, terminiert wurde, indem eine gefälschte Nachricht über die Terminierungen versandt wird.

3.6 Ansätze für Sicherungsmaßnahmen

Nachdem durch die Schnittstellen-Sicht klar geworden ist, daß Sicherungsmaßnahmen für Endsystem, Agentensystem und Agenten erforderlich sind, sollen, für eine erste Einschätzung, die Vor- und Nachteile aufgezeigt werden, wenn die jeweilige Entität eigenverantwortlich für die Überprüfung und Durchsetzung von Sicherungsmaßnahmen zuständig ist.

3.6.1 Sicherung durch das Endsystem

Jegliche Sicherungsmaßnahmen werden durch das Endsystem bzw. das darauf ablaufende Betriebssystem vorgenommen. Dabei erscheint das gesamte Agentensystem mit den darauf ablaufenden Agenten als "Black Box".

Ein Beispiel für solches Vorgehen wäre die Nutzung von Zugriffsrechten des Betriebssystems auf Dateien, um das (lokale) Dateisystem des Endsystems zu sichern.

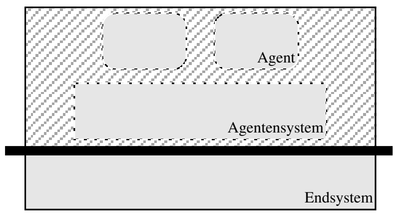

Abbildung 3.2: Sicherung durch das Endsystem

Vorteile:

- Sicherheitseigenschaften können völlig unabhängig von Agentensystem und Agenten zugesichert werden.
- Einfach zu implementieren, da Agentensystem und Agenten nicht verändert werden müssen.

Nachteile:

- Es ist nur ein Schutz des Endsystems erreichbar, der wechselseitige Schutz der anderen Entitäten des Modells ist nicht möglich, da diese vom Endsystem durch die "Black Box"-Sicht auf das Agentensystem nicht unterschieden werden können.
- Die dynamische Anpassung der Sicherheitsmaßnahmen ist faktisch unmöglich, da die Dynamik innerhalb des Agentensystems verborgen bleibt.
- Unterschiedliche Eigentumsverhältnisse von Agenten sind nicht mehr erkennbar, alle Agenten erscheinen als wären sie dem Eigentümer des Agentensystems zugeordnet.

- Ebenso durch die "Black Box"-Sicht bedingt, kann ein semantischer Zusammenhang von Einzelaktionen nicht hergestellt werden.
- Die Menge der möglichen Sicherungsmaßnahmen und ihre Ausprägung ist abhängig vom der Plattform des Endsystem. Dies stellt besonders für das Management von heterogenen Systemen, wie dies im Netz- und Systemmanagement gefordert ist, eine große Einschränkung dar.
- Die Sicherungsmaßnahmen sind selbst schlecht managebar, da sich die Management-Schnittstellen und Mechanismen zwischen verschiedenen Endsystem-Plattformen unterscheiden.

Somit sind Sicherungsmaßnahmen durch das Endsystem höchstens geeignet um das Endsystem selbst vor dem Agentensystem und dessen Agenten zu schützen. Für die restlichen Teile eines SmA kann damit kein Schutz gewährleistet werden.

3.6.2 Sicherung durch Agenten

Jegliche Sicherung wird durch die Agenten selbst vorgenommen. Diese sind alleinig für Überwachung und Durchsetzung von Zusicherungen verantwortlich.

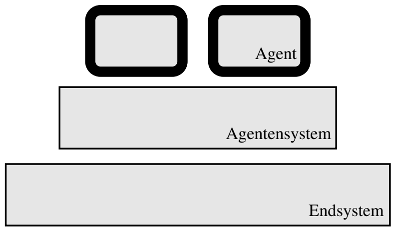

Abbildung 3.3: Sicherung durch Agenten

Vorteile:

- Ein Agent kann sich bezüglich seiner Semantik optimal selbst schützen.
- Feingranular: Schutzfunktionen können optimal auf die Semantik des Agenten abgestimmt werden.

Nachteile:

- Ein Agent kann sich nie vollständig selbst schützen, da er unter der Kontrolle des Agentensystems steht.
- Die möglichen Schutzmaßnahmen sind immer nur so gut, wie die eigene Implementierung dieser Maßnahmen.
- Agentensysteme und Endsysteme müssen Agenten voll vertrauen.
- Es können keine allgemeinen Zusicherungen gemacht werden.
- Es existiert keine zentrale Management-Komponente.

Werden Sicherungsmaßnahmen ausschließlich durch Agenten ergriffen, so ist ein Schutz des Agentensystems und des Endsystems nur dann gewährleistet, wenn alle Agenten deren Schutz realisieren.

3.6.3 Sicherung durch das Agentensystem

Die Sicherungsmaßnahmen werden durch das Agentensystem überwacht und durchgesetzt. Dabei wird die Eigenschaft genutzt, daß das Agentensystem die Laufzeitumgebung der Agenten implementiert, um Sicherungen zwischen Agenten durchzusetzen.

Vorteile:

- Agenten können wechselseitig voreinander geschützt werden, unabhängig davon, ob diese eigene Sicherungsmaßnahmen ergreifen.
- Mit dem Agentensystem steht eine im gesamten SmA vereinheitlichte (und damit plattformunabhängige) Schnittstelle zu Verfügung, die für das Management der Sicherheitsarchitektur genutzt werden kann.
- Sicherheitsmaßnahmen für das Endsystem können plattformunabhängig im Agentensystem realisiert werden, ohne auf spezielle Eigenheiten von Endsystemen eingehen zu müssen.

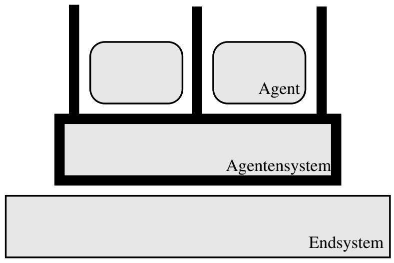

Abbildung 3.4: Sicherung durch das Agentensystem

Nachteile:

- Die Semantik der Schutzfunktionen kann keine Rücksicht auf spezielle Semantiken von Agenten nehmen.
- Endsysteme und Agenten müssen auf die Schutzmaßnahmen im Agentensystem vertrauen.

3.7 Folgerungen für ein Sicherheitsmodell

MASA bietet in seiner vorliegenden Form keinerlei Sicherheit und ist damit den in Kap. 2 skizzierten Angriffen schutzlos ausgeliefert.

Neben den bereits in Kap. 2.5 vorgestellten allgemeinen Anforderungen lassen sich zusammenfassend folgende Punkte für eine verbesserte Architektur und Implementierung identifizieren:

- Eine eindeutige Identifizierung aller Entitäten sollte gewährleistet werden.
- Ein Mechanismus zur wechselseitigen Authentisierung aller Entitäten sollte gefunden werden.
- Eine strikte Unterscheidung zwischen Agentengattung und Agenteninstanz sollte verwirklicht werden.
- Alle Kanäle zwischen Agentensystemen und Agenten (und damit jegliche Kommunikation) sollten gesichert werden. Dabei sollte auch der Schutz von lokalen Kanälen eines Agentensystems verbessert werden.
- Die Benutzung jeglicher Schnittstellen zwischen den Entitäten sollte autorisiert werden. Ebenso wie für das Agentensystem sollte ein äquivalenter Schutz der benutzten CORBA-Dienste bestehen.

Besonders sei erwähnt, daß aktuell viele Schnittstellen über mehrere Kanäle parallel erreichbar sind (beispielsweise können Methoden, sowohl über CORBA, als auch über Java aufgerufen werden). Hier sollte im Sinne von schlanken Schnittstellen eine Reduzierung durchgesetzt werden, da schlanke Schnittstellen durch eine verminderte Quantität kritischer Punkte übersichtlicher sind, was vor allem für die Autorisierung von Vorteil ist, da die Anzahl der hierfür zu überwachenden Schnittstellen sinkt.

Aus den Vor- und Nachteilen der verschiedenen Ansätze von Sicherungsmaßnahmen ergibt sich:

- Das Agentensystem als Laufzeitumgebung sollte Schutzmechanismen für lokale Agenten anbieten und sich selbst vor ihnen schützen.
- Für Agenten sollten Möglichkeiten geschaffen werden selbst individuelle Sicherungsmaßnahmen zu implementieren, deren Durchsetzung durch das Agentensystem unterstützt wird.
- Sicherungsmaßnahmen durch das Endsystem können nur zusätzliche Vorkehrungen darstellen, sie eignen sich nicht zum Schutz der anderen Entitäten.

Demnach muß sich ein Sicherheitsmodell in erster Linie auf Sicherungsmaßnahmen in Agentensystemen und den Agenteninstanzen abstützen.

Kapitel 4

Standards und Sicherheit in bestehenden SmA

4.1 Standards

4.1.1 Mobile Agent System Interoperability Facilities (MASIF)

Da MASA ein zur MASIF-Spezifikation ([OMG 98-03-09]) konformes Agentensystem darstellt, sollen nun die dort gestellten Bedingungen an eine Sicherheitsarchitektur aufgezeigt werden.

Die in der MASIF-Spezifikation durchgeführte allgemeine Analyse der Situation, der Gefahren und die daraus resultierenden generellen Anforderungen werden im wesentlichen bereits durch Kap. 2.1 abgedeckt und wird hier nicht wiederholt.

Authentisierung

Zur Authentisierung werden in der Spezifikation nur sehr generische Aussagen getroffen: MASIF postuliert einen Algorithmus, den sog. *"Authenticator"*, mit dem es möglich ist Agenten zu authentisieren. Nähere Angaben, wie solch ein *Authenticator* die Authentisierung durchzuführen hat, werden nicht gemacht.

Autorisierung

Anforderungen an die Autorisierung in SmA werden in MASIF ebenso allgemein beschrieben, wie das für Authentisierung der Fall ist. Im wesentlichen werden dabei folgende Aussagen gemacht, die als Empfehlungen und nicht als Vorschriften formuliert sind:

- Ein Menge von Regeln wird erstellt und als *"Security Policy"* definiert.
- Sowohl Agenten als auch Agentensysteme können eine oder mehrere *Security Policies* haben.
- Die Entscheidung welche *Security Policy* anzuwenden ist, kann von verschiedenen Faktoren abhängen, die aber nicht konkret festgelegt werden.

- Eine *Security Policy* enthält u.a. Regeln zur Erteilung bzw. Einschränkung der Möglichkeiten von Agenten, Limitierung des Ressourcenverbrauchs, Erteilung bzw. Verweigerung von Zugriffen.
- Für die Durchsetzung einer *Security Policy* ist ein *"Security Service"* verantwortlich.

Anforderungen an ein sicheres Agentensystem

Explizite Forderungen der Spezifikation an den *Security Service* sind im Abschnitt *"Security Service Requirements"* verzeichnet.

Der Begriff *"Credentials"* bezeichnet dabei allgemein alle sicherheitsrelevanten Attribute eines Objekts, wie z.B. die Identität oder Berechtigungen für die Ausführung bestimmter Aktionen.

Die gestellten Forderungen lauten konkret:

1. Authentisierung von Clients bei der entfernten Erzeugung von Agenten
   (a) Der *Security Service* muß Clients authentisieren.
   (b) Basierend auf der Identität des Clients werden die *Credentials* des neu erzeugten Agenten gesetzt.
   (c) Die *Credentials* des Clients bestimmen, welche *Security Policy* anzuwenden ist.
2. Authentisierung von Agentensystemen
   (a) Agentensysteme müssen sich wechselseitig authentisieren können.
   (b) Der Authentisierungsvorgang muß ohne Interaktion eines Benutzers erfolgen können.
3. Authentisierung von Agenten und Delegation
   (a) Migriert ein Agent, so muß dieser seine *Credentials* mit sich führen.
   (b) *Credentials* können am Ziel-Agentensystem verändert werden, abhängig vom Ergebnis der Authentisierung.
   (c) Wenn ein Agent einen entfernten Methoden-Aufruf durchführt, müssen die *Credentials* dieses Aufrufs jene des Agenten sein.
4. Security Policies von Agenten und Agentensystemen
   (a) Ein Agent sollte den Zugriff auf seine Methoden kontrollieren können.
   (b) Der Agent oder sein Agentensystem muß Zugriffskontrollmechanismen einrichten und ihre Einhaltung durchsetzten.
   (c) Werden Zugriffskontrollmechanismen von einem Agenten oder einem Agentensystem selbst definiert und selbst durchgesetzt, so müssen die *Credentials* eines entfernten Agenten dem Ziel Agentensystem bekannt sein.
   (d) Alternativ kann die Durchsetzung der Zugriffskontrollmechanismen von der Kommunikations-Infrastruktur übernommen werden.
5. Integrität, Vertraulichkeit, Replay Detection und Authentisierung
   (a) Der Initiator eines Kommunikationsvorgangs muß die Möglichkeit haben Anforderungen an Integrität, Vertraulichkeit und Art der Authentisierung zu spezifizieren.

(b) Die Kommunikations-Infrastruktur muß diese Anforderungen beachten, oder, falls die Anforderungen nicht erfüllt werden können, einen Fehler melden.

CORBA als Kommunikationsplattform

Da MASA CORBA als Kommunikationsplattform nutzt, ist weiterhin das Kapitel *"Security Service"* (2.4) der MASIF-Spezifikation zu beachten. Dort wird detailliert beschrieben, wie durch Nutzung von CORBA als Kommunikationsplattform die geforderten *Security Service Requirements* realisiert werden können.

Hierzu werden zunächst die aktuell verfügbaren CORBA-Implementierungen in drei Kategorien eingeteilt:

- Keine Sicherheits-Dienste (Kategorie 1): Es sind weder proprietäre noch standardisierte Sicherheits-Dienste implementiert.
- Proprietäre Sicherheits-Dienste (Kategorie 2): Produktspezifische Mechanismen zur Authentisierung und Zugriffskontrolle sind vorhanden, umfassen jedoch nicht den ORB selbst.
- Standardisierte Sicherheits-Dienste (Kategorie 3): Es ist ein ORB implementiert, der konform zur *CORBA Security Services Specification* (vgl. Abschnitt 4.1.2) ist.

Anschließend wird erörtert, wie die vorher an den *Security Service* gestellten Anforderungen im Rahmen der *CORBA Security Services Specification* erfüllt werden können. Faßt man die Diskussion zusammen, wird festgestellt, daß eine MASIF-konforme Implementierung für die Erfüllung der *Security Service Requirements* unter Nutzung von CORBA als Kommunikationsplattform, zwingend auf einen ORB mit *"Security Functionality Level 2"* (vgl. Abschnitt 4.1.2) angewiesen ist.

MASIF weist jedoch darauf hin, daß selbst dies nicht ausreichend ist, um alle gestellten Anforderungen zu erfüllen (aus [OMG 98-03-09]):

> " Although CORBA security does not currently meet all the needs of mobile agent technology, the MAF implementation must use available CORBA security to satisfy its security needs. Future versions of CORBA security should address these issues. "

4.1.2 CORBA Security Service Specification

Die *CORBA Security Service Specification* ([OMG 98-12-17]) ist ein CORBA-Basisservice, der Sicherheitsfunktionen beschreibt und Schnittstellendefinitionen enthält, die ein *CORBA Security*-konformer ORB implementieren muß.

Wegen des großen Umfangs der Spezifikation und weil zum jetzigen Zeitpunkt kein Produkt erhältlich ist, daß die *CORBA Security Service Specification* auch nur annähernd vollständig in Java implementiert, wird an dieser Stelle nur ein sehr knappes Resümee gegeben.

Aufbauend auf einem sehr abstraktem Sicherheitsmodell, definiert die Spezifikation Schnittstellen

- die ein ORB für die Implementierung von Anwendungen bereitstellen muß. Dabei wird generell zwischen Anwendungen, die aktiv selbst keine Sicherheitsmaßnahme ergreifen oder nutzen (*"security unaware"*) und solchen, die eigene Sicherheitsmaßnahmen implementieren und nutzen (*"security aware"*), unterschieden;

- für die Administration der Sicherheitseigenschaften eines ORB;
- für die Implementierung eines sicheren ORB.

Im weiteren wird beschrieben, wie existierende Protokolle (z.B. Kerberos oder SSL) für die Implementierung eines sicheren ORB eingesetzt werden können.

Generell muß ein ORB aber nicht alle in der Spezifikation geforderten Eigenschaften besitzen. Die prinzipielle Konformität wird in zwei Kategorien eingeteilt:

Security Functionality Level 1: Umfaßt nur Funktionalität, mit der im wesentlichen Anwendungen implementiert werden können, die selbst *"security unaware"* sind.

Security Functionality Level 2: Umschließt auch Funktionalität und Schnittstellen, die die Erstellung von Anwendungen erlauben, die *"security aware"* sind.

Ein SmA muß eigene, individuelle Schutzmaßnahmen ergreifen, weil die *CORBA Security Service Specification* dessen Anforderungen nicht erfüllen kann (vgl. Anschnitt 4.1.1). Deshalb ist ein SmA eine Anwendung, die *"security aware"* ist. Demnach muß, für die Realisierung, ein ORB die *Security Functionality Level 2* erfüllen, um die Sicherheitsanforderungen im Rahmen der *CORBA Security Service Specification* erfüllen zu können.

4.2 Überblick über Sicherheitseigenschaften in bestehenden SmA

Da MASA ein zur MASIF-Spezifikation konformes Agentensystem auf Java-Basis ist, werden nun andere, ebenfalls MASIF-konforme SmA, die in Java implementiert sind, kurz betrachtet. Eine detaillierte Untersuchung anderer Agentensysteme, die nicht MASIF-konform bzw. nicht in Java implementiert sind, würde an dieser Stelle den Rahmen sprengen. Eine knappe (keineswegs vollständige) Übersicht bietet [PhKa98].

4.2.1 Grasshopper

Das Agentensystem Grasshopper[1] ist ein Produkt der Firma IKV++ GmbH. Für die nachfolgenden Betrachtungen wurde die Version 1.2 anhand von [Gh Tech] und [Gh Basics][2] untersucht.

Im Grasshopper-Modell werden folgende Entitäten unterschieden:

Agents: Agenten

Agencies: Agentensysteme

Places: Gruppen von Agenten auf einem Agentensystem (vgl. [OMG 98-03-09])

Regions: Gruppen von Agentensystemen (vgl. [OMG 98-03-09])

[1] [Gh Site]

[2] Die verwendete Literatur bezieht sich auf die "Light"-Fassung von Grasshopper. Dort ist nur die Anzahl der Agenten im SmA eingeschränkt, die Sicherheitsfunktionen sind identisch zum Vollprodukt

Die im MASA-Modell vorhandenen Entitäten Endsystem, Implementierer, Benutzer und Benutzeroberfläche von Agenten (Applet) werden nicht identifiziert. Ein Endsystem wird implizit als Teil eines Agentensystems betrachtet. Weiterhin wird keine Unterscheidung zwischen Agentengattung und Agenteninstanz getroffen. Zu eindeutigen Identifikatoren für die Grasshopper-Entitäten werden keine Aussagen gemacht. Kanäle werden in der untersuchten Grasshopper-Version mit den Kommunikationsprotokollen CORBA/IIOP, Java RMI und Sockets realisiert.

Bezüglich der Sicherheit unterscheidet Grasshopper konzeptionell zwischen zwei Ausprägungen:

"external security": Hier werden nur die Sicherheitsanforderung auf Kanälen betrachtet.

"internal security": Untersucht nur die Sicherung der Schnittstellen des Agenten-/Endsystems.

Eine Authentisierung der einzelnen Grasshopper-Entitäten wird dabei nicht explizit betrachtet. Für die Autorisierung von Schnittstellen werden nur jene des Agenten-/Endsystems betrachtet. Schnittstellen der Agenten, insbesondere im Hinblick auf die unterschiedlichen Kanaltypen, werden nicht untersucht. *Places* und *Regions* sind nicht in das Sicherheitsmodell miteinbezogen.

In der Implementierung werden die von Grasshopper unterschiedenen Sicherheitsausprägungen folgendermaßen realisiert:

external security: Für die Sicherung der Kanaltypen Java RMI und Socket wird das SSL-Protokoll (vgl. Kap. 6.1.2) verwendet. Für CORBA/IIOP stehen keine Sicherungsmaßnahmen zur Verfügung. Zur Erteilung der für die SSL-Kommunikation benötigten X.509-Zertifikate werden keine Aussagen gemacht. Ebenso wird nicht dargelegt, welcher Entität des Grasshopper-Modells, die verwendeten X.509-Zertifikate zuzurechnen sind.

internal security: Die zur Verfügung stehenden Mittel zur Autorisierung und Überwachung der Schnittstellen basieren komplett auf der *Java Platform 2 Security Architecture* (vgl. Abschnitt 6.1.1) und umfassen somit nur die Schnittstellen, die durch das Java-API bereitgestellt werden.

Bewertung

Insgesamt müssen die Sicherheitseigenschaften von Grasshopper als nur in Ansätzen ausreichend für die Anforderungen an ein SmA bewertet werden. Die für Sicherheitsfragen relevante Gruppe der Benutzer fehlt im Grasshopper-Modell. Auf die Mobilität von Agenten, und den hierdurch notwendigen Integritätsschutz ihrer Daten, wird nicht eingegangen. Ebenso werden Delegation und unterschiedliche Vertrauensverhältnisse nicht betrachtet. An den bestehenden Mechanismen in der Implementierung ist, neben den Folgen aus dem dürftigen Konzept, vor allem der fehlende Schutz des CORBA-Kanals und die unklare Zuordnung der X.509-Zertifikate zu den Entitäten zu bemängeln.

4.2.2 Aglets

Aglets[3] ist ein SmA das am IBM Tokyo Research Laboratory entwickelt wurde.

Das Sicherheitsmodell von Aglets wird in [KLM 97] ausführlich beschrieben. Die darin unterschiedenen Entitäten sind:

Aglet: Instanz eines Aglet-Programms – Entspricht einer Agenteninstanz im MASA-Modell.

AgletManufacturer: Hersteller eines *Aglet*-Programms – Entspricht einem Implementierer einer Agentengattung im MASA-Modell.

AgletOwner: Eigentümer/Benutzer eines *Aglet* – Entspricht einem Benutzer einer Agenteninstanz im MASA-Modell.

Context: Programm das *Aglets* ausführt – Entspricht einem Agentensystem im MASA-Modell.

ContextManufacturer: Hersteller eines Context

ContextMaster: Eigentümer/Administrator eines *Context*.

Domain: Gruppe von *Contexts*, die dem gleichen Verantwortlichkeitsbereich unterliegen.

DomainAuthority: Eigentümer/Administrator einer *Domain*.

Im Vergleich zum MASA-Modell sind die folgenden Punkte auffällig. Das Endsystem des MASA-Modells wird unter dem *Context* subsumiert, die Benutzeroberflächen von Agenten werden nicht betrachtet. Eine explizite Unterscheidung zwischen Agentengattungen und Agenteninstanzen wird nicht getroffen, obwohl diese implizit durch den *AgletManufacturer* vorhanden ist. Identifikatoren für die einzelnen Entitäten werden informell angegeben, aber nicht explizit spezifiziert. Zur Authentisierung von *AgletManufacturer*, *AgletOwner*, *ContextManufacturer* und *ContextMaster* wird nur allgemein die Verwendung von Public-Key-Verfahren vorgeschlagen, die restlichen Entitäten werden nicht betrachtet. Ebenso knapp und allgemein wird die Kanalsicherung mittels Hash-Werten in Kombination mit Verschlüsselung und der Integritätsschutz der Daten eines Aglet durch digitale Signatur betrachtet.

Zur Autorisierung von Schnittstellen werden ausschließlich jene des *Context* bzw. des Endsystems untersucht. Hierfür wird eine umfassende Policy-Sprache informell spezifiziert, die unterschiedliche Benutzertypen (wie *manufacturer*, *owner*, *master* und *authority*) unterscheidet und Entscheidungshierarchien zwischen verschiedenen Policies definiert.

Eine Implementierung von Aglets liegt derzeit in zwei verschiedenen Versionen vor:

- Version 1.0.3, spezifiziert in [OsKa 97]
- Version 1.1beta1, spezifiziert in [OKO 98]

Beide Versionen basieren auf JDK 1.1.x. JDK 1.2 wird nicht unterstützt.

Die Version 1.0.3 soll hier nicht näher betrachtet werden, da zu erwarten ist, daß diese in Kürze vollständig durch eine Endversion basierend auf 1.1beta1 abgelöst wird. In der Version 1.1beta1 sind die folgenden Teile realisiert:

[3] [Aglet Site]

- Authentisierung von *Contexts* nur auf Gruppenbasis – alle *Context* innerhalb einer Gruppe können nicht voneinander unterschieden werden.
- Gruppen von *Contexts* werden durch einen gemeinsamen, geheimen Schlüssel gebildet, der manuell verteilt werden muß.
- Authentisierung der *AgletOwner* durch Paßwörter.
- Einfache Policy-Sprache in Anlehnung an jene von JDK 1.2 für die Autorisierung der *Context*- bzw. Endsystem-Schnittstelle.
- Kanalsicherung zwischen *Contexts* nur in Form von Prüfsummen (Hash-Werte), keine Verschlüsselung.

Bewertung

Das Aglets-Sicherheitsmodell bietet eine fortgeschrittene, wenn auch extrem detailarme Darstellung der Sicherheitsproblematik in SmA. Nicht betrachtet werden allerdings wichtige Punkte wie

- die Authentisierung von Agenten und Agentensystemen
- die Autorisierung der Schnittstellen von Agenten
- die Delegation von Rechten

Ebenso unterbleibt eine nähere Untersuchung von (Sicherheits-)Domänen, obgleich sie mit den Entitäten *Domain* und *DomainAuthority* bereits vorbereitet sind. Positiv hervorzuheben ist die relativ ausführliche, wenn auch größtenteils informelle Spezifikation einer Policy-Sprache zur Autorisierung der Schnittstelle des End-/Agentensystems.

In der vorliegenden Implementierung wurde allerdings ein Großteil der im Modell vorhandenen Sicherheitseigenschaften nicht umgesetzt. Vor allem die im Modell relativ konkret spezifizierte Policy-Sprache wurde nicht realisiert. Als besonders kritisch für den praktischen Einsatz muß die fehlende Verschlüsselung auf Kanälen angesehen werden.

Kapitel 5

Ein Sicherheitsmodell für MASA

In diesem Kapitel wird ein Sicherheitsmodell für MASA entwickelt, das Authentisierung und Autorisierung realisiert. Es berücksichtigt keine technischen Details und ist deshalb auf jedes zu MASA strukturell äquivalente Agentensystem übertragbar. Im nächsten Kapitel wird dann ein Konzept für eine Implementierung angegeben, welches die hier angestellten Überlegungen konkret auf MASA abbildet.

5.1 Einführung in das Sicherheitsmodell

Zum Entwurf eines Sicherheitsmodells stellt sich zu Beginn die Frage nach der prinzipiellen Vertrauenswürdigkeit der Umgebung. In Abhängigkeit der Antwort auf diese Frage lassen sich zwei mögliche extreme Grundannahmen formulieren:

- Prinzipiell vertrauenswürdig: “Es ist alles erlaubt, was nicht explizit verboten ist.” (positivistische Grundannahme)
- Prinzipiell nicht vertrauenswürdig: “Es ist alles verboten, was nicht explizit erlaubt ist.” (negativistische Grundannahme)

Für ein SmA kann dabei nur die negativistische Grundannahme formuliert werden, da für die positivistische Grundannahme eine konkrete Abschätzung der Gefahren notwendig wäre, was durch den generischen Ansatz eines SmA nicht möglich ist.

5.1.1 Die Basis des Sicherheitsmodells

Ausgehend vom MASA-Modell aus Kap. 3.1 bildet die folgende Verfeinerung die Basis aller Betrachtungen in diesem Kapitel, die in Abbildung 5.1 dargestellt ist.

Im MASA-Modell wurden zur Vereinfachung nur schlicht “Benutzer” betrachtet. Für ein Sicherheitsmodell sind jedoch auch die Rollen, die ein “Benutzer” einnimmt zu betrachten. Prinzipiell können “Benutzer” nämlich als Eigentümer und als Anwender auftreten, und diese sind nicht zwangsweise identisch.

Am Beispiel eines Agentensystems heißt dies, daß der “Benutzer”, der das Agentensystem startet und betreibt, und damit sein Eigentümer ist, in der Regel nicht alle Agenteninstanzen, die auf diesem Agentensystem erzeugt werden, selbst startet, sondern dies ein anderer Anwender ausführt.

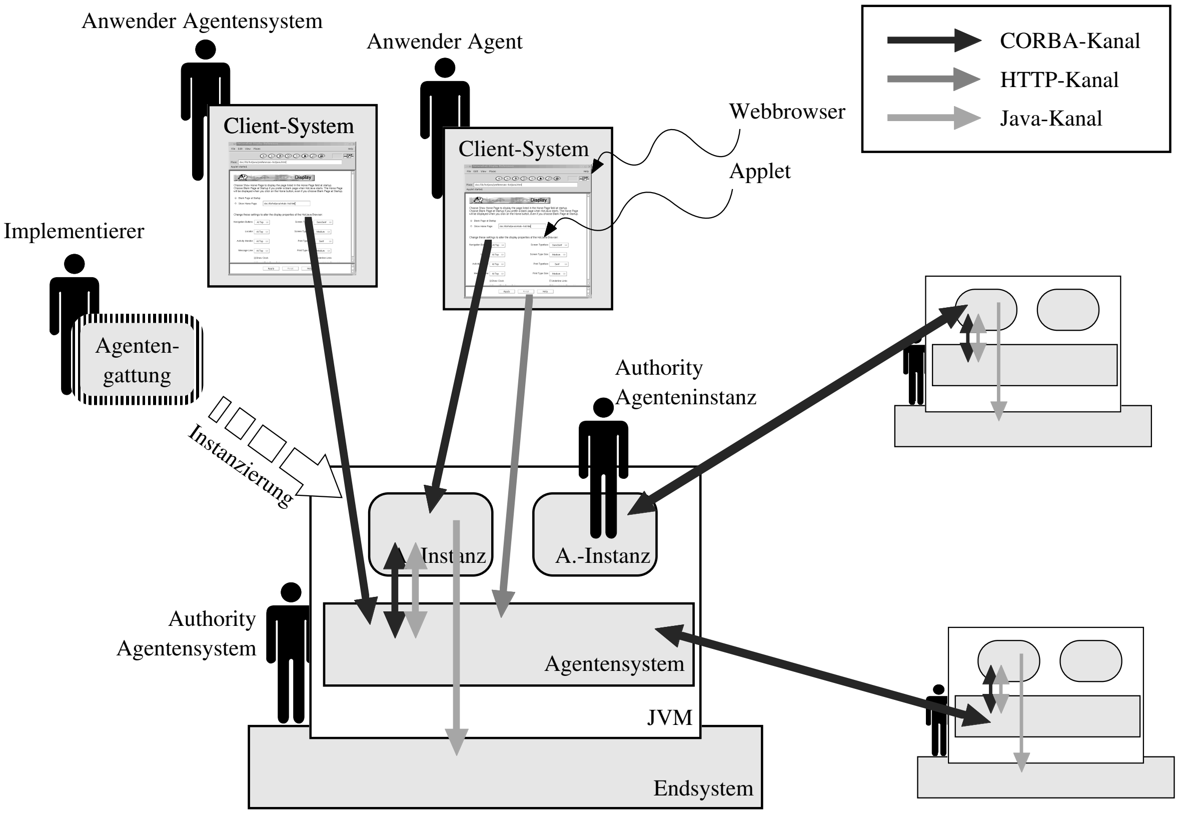

Abbildung 5.1: Basis des Sicherheitsmodells

Auf das Beispiel der Agenteninstanz zur Durchsatzmessung aus Kapitel 1.2 angewandt würde das bedeuten, daß der Provider der Eigentümer der Instanz des Meßagenten ist, der Management-Administrator aber der Anwender der Agenteninstanz.

In der MASIF-Spezifikation werden Eigentümer als *"Authority"* bezeichnet, weshalb dieser Begriff von nun an synonym verwendet wird.

Damit spalten sich die "Benutzer" des MASA-Modells in die folgenden Entitäten auf:

- Anwender einer Agenteninstanz
- Authority einer Agenteninstanz
- Anwender eines Agentensystems
- Authority eines Agentensystems

Die Oberklasse dieser Entitäten wird im Folgenden mit dem Begriff "Person" bezeichnet. "Person" spiegelt auch die Verhältnisse in einer realen Umgebung wider, in der nicht nur natürliche Personen ("Benutzer") auftreten, sondern auch juristische Personen relevant sind (beispielsweise eine GmbH als Authority eines Agentensystems), welche dann durch natürliche Personen vertreten werden.

Alle sonstigen Entitäten und die Kanäle werden unverändert vom MASA-Modell übernommen.

5.1.2 Vorgehen zur Erläuterung des Sicherheitsmodells

Fehlendes Vertrauen zwischen den einzelnen Entitäten in einem SmA ist die Ursache, weshalb überhaupt ein Sicherheitsmodell erstellt werden muß. Die im Einführungsbeispiel in Kap. 1.2 gestellten Fragen treten nur auf, weil der Händler dem Agenten nicht bedingungslos vertrauen kann. Deshalb werden zu Beginn des Sicherheitsmodells in Abschnitt 5.2 grundlegende Vertrauensbeziehungen identifiziert, die einige entscheidende Einschränkungen für die nachfolgenden Betrachtungen nach sich ziehen.

Um, unter Beachtung der negativistischen Grundannahme, eine wohldefinierte Ausgangsbasis schaffen zu können werden in Abschnitt 5.3 abgeschottete Ausführungsumgebungen für alle Entitäten betrachtet.

Das weitere Vorgehen wird dann durch die in Kap. 2 identifizierten Anforderungen und die in Kap. 3 entwickelten drei Sichtweisen bestimmt.

Abschnitt 5.4 beschäftigt sich mit der durch die Kanal-Sicht (Kap. 3.3) motivierten Sicherung der Kanäle.

Aus der Entitäten-Sicht (Kap. 3.2) wurde unmittelbar deutlich, daß eine eindeutige und sichere Identifizierung der Entitäten notwendig ist. In Abschnitt 5.5 werden eingangs die Möglichkeiten hierzu für alle Entitäten des Modells untersucht und anschließend, basierend auf Zertifikathierarchien, eine konkretes Verfahren angegeben, wie die Authentisierung einzelner Entitäten realisierbar ist.

Mit dem Integritätsschutz von Informationen beschäftigen sich die beiden daran anschließenden Abschnitte. Abschnitt 5.6 bezieht dabei die konzeptionelle Unterscheidung von Agenteninstanzen und Agentengattungen in das Sicherheitsmodell ein, während sich Abschnitt 5.7 mit den durch die Mobilität der Agenteninstanzen gefährdeten Daten beschäftigt. Darin werden die im Zuge der Authentisierung eingeführten Mittel nicht nur zur Sicherstellung der Integrität von Informationen, sondern auch zur gesicherten Kenntlichmachung des Erstellers von Informationen eingesetzt.

Basierend auf der Authentisierung und den mit einem Integritätsschutz versehenen Daten, wird in Abschnitt 5.8 die Autorisierung in einem SmA untersucht, deren Notwendigkeit durch die Schnittstellen-Sicht (Kap. 3.4) ergibt. Um die Kooperation zwischen Agenten zu ermöglichen, gibt Abschnitt 5.9 ein Konzept zur Delegation von Rechten an.

Zum Abschluß der Modells wird dann in Abschnitt 5.10 beschrieben, wie mit den bis dato vorgestellten Mitteln, der Begriff einer Sicherheitsdomäne definiert werden kann. Dieser Domänenbegriff wird dann den in einem SmA notwendigen Anforderung aus organisatorischen Strukturen und Verantwortlichkeiten gerecht. In Abschnitt 5.11 werden schließlich konkrete Fallbeispiele aus dem Lebenszyklus einer Agenteninstanz, unter Anwendung des entwickelten Modells, betrachtet.

Hinweise zu vorausgesetzten Grundlagen:

Für das Verständnis des in diesem Kapitel vorgestellten Modells sind Grundkenntnisse in kryptologischen Methoden und Verfahren, insbesondere der Public-Key-Verfahren unerläßlich. Eine detaillierte Darstellung ist an dieser Stelle wegen des großen Umfangs nicht möglich und unterbleibt deshalb. Ein leicht verständlicher Überblick ist beispielsweise in [Tane 98b] enthalten, [Baue 97] enthält eine kurze, mathematisch orientierte Betrachtung.

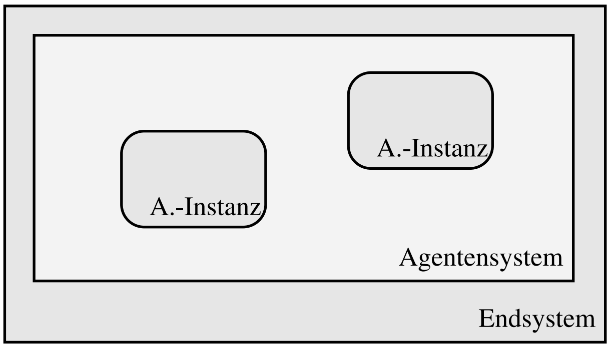

Abbildung 5.2: Einbettungsbeziehung

5.2 Erzwungenes Vertrauen durch Einbettungsbeziehungen

Betrachtet man die Einbettungsbeziehungen der Entitäten Endsystem, Agentensystem und Agenteninstanz in Abb. 5.2, so fällt die hierarchische Einbettung der Entitäten ineinander auf.

Ein Endsystem bzw. das darauf ablaufende Betriebssystem stellt die Ausführungsumgebung für Agentensysteme dar. Diese werden unter der absoluten Kontrolle des Endsystems ausgeführt. Völlig äquivalent ist die Beziehung zwischen Agentensystem und Agenteninstanzen gestaltet.

Die Folge dieser Einbettungsbeziehungen ist, daß eine Komponente, die komplett unter der Kontrolle einer anderen ausgeführt wird, dieser vertrauen muß. Konkret:

- Ein Agentensystem muß seinem Endsystem vertrauen.
- Eine Agenteninstanz muß seinem Agentensystem vertrauen.

Dies bedeutet, daß von Seiten der Agenteninstanzen/Agentensysteme Fragen, wie

- Wird ein(e) Agenteninstanz/Agentensystem vom Agentensystem/Endsystem korrekt ausgeführt?
- Wird ein(e) Agenteninstanz/Agentensystem nicht vorzeitig terminiert?
- Wird eine Agenteninstanz auf das angeforderte Agentensystem wunschgemäß migriert?

nahezu unbeantwortet bleiben müssen. Zwar gibt es Versuche (beispielsweise in [SaTs 98], [Vign 98a], [Hohl 98]) diese Problematik zumindest eingeschränkt zu lösen, da diese sich aber noch in einem frühen Stadium der Entwicklung befinden, sollen sie hier nicht weiter betrachtet werden.

Demnach werden im weiteren Verlauf keine Sicherungsmaßnahmen betrachtet, die von einer eingebetteten Entität ergriffen werden und sich gegen jene Entität richten, die die eigene Laufzeitumgebung bereitstellt. Konkret gelten die folgenden uneingeschränkten Vertrauensbeziehungen:

- Ein Agentensystem vertraut "seinem" Endsystem.
- Eine Agenteninstanz vertraut "ihrem" Agentensystem.

5.3 Wechselseitige Abschottung der Entitäten

Um den in Kap. 2.5 geforderten wechselseitigen Schutz der Entitäten realisieren zu können, muß die Existenz getrennter Umgebungen für alle Entitäten gewährleistet werden. Hierbei sind nur Agentensysteme und Agenteninstanzen zu betrachten, da die anderen Entitäten des Modells per se über eigene Ausführungsumgebungen verfügen.

5.3.1 Abschottung zwischen Agentensystemen

In der Praxis wird die Trennung der Ausführungsumgebungen der Agentensysteme in den meisten Fällen schon dadurch gewährleistet, daß verschiedene Agentensysteme auf verschiedenen physischen Endgeräten ausgeführt werden. Für den Fall, daß mehrere Agentensysteme auf einem Endsystem ablaufen sei angenommen, daß auf dem Endsystem ein Betriebssystem abläuft, welches ein Prozeßmodell implementiert, das eine Abschottung der Prozesse voreinander realisiert. Mit dieser Annahme genügt die Forderung, daß jedes Agentensystem durch einen Prozeß auf dem Endsystem realisiert wird.

5.3.2 Abschottung zwischen Agenteninstanzen

Für den wechselseitigen Schutz von Agenteninstanzen, die auf dem gleichen Agentensystem ausgeführt werden, muß die Trennung ihrer Ausführungsumgebungen näher betrachtet werden.

Das Agentensystem, welches die einzelnen Umgebungen bereitstellt, muß dabei äquivalente Eigenschaften aufweisen, die ein Betriebssystem zur Trennung von Prozeßumgebungen realisiert:

- Speicherschutz
- Trennung der Laufzeitumgebungen von Agenteninstanzen
- Ressourcenüberwachung
- Trennung der Namensräume

Eine Trennung der Namensräume der Agenteninstanzen ist für eine Trennung der Ausführungsumgebungen essentiell, insbesondere in Systemen, die die Laufzeitbindung von Bibliotheken unterstützen. Zwar sind nach Kap. 3.2 keine Kollisionen in der Namensgebung zwischen den Agenteninstanzen selbst zu befürchten, dies gewährleistet jedoch noch nicht, daß Komponenten, die von Agenteninstanzen “mitgebracht” und benutzt werden, sich ebenfalls in disjunkten Bereichen ihres Namensraums befinden.

An einem Beispiel erläutert könnte dies bedeuten: Benutze ein Agent A eine Laufzeitbibliothek namens L. Diese sei nicht Teil des Agentensystems, sondern der statischen Gattungsdaten des Agenten, und deshalb erst dann dem Agentensystem bekannt, wenn eine Instanz des Agenten das Agentensystem betritt. Ein zweiter Agent B benutze ebenfalls eine Bibliothek namens L, diese sei jedoch von einem anderen Hersteller und implementiere eine andere Semantik, als jene die vom Agenten A benutzt wird.

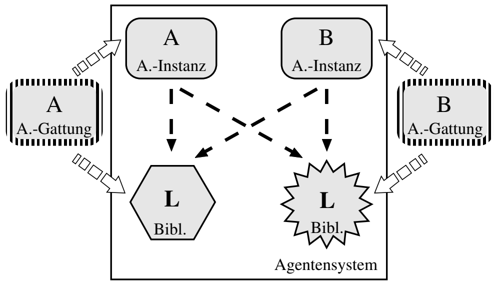

Abbildung 5.3: Nicht eindeutiger Bibliothekszugriff mit gemeinsamen Namensraum

Werden nun Instanzen von A und B auf einem Agentensystem ausgeführt, so kann ohne getrennte Namensräume im allgemeinen keine Aussage darüber getroffen werden, welche Bibliothek L durch A bzw. B benutzt wird. Dies wird im konkreten Fall durch die verwendete Implementierungssprache bestimmt, bzw. durch die verwendete Laufzeitumgebung. Im wesentlichen bestimmt dabei die Implementierung des Runtime-Linkers, ob z.B. eine "first load, first use"-Semantik oder "last load, first use"-Semantik benutzt wird.

Nur die Trennung der Namensräume der beiden Agenteninstanzen kann gewährleisten, daß jeder der beiden Agenten auf "seine" Bibliothek zugreift, obwohl beide Versionen der Bibliothek unter dem gleichen Namen angesprochen werden.

An diesem Beispiel wird deutlich, wie sich bei gemeinsamen Namensräumen durch Einschleusung von Bibliotheken auf ein Laufzeitsystem mit "last load, first use"-Semantik Maskeradeattacken durchführen ließen, indem eine Bibliothek mit neuer Semantik eine bestehende gleichen Namens ersetzt.

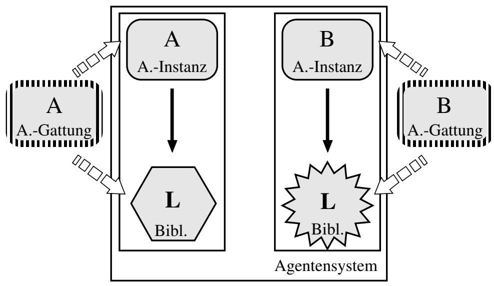

Abbildung 5.4: Bibliothekszugriff mit getrennten Namensräumen

Da auf dynamische Laufzeitbibliotheken, ohne massive Einschränkung des generischen Ansatzes von SmA, nicht verzichtet werden kann, folgt hieraus, daß eine Trennung der Namensräume für Agenteninstanzen unabdingbar ist.

5.3.3 Abschottung zwischen Agentensystem und Agenteninstanzen

Äquivalent zur Trennung der einzelnen Umgebungen der Agenteninstanzen muß die Ausführungsumgebung des Agentensystems zu denen der Instanzen gewährleistet werden. Hierfür kommen ebenfalls die im vorhergehenden Abschnitt genannten Techniken zum Einsatz.

Insbesondere muß das Agentensystem selbst über einen eigenen Namensraum verfügen, der getrennt von jenen der Agenteninstanzen ist, was einen entsprechenden Schutz des Agentensystems selbst bewirkt. So werden vor allem Maskerade-Attacken auf Bibliotheken des Agentensystems verhindert, die eine Gefährdung des gesamten Agentensystems darstellen.

5.4 Kommunikationskanäle

Konzeptionell lassen sich zur Gestaltung der Kanäle (vgl. Tab. 3.2) einige allgemeine Verfahren bestimmen, um Angriffe, wie sie in Kap. 2.3 beschrieben sind, zu unterbinden. An dieser Stelle ist zu betonen, daß hier nur die Kanäle an sich betrachtet werden, die Sicherung der "Enden" der Kanäle, die Schnittstellen, werden später ausführlich im Rahmen der Autorisierung behandelt.

Chiffrierung und digitale Signatur der über einen Kanal versandten Daten bieten sich an, um Angriffe durch Abhören, Umänderung und Man-in-the-Middle-Attacken unterbinden zu können. Da, technisch gesehen, sowohl CORBA/IIOP als auch HTTP mittels Punkt-zu-Punkt Verbindungen realisiert werden, kann dies durch Nutzung von Public-Key-Verfahren geleistet werden. Kombiniert man nun noch Chiffrierung und Signatur mit einem Protokoll welches die Daten mit Zeitstempeln oder Seriennummern versieht, lassen auch Attacken durch Wiedereinspielen wirksam ausschließen.

Den Java-Kanal betreffend sind Angriffe wie Abhören, Umänderung, Man-in-the-Middle und Wiedereinspielen ausgeschlossen, da dieser bereits durch die JVM mit ihrem Speicherschutz (siehe hierzu 6.1.1) hinreichend geschützt wird.

5.5 Authentisierung

In diesem Abschnitt wird festgelegt, welche Entitäten des MASA-Modells aus Kapitel 3.1 authentisierbar sind, und von welchen Entitäten dieses durchgeführt wird.

5.5.1 Authentisierbare Entitäten

Für eine erfolgreiche Authentisierung muß eine der folgenden Voraussetzungen erfüllt sein:

- Die zu authentisierende Entität muß von der authentisierenden Entität direkt inspiziert werden können, oder
- eine andere, vertrauenswürdige Instanz, die selbst authentisierbar sein muß, übernimmt stellvertretend die Authentisierung, und bietet hierfür entsprechende Schnittstellen an.

Handelnde Entitäten: Personen, Agenteninstanzen und Agentensysteme

Alle unmittelbar handelnden Entitäten lassen sich verallgemeinert als Klasse von Subjekten verstehen. Darunter fallen alle Personen, sowie Agenteninstanzen und Agentensysteme. In Anlehnung an die Terminologie des *CORBA Security Service* werden diese als *Principals* bezeichnet. Da für ein sicheres SmA alle Handlungen autorisiert werden müssen, folgt daraus unmittelbar, daß es möglich sein muß alle *Principals* zu authentisieren.

Um die Authentisierung von *Principals* durchführen zu können, müssen diese zunächst mit einem eindeutigen Identifikator versehen werden. Tabelle 3.1 gibt einen solchen Identifikator bereits für nahezu alle Typen an.

Für die noch ausstehende Klasse der Personen erscheint eine Kombination von Namen und organisatorischer Klassifizierung ausreichend, um diese hinreichend identifizieren zu können. Der *Distinguished Name* (DN) des X.500 Standards ([X.500]) gibt ein Beispiel hierfür an.

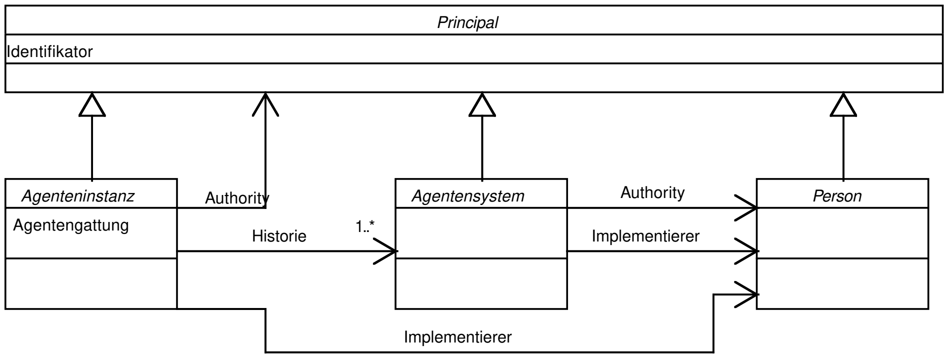

Abbildung 5.5: Klassifizierung von *Principals*

Neben dem gemeinsamen Attribut Identifikator aller *Principals*, müssen die Subtypen Agentensystem und Agenteninstanz noch mit weiteren Attributen versehen werden, um ihre Beziehungen zu Personen des Modells herzustellen.

Für Agentensysteme und Agenteninstanzen sind jeweils die Authority und der Implementierer zu betrachten. Dabei kann die *Authority* einer Agenteninstanz selbst wiederum eine Agenteninstanz, ein Agentensystem oder eine Person, also ein *Principal*, sein. Die *Authority* eines Agentensystem kann dagegen nur eine Person sein. Ein Implementierer kann ebenfalls nur vom Typ Person sein.

Um der Mobilität von Agenteninstanzen Rechnung zu tragen, muß die Historie einer Agenteninstanz betrachtet werden. Hierzu muß eine Liste von besuchten Agentensystemen als Attribut einer Agenteninstanz hinzugefügt werden.

In Abbildung 5.5 sind Klassenbeziehungen und Attribute in einem UML[1]-Diagramm zusammengefaßt. Es beschreibt die Klassenhierarchie durch Generalisierungen, sowie die typisierten Attribute der einzelnen Klassen durch Assoziationen. Dabei geben die Pfeilspitzen der Assoziationen die Richtung an, in der diese sichtbar ist. D.h. beispielsweise, daß aus Sicht eines Agentensystems die Person, die als Authority auftritt, definiert ist. Aus Sicht der Person aber kann nicht unmittelbar bestimmt werden, für welche Agentensysteme sie als Authority fungiert.

Agentengattungen

Agentengattungen sind selbst zwar keine handelnden Entitäten, dennoch muß ihre Unveränderlichkeit sichergestellt werden. Dabei muß auch nachvollziehbar sein, durch welche Person eine Agentengattung erstellt worden ist. Durch den Implementierer einer Gattung läßt sich in der Autorisierung nämlich ableiten, inwieweit man einer Agenteninstanz überhaupt vertrauen kann, weil der Implementierer, als "Bürge für die Korrektheit", als vertrauenswürdig eingestuft werden muß (vgl. Abschnitt 5.6).

Erreicht wird die Zuordnung des Implementierers, indem die Daten der Agentengattung durch ihn digital signiert werden.

[1][RJB 98]

Applets

Applets müssen entweder von Anwendern authentisiert werden, nämlich dann, wenn sie Agenteninstanzen steuern wollen, oder von Agenteninstanzen, wenn eine Aktion im Auftrag eines Applets ausgeführt werden soll.

Anwender können Applets authentisieren, wenn ihr Code digital signiert wurde, man bezeichnet diese dann auch als *SignedApplets*. Die gängigen Webbrowser bieten hierzu entsprechende Benutzerschnittstellen an. Sieht man Applets als Teil der Agentengattung, sind sie bereits signiert und damit für den Anwender authentisierbar.

Für Agenteninstanzen besteht aber keine Möglichkeit die Authentizität eines Applets zu überprüfen, da

- Applets auf einem entfernten System ausgeführt werden, und somit nicht direkt durch die Agenteninstanz inspiziert werden können, und
- kein Webbrowser, der als stellvertretende Instanz die Authentisierung theoretisch vornehmen könnte, Schnittstellen hierfür anbietet.

Hieraus resultiert ein zum gegenwärtigen Zeitpunkt nicht lösbares Sicherheitsproblem. Es ist nämlich denkbar, daß beispielsweise ein feindliches Applet auf einem Browser mit Wissen des Anwenders zum Einsatz kommt, welches unerwünschte Aktionen gestattet. Auf diese Problematik wird in Kap. 8 noch eingegangen.

Aus der Sicht des Anwenders ist also die Authentisierung von Applets geklärt. Aus der Sicht von Agenteninstanzen soll sie nicht weiter betrachtet werden. Dies bedeutet jedoch nicht, und das sei hier betont, daß ein Anwender, der ein Applet steuert, ebenfalls nicht authentisiert werden kann. Da dies durch die oben aufgezeigte Authentisierbarkeit von Personen möglich ist, relativiert sich das Problem aus Sicht der Agenteninstanz, weil mit dem Anwender der ursächlich Verantwortliche der durch das Applet übermittelten Aktion, sicher identifiziert werden kann.

Endsystem und Client-Systeme

Eine ähnliche Situation zeigt sich bei den Endsystemen und den Client-Systemen. Für ein Agentensystem oder eine Agenteninstanz kann keine plattformübergreifende Möglichkeit vorausgesetzt werden, ein entferntes Endsysteme bzw. Client-Systeme *direkt* zu authentisieren. Die Authentisierung von Endsystemen muß deshalb *mittelbar* über das entfernte Agentensystem vorgenommen werden. Das entfernte Agentensystem fungiert dann als Stellvertreter für das Endsystem. Diesem Agentensystem muß somit insofern vertraut werden, als das von ihm angegebene Endsystem tatsächlich jenes ist, auf dem das Agentensystem abläuft.

Da auf Client-Systemen keine Entität existiert, die stellvertretend ihre Authentisierung übernehmen könnte, muß diese unterbleiben. Es sei ausdrücklich erwähnt, daß der Webbrowser über keine Möglichkeiten zur Authentisierung seines Client-Systems verfügt. Ebenso sind IP-Adressen keine sicheren Identifikatoren für Client-Systeme, da diese durch Techniken wie Proxies, Firewalls oder Network Address Translation (NAT) (sogar gewollt) verändert werden können.

Endsysteme und Client-Systeme werden deshalb, ebenso wie Applets, für die Authentisierung aus der Sicht von entfernten Entitäten nicht weiter betrachtet.

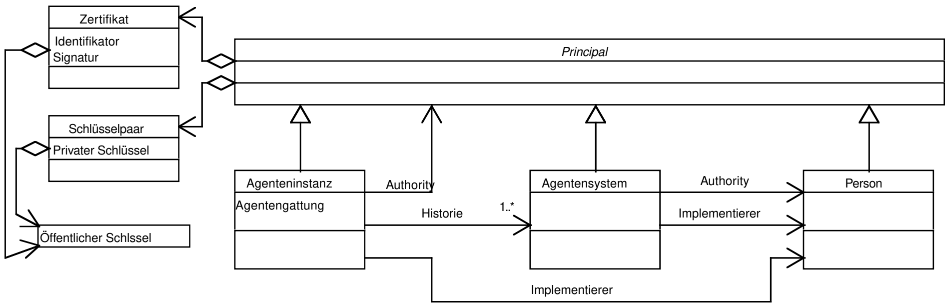

Abbildung 5.6: *Principals* mit Zertifikaten

5.5.2 Zertifikate als Beleg der Identität

Somit verbleiben folgende Klassen von Entitäten für die weiteren Betrachtungen: Agenteninstanzen, Agentensysteme und Personen, alle zusammengefaßt unter der Superklasse *Principal*. Um nun die Authentizität des Identifikators eines *Principals* belegen zu können bieten sich Public-Key-Verfahren, wie im Folgenden beschrieben, an.

Seien *Principals* nun mit einem Schlüsselpaar, bestehend aus dem geheimen und dem öffentlichen Schlüssel, ausgestattet.

Nachdem jeder *Principal* eindeutig identifiziert werden kann und Eigentümer eines Schlüsselpaares ist, kann daraus ein Zertifikat gebildet werden. Dies geschieht, indem die Entität ihren Identifikator mit ihrem geheimen Schlüssel unterzeichnet, woraus eine zugehörige Signatur entsteht. Das 3-Tupel aus Identifikator, Signatur und öffentlichem Schlüssel ist dann das Zertifikat der Entität.

Damit kann ein Kommunikationspartner den ersten Schritt einer Authentisierung durchführen. Bekommt er nämlich den Identifikator, die Signatur des Identifikators und den öffentlichen Schlüssel übermittelt, kann er feststellen ob der Identifikator vom Eigentümer des zum öffentlichen Schlüssel gehörigen privaten Schlüssels unterzeichnet worden ist.

Somit reduziert sich das Problem der Authentisierung auf die Frage, ob das verwendete Schlüsselpaar tatsächlich zu jenem *Principal* gehört, dessen Identifikator unterschrieben worden ist. Dieses Problem wird durch Zertifikatketten gelöst. Wurde nämlich das Zertifikat des zu überprüfenden *Principals* durch ein anderes *Principal* signiert, so steht dieser als Bürge für die Echtheit des fraglichen Zertifikats mit seinem eigenen Zertifikat ein. Da das Zertifikat des Bürgen wiederum signiert sein kann, entsteht eine Zertifikatkette. Vertraut die überprüfende Instanz einem Zertifikat aus dieser Kette, so ist die Authentisierung des fraglichen *Principals* gelungen.

Um nun alle *Principals* des Modells authentisieren zu können genügt es also

- in der Klasse *Principal* ein Attribut Schlüsselpaar einzuführen,
- in der Klasse *Principal* das Attribut Identifikator durch ein Attribut Zertifikat zu ersetzen, in dem der Identifikator bereits enthalten ist,
- ein Verfahren zum Ausstellen und Signieren von Zertifikaten zur Bildung einer Zertifikatkette festzulegen.

Damit können dann Agentensysteme, Agenteninstanzen und jegliche Personen auf der Basis von vertrauenswürdigen Zertifikaten authentisiert werden.

Außerdem wurde damit auch eine Voraussetzung für die abhörgesicherte Kommunikation zwischen *Principals* geschaffen, da die Schlüsselpaare der *Principals* für eine chiffrierte Kommunikation nach Abschnitt 5.4 verwendet werden können.

5.5.3 Ausstellen von Zertifikaten in SmA

Die Ausstellung von Zertifikaten und die Errichtung einer Vertrauensbeziehung darüber ist an zwei Bedingungen gebunden, die zwar einfach zu formulieren, im Fall SmA aber teilweise schwierig zu realisieren sind:

1. Eine Zertifikat kann nur solange gelten, wie sich die Identität, respektive der Identifikator, einer Entität nicht ändert.
2. Die Entität, für die ein Zertifikat ausgestellt wird, muß ihren geheimen Schlüssel schützen können.

Punkt 1 wird mit den in Abschnitt 5.5.1 vorgestellten Identifikatoren für die Lebensdauer aller *Principals* gewährleistet.

Punkt 2 ist für den Fall von realen Personen trivial. Zum Beispiel kann der private Schlüssel mittels eines Paßworts chiffriert werden, wobei dieses Paßwort dann nur dem Eigentümer des Schlüssels bekannt ist.

Ebenso kann ein Agentensystem seinen privaten Schlüssel schützen, indem er in einem Bereich abgelegt wird, der für Agenteninstanzen und Personen nicht zugänglich ist. Im wesentlichen ist dieser Fall äquivalent zu einem klassischen Anwendungsprogramm, welches mit kryptographischen Schlüsseln arbeitet.

Für Agenteninstanzen kann Punkt 2 allerdings nicht gewährleistet werden. Da Agenteninstanzen ohne interaktive Eingriffe von Personen handlungsfähig sein sollen, ist die Verwendung von paßwortgeschützter Schlüssel nicht praktikabel. Ebenso kann eine Agenteninstanz ihren privaten Schlüssel nicht durch Ablage in einem gesicherten Bereich schützen, da sie völlig unter der Kontrolle des Agentensystems steht. Damit hat das Agentensystem aber auch Zugriff auf den geheimen Schlüssel der Agenteninstanz.

Dies wäre solange unproblematisch, solange die Agenteninstanz auf einem vertrauenswürdigen Agentensystem abläuft. Migriert sie aber auf ein Agentensystem mit feindlichen Absichten, kann dieses den privaten Schlüssel auslesen und für eigene oder fremde Attacken nutzbar machen. So könnten feindliche Entitäten ihre Identität fälschen, oder an Informationen gelangen, die für die Entschlüsselung durch den privaten Schlüssel chiffriert wurden.

Weiterhin macht es keinen Sinn den privaten Schlüssel während einer Migration mit sich zu führen, da dieser ansonsten abgehört werden kann, wenn ein Kanal ungesichert ist.

Aus diesen Gründen muß also für Agenteninstanzen ein Weg gefunden werden, wie diese mit Schlüsselpaaren versorgt werden, deren Vertrauenswürdigkeit gewährleistet werden kann.

Hierzu bietet sich das Agentensystem als Indirektionsstufe an. Agentensysteme erfüllen die Punkte 1 und 2. Diese können nun

- neue Schlüsselpaare und
- neue Zertifikate

für Agenteninstanzen ausstellen, die ihre Gültigkeit behalten, solange die Agenteninstanz sich auf dem Agentensystem befindet. Diese Zertifikate sollen, wegen der Begrenzung ihrer Gültigkeit, als Sitzungszertifikat bezeichnet werden.

Um die Vertrauenswürdigkeit des Sitzungszertifikats überprüfen zu können, müssen dann die folgenden Schritte durchgeführt werden:

1. Zuerst muß die Identität des Agentensystems sichergestellt werden, d.h. die Gültigkeit und Vertrauenswürdigkeit des Zertifikats des Agentensystems überprüft werden. Hierzu durchsucht man die Zertifikatkette des Agentensystems nach einem vertrauenswürdigen Unterzeichner. Findet man einen solchen vertrauenswürdigen Unterzeichner, so kann die Identität des Agentensystems als gesichert angesehen werden.

2. Dann kann entschieden werden, ob man diesem Agentensystem per se vertrauen kann, insofern dieses Agentensystem keine feindlichen Absichten hegt, Agenteninstanzen korrekt ausführt, etc.

3. Wurde auch über Schritt 2 positiv entschieden, so kann das Sitzungszertifikat der Agenteninstanz dahingehend überprüft werden, ob es von jenem vertrauenswürdigen Agentensystem auch unterschrieben wurde.

Mit dem Gelingen von Schritt 3 ist das Sitzungszertifikat einer Agenteninstanz erfolgreich überprüft und damit die Identität der Instanz gesichert, wobei das Agentensystem als Bürge darüber auftritt. Schlägt dagegen einer der genannten Schritte fehl, so kann die Agenteninstanz nicht authentisiert werden, und folglich ist diese als nicht vertrauenswürdig einzustufen, da möglicherweise ein Angriffsversuch vorliegt.

Der Nachteil dieses Verfahrens besteht darin, daß sich, obwohl die Identität einer Agenteninstanz während seines Lebenszyklus unverändert bleibt, das Zertifikat mittels dessen er sich authentisiert sehr wohl verändert, was zu einem erhöhten Aufwand bei der Authentisierung führt.

Ausgehend davon, daß Schlüsselpaare auch zur Chiffrierung auf Kanälen benutzt werden, birgt das vorgestellte Verfahren allerdings auch einen Vorteil. Betritt nämlich eine Agenteninstanz ein bösartiges Agentensystem, welches ihren privaten Schlüssel zum Zweck des Abhörens der Kommunikation preisgibt, so wird zwar jegliche Kommunikation abhörbar, migriert die Instanz aber auf ein anderes, "gutes" Agentensystem, ist ihre Kommunikation wieder abhörsicher, da sie dort mit einem neuen Schlüsselpaar versehen wird und somit der alte private Schlüssel wertlos wird. Damit ist die Kommunikation der Agenteninstanz zumindest nur zeitweise ungeschützt.

5.5.4 Ein Verfahren zur Erteilung von Zertifikaten

Unter den oben vorgestellten Randbedingungen wird nun ein Verfahren vorgestellt, wie in einem SmA Zertifikate erstellt, und davon abgeleitet, ihre Vertrauenswürdigkeit überprüft werden kann.

Zertifikate für die Authority eines Agentensystems

Das Zertifikat für die Authority eines Agentensystems muß "out of band" erteilt werden. Dies bedeutet, daß ein solches Zertifikat nicht durch eine Entität des SmA selbst erstellt werden

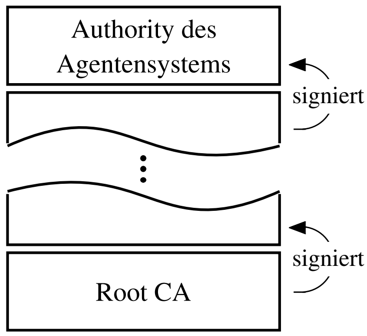

Abbildung 5.7: Zertifikatkette der Authority eines Agentensystems

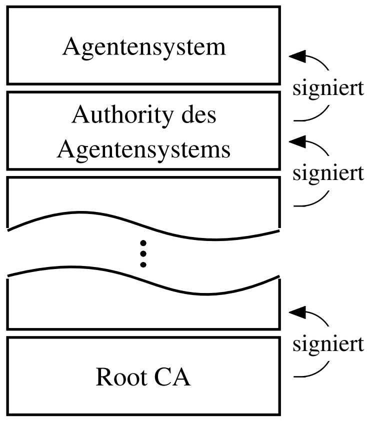

Abbildung 5.8: Zertifikatkette eines Agentensystems

kann. Ein solches Zertifikat könnte beispielsweise durch eine bereits bestehende Zertifizierungshierarchie erstellt werden. In Abb. 5.7 wird dies angenommen.

Die "out of band"-Verteilung begründet sich damit, daß alle Zertifikate für Agentensysteme und Agenteninstanzen direkt bzw. indirekt auf dem Zertifikat einer Authority eines Agentensystems basieren. Würde das Authority-Zertifikat selbst aus einem Teil des SmA generiert werden, würde es zu einem "Kreisschluß" in den Zertifikatketten kommen, womit keinen Vertrauensbeziehungen mehr aufgebaut werden könnten.

Zertifikate für Agentensysteme

Damit kann nun das Zertifikat eines einzelnen Agentensystems erstellt werden:

- Beim Start des Agentensystems erzeugt dieses selbständig ein Schlüsselpaar.
- Das Agentensystem generiert ein Zertifikat, daß die charakteristischen Eigenschaften des Agentensystems (wie z.B. Name des Endsystems, Version des Agentensystems, etc.) enthält.
- Die Authority des Agentensystems signiert das vom Agentensystem generierte Zertifikat mit ihrem Schlüssel.

Die sich aus diesem Verfahren ergebende Zertifikatkette ist in Abb. 5.8 dargestellt. Somit wird die Authority eines Agentensystems immer durch das vorletzte Zertifikat in der Zertifikatkette des Agentensystems definiert.

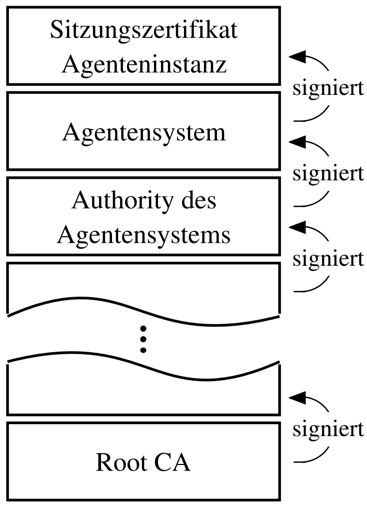

Abbildung 5.9: Zertifikatkette einer Agenteninstanz

Sitzungszertifikate für Agenteninstanzen

Wird eine Agenteninstanz auf einem Agentensystem erzeugt, muß ein neues Sitzungszertifikat für die Instanz erstellt werden:

- Das Agentensystem erzeugt ein neues Schlüsselpaar.
- Das Agentensystem erzeugt ein Zertifikat für die Agenteninstanz.
- Das Agentensystem signiert das Zertifikat.

Nun besitzt die Agenteninstanz ein neues Sitzungszertifikat, aus deren zugehöriger Zertifikatkette aus Abb. 5.9 sich die Identitäten

- der Agenteninstanz (Zertifikat 0 der Kette)
- des Agentensystems (Zertifikat 1 der Kette)
- der Authority des Agentensystems (Zertifikat 2 der Kette)

ersehen lassen.

Zertifikate für Implementierer und Anwender von Agenten

Ebenso wie Zertifikate für die Authority des Agentensystems, werden Zertifikate für Implementierer und Anwender von Agenten nicht von Agentensystemen oder Agenteninstanzen erzeugt, sondern werden in der Regel "out of band" von unabhängigen Instanzen generiert und ausgestellt.

Dies kann wiederum durch eine unabhängige Zertifizierungshierarchie geschehen. Während das Zertifikat der Authority des Agentensystems aber nicht von einem Teil des SmA selbst generiert werden kann, ist es für die Zertifikate von Implementierern und Anwendern möglich, eine Zertifizierungshierarchie zu benutzen, die als Dienst im SmA selbst verankert ist (vgl. Abschnitt 5.5.5).

5.5.5 Zertifikatinfrastruktur

Arbeitet man mit Zertifikaten, stellen sich unmittelbar weitere Fragen, die die Gültigkeit von Zertifikaten betreffen.

Gültigkeitsdauer

Zertifikate sollten, im Moment in dem sie ausgestellt werden, mit einer Gültigkeitsdauer versehen werden. Da weder für Agentensysteme, Agenteninstanzen, noch für Personen im allgemeinen Fall der tatsächliche Nutzungszeitraum zur Zeit der Zertifikaterstellung bekannt ist, ist hierfür ein Zeitraum zu wählen, der von den Sicherheitsrichtlinien der ausstellenden Entität bestimmt wird. Will man beispielsweise strenge Richtlinien durchsetzten, so wird man in der Regel vergleichsweise kurze Gültigkeitszeiträume wählen.

Läuft ein Zertifikat ab, benötigt aber der Eigentümer weiterhin ein Zertifikat, muß er von der ausstellenden Entität ein neues, bzw. verlängertes Zertifikat anfordern.

Kompromitierte Zertifikate

Für den Fall, daß ein Schlüsselpaar kompromitiert wurde, d.h. der private Schlüssel in "fremde" Hände gelangt ist, dürfen alle Zertifikate, die auf dem zugehörigen Schlüsselpaar, basieren nicht mehr verwendet werden, da diese nicht mehr geeignet sind ausschließlich die Identität des Schlüsselpaareigentümers zu belegen.

Somit stellt sich die Frage, wie ein solches Zertifikat im SmA widerrufen werden kann. Ein Widerruf durch eine nachträgliche Veränderung der im Zertifikat verzeichneten Gültigkeitsdauer ist ausgeschlossen, da diese selbst Teil des unveränderlichen Zertifikats ist. Somit muß eine Möglichkeit geschaffen werden, wie Listen von nicht mehr vertrauenswürdigen Zertifikaten im SmA verteilt werden. Solche Listen werden als *Certificate Revocation List* (CRL) bezeichnet.

Ansätze für die Verteilung von *CRLs* könnten beispielsweise sein:

- Verteilter Ansatz: Über Broadcast-Nachrichten werden *CRLs* verschickt
- Zentralistischer Ansatz: Ein spezieller Server hält *CRLs* vor und bietet Schnittstellen an um *CRLs* einsehen zu können.

Selbstverständlich sind auch Mischformen dieser Ansätze denkbar.

Weiterhin sind im Hinblick von Sicherheitseigenschaften die Frage zu klären, welche Entitäten überhaupt berechtigt sind, ein Zertifikat zu widerrufen, und welche Folgen dies für den Eigentümer des Zertifikats hat. Daß hierüber ebenfalls eine entsprechende Sicherheitsarchitektur zu formulieren ist, wird unmittelbar aus einem Beispiel klar: Durch den unberechtigten Widerruf eines Zertifikats, z.B. eines Agentensystems, kann sehr leicht eine DoS-Attacke durchgeführt werden, da mit dem Widerruf des Zertifikats das Agentensystem nicht mehr authentisiert werden kann.

Die Fragen wie CRLs sinnvoll verteilt werden, die Sicherheitsarchitektur von CRLs, sowie unter welchen Umständen neue Zertifikate erteilt werden, erfordern weitere Untersuchungen, die an dieser Stelle jedoch unterbleiben.

Zertifikatdienst

Neben der Veröffentlichung von CRLs stellt sich auch noch die Frage, inwieweit bereits abgelaufene, oder nicht mehr benutzte Zertifikate weiterhin verfügbar sein müssen.

Beispielsweise ist denkbar, daß zum Zweck des Logging die Sitzungszertifikate von Agenteninstanzen in einem zentralen Repository hinterlegt werden.

Dies führt unmittelbar zum Begriff eines Zertifikatservers, der diese Aufgaben übernehmen kann. Weiterhin könnten dort Schnittstellen zur Verifikation von Zertifikaten (inklusive eines Abgleichs mit den im Zertifikatserver ebenfalls verwalteten CRLs), und zur Erstellung von Personenzertifikaten angeboten werden.

Kombiniert man solche Zertifikatserver führt dies wiederum zu einer klassischen CA-Infrastruktur.

Eine genaue Betrachtung dessen, wie ein solcher Zertifikatserver aufgebaut sein muß, seiner Sicherheitseigenschaften und des Aufbaus von CA-Hierarchien oder der Möglichkeit der Kombination mit bestehenden CA-Infrastrukturen, wird an dieser Stelle nicht weiter durchgeführt.

5.5.6 Zusammenfassung der Authentisierungsmöglichkeiten

Mit den in diesem Abschnitt vorgestellten Verfahren lassen sich alle unmittelbar handelnden Entitäten (Principals) anhand eines Zertifikats authentisieren. Unter Einbeziehung der in den vorangegangenen Abschnitten gewonnenen Erkenntnisse ergibt sich zusammengefaßt das in Tabelle 5.1 dargestellte Bild. Zur Vereinfachung wurde für den Fall, daß Personen als Authentisierende auftreten, diese nicht detailliert aufgelistet, da es dann unerheblich ist, in welcher Rolle eine Person agiert.

	Authentisierender		
Authentisierter	Agenteninstanz	Agentensystem	Person
Agentengattung	+	+	+
Agenteninstanz	+	+	+
Agentensystem	+	+	+
Endsystem	-	-	-
Client-System	-	-	-
Webbrowser	-	-	-
Applet	-	-	+
Implementierer Agentengattung	+	+	+
Authority Agenteninstanz	+	+	+
Anwender Agenteninstanz	+	+	+
Authority Agentensystem	+	+	+
Anwender Agentensystem	+	+	+

Tabelle 5.1: Übersicht über Autorisierungsmöglichkeiten

Aus Tabelle 5.2 ist ersichtlich, welche Entitäten eigene Zertifikate besitzen und wie diese erhalten werden können. Daraus läßt sich ablesen, daß fundamentale Informationen, wie z.B. die Authority einer Agenteninstanz, nur indirekt über das Agentensystem gewonnen werden können, da diese nur als Attribute einer Klasse verfügbar sind. Entscheidend für die Sicherheit eines SmA ist also die Vertrauenswürdigkeit des Agentensystems.

Zwar kann beispielsweise das Zertifikat der Authority einer Agenteninstanz durch einen Anwender überprüft werden, indem der Anwender manuell in der Zertifikatkette nach für ihn vertrauenswürdigen Unterzeichnern sucht. Dabei muß er aber die Gewißheit haben, daß das Agentensystem das zu untersuchende Zertifikat korrekt übermittelt hat.

Entität	eigenes Zertifikat	gespeichert als
Agentengattung	dig. signiert	Teil der Daten
Agenteninstanz	+	eigenes Attribut der Agenteninstanz
Agentensystem	+	eigenes Attribut des Agentensystems
Endsystem	-	-
Client-System	-	-
Webbrowser	-	-
Applet	dig. signiert	Teil der Daten
Implementierer Agentengattung	+	aus dig. Signatur der Agentengattung
Authority Agenteninstanz	+	aus Attribut der Agenteninstanz
Anwender Agenteninstanz	+	wird direkt vom Anwender übermittelt
Authority Agentensystem	+	aus Zertifikatkette AS
Anwender Agentensystem	+	wird direkt vom Anwender übermittelt

Tabelle 5.2: Übersicht über Eigentümer eigener Zertifikate

5.6 Der Code einer Agentengattung

Für die Realisierung eines sicheren SmA können grundsätzlich zwei verschiedene Vertrauensbeziehungen zu Agenten unterschieden werden:

- Der Code eines Agenten, die Agentengattung, ist vertrauenswürdig. D.h. man vertraut dem Implementierer des Codes, daß die für die Gattung angegebene Semantik korrekt implementiert wurde und z.B. kein Trojanisches Pferd enthält.
- Die Daten einer Agenteninstanz sind vertrauenswürdig. D.h. die aktuelle Belegung aller Attribute einer Agenteninstanz ist insofern korrekt, daß unter Verwendung von vertrauenswürdigem Code alle Aktionen eines Agenten sich im Rahmen der Semantik des Agenten bewegen.

Entsprechend dieser Unterscheidung muß in einem Sicherheitsmodell die Behandlung von Agentengattung und Agenteninstanz getrennt betrachtet werden.

Am Beispiel des folgenden Szenarios läßt sich die Notwendigkeit dieser Unterscheidung unmittelbar erkennen: Ein Implementierer (Hersteller) I erstellt eine Agenteninstanz. Dann wird diese von zwei völlig verschiedenen Anwendern (aus unterschiedlichen Firmen) eingesetzt. Nun ist es vorstellbar, daß, obwohl I prinzipiell vertrauenswürdig ist, man der Authority (der eine Anwender) der einen Instanz vertraut (weil dieser z.B. aus der gleicher Firma ist), man jedoch der Authority (der andere Anwender) der anderen Instanz nicht vertrauen kann (da dieser z.B. für einen Konkurrenten tätig ist).

Aber auch für den Aufbau einer Vertrauensbeziehung zu einer Agentengattung ist die Trennung sinnvoll. So kann beispielsweise Code, der von einem Ort geladen wird, der als sicher vor Manipulationen angesehen werden kann, leichter als vertrauenswürdig eingestuft werden, als solcher der von einer unbekannten Quelle stammt.

Für die persistente Speicherung von Agenteninstanzen sollten demnach eigens dafür vorgesehene Datenbasen verwendet werden. Eine solche Datenbasis wird im Folgenden als *Code Repository* bezeichnet.

Wird nun eine neue Agenteninstanz erzeugt oder migriert eine bestehende Agenteninstanz, so stellt sich die Frage, wie ein Agentensystem an den zur Ausführung des Agenten notwendigen Code gelangt.

Für den Fall der Erzeugung einer neuen Instanz macht die MASIF-Spezifikation hierzu keine näheren Angaben, während für eine Migration durchwegs von einem Modell ausgegangen wird, in welchem Code, der zur Ausführung einer Agenteninstanz benötigt wird, früher oder später vom zuletzt besuchten Quellagentensystem der Migration geladen wird. Die Möglichkeit, daß Agentensysteme eigene Datenbasen für Code besitzen wird nicht betrachtet.

Kombiniert man nun Code Repositories mit dem in MASIF vorgestellten Konzept, ergibt sich ein mehrstufiges Verfahren zum Laden von Code:

- Der Code wird aus einem Code Repository geladen. Dabei kann ein Agentensystem eine Liste vertrauenswürdiger Code Repositories besitzen und diese nacheinander abfragen.
- Schlägt der erste Schritt fehl, so wird der Code von dem zuletzt besuchten Quellagentensystem geladen.

Die Vorteile dieses kombinierten Konzepts bestehen darin, daß

- Agentengattung und Agenteninstanz unabhängig voneinander authentisiert werden können.
- das Vertrauen zu Agentengattungen in Abhängigkeit des Speicherorts der Gattungsdaten formuliert werden kann.
- die Konformität zum MASIF-Standard gewahrt bleibt.

5.7 Sicherung der Daten einer Agenteninstanz

Die Mobilität von Agenteninstanzen wirft ein weiteres Problem auf. Da jegliche Daten einer Agenteninstanz der Gefahr der illegalen Veränderung ausgesetzt sind, müssen Maßnahmen ergriffen werden, welche die Authentizität dieser Daten sicherstellen können. Dabei meint "sicherstellen der Authentizität" nicht, daß die Unveränderlichkeit der Daten garantiert werden kann, sondern daß Veränderungen zuverlässig erkannt werden können.

Dabei genügt es nicht, daß sichere Kanäle zur Kommunikation verwendet werden, da dies nur ein Veränderung durch außenstehende Entitäten ausschließt. Migriert eine Agenteninstanz jedoch auf ein Agentensystem mit feindlichen Absichten, sind ihre Daten immer noch einer ungewollten Veränderung preisgegeben.

Daten einer Agenteninstanz lassen sich in zwei grobe Kategorien unterscheiden:

1. Daten, die eine (ständige) Veränderung bis zur Terminierung der Agenteninstanz erfahren.
2. Daten, die einmalig gesetzt werden und dann bis zur Terminierung der Agenteninstanz unveränderlich bleiben sollen.

Die Authentizität von Daten der ersten Kategorie ist äußerst schwierig zuzusichern, da prinzipiell die Rechenvorschrift bekannt sein müßte, die diese Daten erzeugt haben. Durch Nachvollziehung dieser Berechnungen ließe sich dann die Authentizität der Daten überprüfen. Eine nähere Betrachtung dieser Problematik wäre an dieser Stelle zu umfangreich und muß deshalb unterbleiben. Ein Lösungsansatz findet sich in [NeLe 98].

Die zweite Kategorie von Daten läßt sich aber mit vergleichsweise einfachen Mitteln gegen Veränderungen sichern. Dabei können die Daten durch die Agenteninstanz selbst oder aber auch durch ein Agentensystem oder eine Person erzeugt werden. In den folgenden Abschnitten werden die unveränderlichen Daten weiter kategorisiert und Sicherungsmaßnahmen vorgestellt.

5.7.1 Private, unveränderliche Daten

Daten, die einmalig gesetzt und später nur von einem bestimmten "Empfänger" gelesen werden sollen, lassen sich in den meisten Fällen sichern.

Hierzu werden die zu sichernden Daten sofort nach ihrer Erzeugung mit dem öffentlichen Schlüssel des Empfängers verschlüsselt, was sicherstellt, daß nur der Empfänger die Daten lesen kann.

Dieses Verfahren ist im Fall von Agentensystemen oder Personen als Empfänger immer anwendbar. Einzig im Fall von Agenteninstanzen als Empfänger kann es nur solange angewendet werden, wie die empfangende Instanz auf jenem Agentensystem verharrt, zu dem sie sich zum Zeitpunkt der Verschlüsselung befand. Der Grund hierfür ist die Tatsache, daß der private Schlüssel der Empfänger-Agenteninstanz nur für die Dauer einer Sitzung Bestand hat, nach einer Migration jedoch nicht mehr verfügbar ist, und somit die Empfänger-Agenteninstanz die Daten nicht mehr entschlüsseln kann.

Um nun auch die Authentizität der Daten gewährleisten zu können, versieht der Erzeuger der Daten diese mittels seines privaten Schlüssels mit einer digitalen Signatur. Wenn ein Agentensystem oder eine Person diese Daten erzeugt hat, ist dieser Vorgang problemlos, da sie über eigene persistente Schlüsselpaare verfügen.

Im Fall von Agenteninstanzen als Erzeuger tritt jedoch wiederum das Problem auf, daß Agenteninstanzen keine persistenten Schlüsselpaare besitzen. Hierzu bieten sich zwei Lösungsmöglichkeiten an:

1. Stellvertretende Signatur:

   Äquivalent zum Verfahren zur Authentisierung der Agenteninstanzen kann aber hierfür erneut das Agentensystem herangezogen werden, welches dann stellvertretend für die Instanz die Daten signiert. Bei einer späteren Überprüfung der Signatur muß dann wiederum zuerst über die Vertrauenswürdigkeit des signierenden Agentensystems entschieden werden, bevor nach einer Verifikation der Signatur, die Daten als authentisch angesehen werden können. Dabei tritt dann die Agenteninstanz nur noch durch ihren Identifikator in Erscheinung, da zur Signierung der Schlüssel des Agentensystems benutzt wurde.

2. Eigene Signatur:

   Die Agenteninstanz benutzt den zu ihrem Sitzungszertifikat gehörenden privaten Schlüssel um die Daten zu signieren. Voraussetzung für dieses Verfahren ist jedoch die Existenz eines Zertifikatdienstes nach Abschnitt 5.5.5, damit der Empfänger der

Daten später das mittlerweile unter Umständen nicht mehr aktuelle Sitzungszertifikat der Agenteninstanz überprüfen kann.

Die erste Lösung hat den Vorteil, daß sie sich auch ohne Zertifikatdienst realisieren läßt, benötigt aber eine weitere Stufe in der Vertrauenskette. Sie sollte also nur umgesetzt werden, wenn keine Zertifikatinfrastruktur vorhanden ist, ansonsten sollte der zweiten Lösung der Vorzug gegeben werden.

5.7.2 Öffentliche, unveränderliche Daten

Sollen Daten, die eine Agenteninstanz mit sich führt, zwar gegen Veränderungen geschützt sein, die Daten selbst aber beliebig lesbar bleiben, findet das gleiche Verfahren Anwendung wie im vorigen Abschnitt beschrieben, einzig der Schritt der Verschlüsselung unterbleibt dann. Damit bleiben die Daten selbst unverschlüsselt, sind aber mit einer überprüfbaren digitalen Signatur versehen.

5.7.3 Listen von elementweisen unveränderlichen Daten

Ein spezieller Fall von unveränderlichen Daten stellen Listen dar, deren Einträge jeweils unveränderlich sein sollen, die Liste selbst aber während des Lebenszyklus der Agenteninstanz sukzessive erweitert werden können soll.

Korjoth, Asokan, Gülcü geben in [KAG 98] hierfür Protokolle an, mit deren Hilfe, folgende Eigenschaften einer solchen Liste zugesichert werden können:

1. Listeneinträge können nur von einem gemeinsamen Empfänger der Daten gelesen werden.
2. Authentizität der einzelnen Listeneinträge
3. Eindeutige, unveränderliche Zuordnung des Erzeugers zu einem Listeneintrag
4. Fortschreibbare Integrität der gesamten Liste.
5. Öffentliche Verifizierbarkeit der Listenintegrität.
6. Schutz vor nachträglichem Einfügen von Elementen.
7. Schutz vor nachträglichem Entfernen von Elementen.

In [KAG 98] werden hierfür insgesamt vier Protokolle (genannt P1 bis P4) angegeben, von denen zwei eine vorhandene Public-Key-Infrastruktur zur Voraussetzung haben, und sich damit hervorragend für die Verwendung im Rahmen dieses Konzepts eignen. Neben den oben beschriebenen Eigenschaften, die sie gemeinsam realisieren, unterscheiden sie sich in folgenden Punkten

- KAG-P1: Die Identität des Erzeugers eines Listeneintrags kann öffentlich gelesen werden.
- KAG-P2: Die Identität des Erzeugers eines Listeneintrags kann nur vom Empfänger der Daten gelesen werden.

Zur Zusicherung der Eigenschaften 1 und 2 werden dabei die Signier- und Verschlüsselungsmechanismen der Public-Key-Verfahren benutzt, für die Eigenschaften 3 bis 7 werden Hash-Funktionen benutzt.

Damit ist es möglich, vom Beginn der Liste an, Schritt für Schritt die Integrität der Liste zu überprüfen. Kann dabei ein Schritt nicht erfolgreich überprüft werden, so ist zumindest die Integrität des bis dahin überprüften Teils der Liste noch gewährleistet.

Protokoll KAG-P1a

Für Listen, deren Einträge öffentlich lesbar sein sollen, wird kein explizites Protokoll angegeben. Durch eine kleine Modifikation von KAG-P1 lassen sich jedoch auch solche Listen realisieren.

Betrachtet man aus [KAG 98] den Verschlüsselungs-/Signierschritt von KAG-P1

$$O_i = \mathcal{SIG}_i(\mathcal{ENC}_0(o_i, r_i), h_i)$$

Listenindex	i
Listeneintrag	O_i
Erzeuger des i-ten Eintrags	S_i
Signaturfunktion des Erzeugers	$\mathcal{SIG}_i$
Verschlüsselungsfunktion für den Empfänger	$\mathcal{ENC}_0$
Wert des Listeneintrags	o_i
Zufallswert	r_i
Hash-Funktion	$\mathcal{H}$
Hash-Vorschrift	$h_i = \mathcal{H}(O_{i-1}, S_{i+1})$
Fortschreibungsvorschrift	$S_i -> S_{i+1} : \{O_k \vert 0 <= k <= i\}$

läßt sich durch Entfernen der Verschlüsselung $\mathcal{ENC}_0$ der modifizierte Schritt

$$O_i' = \mathcal{SIG}_i(o_i, r_i, h_i)$$

gewinnen. Unter Beibehaltung der Hash- und Fortschreibungsvorschrift erhält man damit das neue Protokoll KAG-P1a, welches alle Eigenschaften von KAG-P1 umfaßt, jedoch die Listeneinträge öffentlich lesbar bleiben.

Der Beweis über die Korrektheit und die Eigenschaftszusicherung erfolgt dabei äquivalent wie in [KAG 98] für das KAG-P1 ausgeführt und soll hier nicht wiederholt werden.

5.7.4 Die Attribute einer Agenteninstanz

Um nun die Authentizität der in den vorangegangenen Abschnitten definierten Attribute einer Agenteninstanz sicherstellen zu können, müssen nur noch die eben definierten Verfahren auf die einzelnen Attribute angewendet werden:

Die Historie der besuchten Agentensysteme kann mittels KAG-P1a gesichert werden. Authority, Implementierer und Agentengattung sind typische Vertreter von Daten, die einmalig gesetzt werden, und zwar bei der Erzeugung der Agenteninstanz, und deren Unveränderlichkeit für den Rest des Lebenszyklus der Instanz sichergestellt werden muß. Dies kann nach Abschnitt 5.7.2 erfolgen, indem das Agentensystem, auf dem die Instanz erzeugt wird, diese Werte mit einer Signatur versieht. Dabei kann die Identität des Agentensystems, bei einer späteren Überprüfung der Signatur, aus dem ersten Eintrag der Historie gewonnen werden.

Damit ist die Authentizität aller Attribute einer Agenteninstanz auf eine Signatur durch das erzeugende Agentensystem zurückgeführt. Somit ist es einem Angreifer nicht mehr möglich Attribute zu verändern. Ebenfalls geschützt ist die Historie der Agentensysteme. Durch KAG-P1a ist es einem feindliches Agentensystem nicht möglich die Historie partiell zu verändern. Es kann auch nicht unterlassen, sich selbst in die Liste einzutragen. Dies würde nämlich auf dem nächsten Agentensystem, auf das die Instanz migriert, bemerkt werden, da die Integrität der Liste dann nicht mehr intakt wäre.

Unter Benutzung der oben beschriebenen Verfahren ist es selbstverständlich auch möglich und angebracht, die für eine Agenteninstanz individuellen Attribute und Daten zu sichern, solange diese keinen nachträglichen Veränderungen unterworfen sind.

Beispielsweise könnte ein Management-Agent, der von seiner Authority beauftragt wurde QoS-Daten auf konkurrierenden Endsystemen zu messen, die auf jedem Endsystem anfallenden Daten einzeln nach Abschnitt 5.7.1 mit dem öffentlichen Schlüssel seiner Authority chiffrieren und dann selbst signieren. Damit sind die Meßwerte vor Ausspähen durch andere Endsysteme ("Wie gut ist die Konkurrenz?") und vor Veränderungen geschützt.

5.8 Autorisierung

Nachdem mit den vorangehenden Kapiteln die Authentisierung einer handelnden Entität ermöglicht wird, ist die Grundvoraussetzung für eine Autorisierung von Aktionen gegeben.

5.8.1 Die zu autorisierenden Schnittstellen

Nachdem bereits in Kap. 3.4 die für die Autorisierung relevanten Schnittstellen vorgestellt wurden, soll nun detaillierter betrachtet werden, wo und wie die Autorisierung für diese Schnittstellen durchgeführt werden kann.

Betrachtet man die verschieden Ansätze aus Kap. 3.6, so kommt man zu dem Schluß, daß für eine Sicherheitsarchitektur, welche die gestellten Anforderungen erfüllt, eine Kombination der vorgestellten Ansätze notwendig ist.

Für eine grobe Trennung zwischen Ressourcen, die durch das Agentensystem generell erreichbar sein sollen und jenen, die niemals erreichbar sein sollen, eignen sich Sicherungsmaßnahmen im Endsystem. Wegen der Plattformabhängigkeit und schlechten Managebarkeit darf die Trennung dabei nur so vorgenommen werden, daß diese über einen langen Zeitraum Bestand hat und vor allem keine Änderungen dieser Sicherungsmaßnahmen zur Laufzeit des Agentensystems vorgenommen werden müssen. Zusicherungen, die mit "immer" oder "niemals" attributiert sind, können so umgesetzt werden.

Beispielsweise könnte eine solche Trennung Teile des lokalen Dateisystems vom Zugriff des Agentensystems ausschließen oder den Zugriff auf ein bestimmtes Netz-Interface sperren, auf dessen Segment niemals zugegriffen werden darf.

Die fallweise Autorisierung der Schnittstellen des Endsystems muß demnach eine Ebene höher, also durch das Agentensystem vorgenommen werden, denn zum einen kann das Agentensystem die notwendige Unterscheidung verschiedener Entitäten vornehmen. Zum anderen entsteht hierdurch eine zentrale, im gesamten SmA plattformunabhängige Komponente, die eine vereinheitlichte Managementschnittstelle bereitstellt, was die Voraussetzung für die Erfüllung der Forderung nach einfacher Managebarkeit ist. Daneben ist das Agentensystem auch für die Autorisierung seiner eigenen Schnittstelle verantwortlich.

Aber auch der Zugriff auf Schnittstellen einer Agenteninstanz müssen durch das Agentensystem autorisierbar sein. Dies ermöglicht, daß Agenteninstanzen die eigene Schnittstellen anbieten und dafür keine eigenen Autorisierungsmaßnahmen ergreifen, aber kritische Aktionen auf dem Agentensystem oder Endsystem ausführen, trotzdem ausgeführt werden können. Dies wird an einem Beispiel deutlich:

Eine Agenteninstanz E sei vertrauenswürdig und könne deshalb eine kritische Aktion A auf dem Endsystem ausführen. Bietet E aber auch eine eigene Schnittstelle zum Auslösen von A an und autorisiert deren Nutzung nicht, so könnte ein Angreifer diese Schnittstelle nutzen, um A unberechtigterweise auf dem Endsystem auszuführen, da aus der Sicht des Agentensystems A vom vertrauenswürdigen E ausgeführt wird. Kann aber das Agentensystem Zugriffe auf die Schnittstellen von E autorisieren, kann die unberechtigte Nutzung dieser Schnittstelle ohne Zutun der Agenteninstanz verhindert werden.

Eine extrem feingranulare, individuelle Überprüfung von Aktionen kann nur durch die jeweilige Entität selbst erfolgen, die die Schnittstelle für die Aktion bereitstellt. So sind beispielsweise Agenteninstanzen für die Überprüfung der Parameter bei Benutzung einer Schnittstelle selbst verantwortlich, da nur sie die genaue Semantik der Funktion und damit die in der Signatur verzeichneten Parameter kennen. Gleiches gilt für die Schnittstellen, die das Agentensystem selbst anbietet.

Zusammenfassend zeigt Tabelle 5.3 die Beziehungen zwischen Schnittstellen und den für deren Autorisierung zuständigen Entitäten

	Autorisierende Entität		
Schnittstelle	Endsystem	Agentensystem	Agenteninstanz
Endsystem, nie erreichbar	X		
Endsystem, prinzipiell erreichbar		X	
Agentensystem		X	
Agenteninstanz		X	X

Tabelle 5.3: Übersicht Schnittstellen und deren Autorisierung

5.8.2 Entscheidungshierarchie

Nach vorhergehendem Absatz tritt ein weiteres Problem in Erscheinung. Die Schnittstelle einer Agenteninstanz wird danach sowohl durch das Agentensystem als auch durch die Agenteninstanz selbst autorisiert. Damit sind Kollisionen in den Entscheidungen nicht auszuschließen.

Die einfachste Art solche Konflikte zu lösen ergibt sich durch eine einfache Priorisierung der Entscheidungsträger. So gelangt man zu einer Entscheidungshierarchie.

Da das Agentensystem nicht nur für seine eigene Sicherheit verantwortlich zeichnet, sondern auch für andere Agenteninstanzen und insbesondere das Endsystem, genießen "Negativ"-Entscheidungen des Agentensystems eine höhere Priorität als Entscheidungen von Agenteninstanzen. D.h. wenn eine Agenteninstanz zwar die Benutzung einer eigenen Schnittstelle genehmigt, kann das Agentensystem dies immernoch ablehnen.

Im umgekehrten Fall kann eine Ablehnung durch die Agenteninstanz nicht durch das Agentensystem "aufgehoben" werden.

5.8.3 Formulierung von Entscheidungsregeln: Policies

Die zentrale Frage der Autorisierung ist

"Darf eine Entität E eine Aktion A durchführen?"

Damit diese Frage fallweise ohne Eingriff einer Person beantwortet werden kann müssen Regeln formuliert werden, nach denen diese Entscheidungen zu treffen sind. Ein Satz solcher Regeln wird im folgenden Policy genannt.

Adressierung von Entitäten in Policies

In einer solchen Policy muß es zunächst also die Möglichkeit geben die Entität E zu adressieren. Nun ist es allerdings nicht praktikabel, diese Adressierung allein über den Identifikator von E zu bewerkstelligen, da die Menge der möglichen Entitäten, und damit die konkreten Identifikatoren, in einem SmA weder endlich noch von vorneherein aufzählbar sind. Dies hat zur Folge, daß sich hierauf basierend keine allgemeinen Regeln formulieren lassen.

Vielmehr ist es notwendig weitere charakteristische Parameter einer Entität zu Rate zu ziehen, da nur solche Parameter vor der tatsächlichen Anfrage bekannt sind.

In Abschnitt 5.5.1 sind diese charakteristischen Parameter in Form von Attributen der *Principals* bereits definiert worden und können nun für die Formulierung von Regeln benutzt werden.

Daraus abgeleitet lassen sich die folgenden Überprüfungen anstellen:

Sei E ein *Principal*

- Ist E vom Typ Person/Agenteninstanz/Agentensystem?
- Ist E von X signiert?

Falls E ein Agentensystem ist:

- Ist die Authority von E gleich Z?
- Ist der Implementierer von E gleich Z?

Falls E eine Agenteninstanz ist:

- Hat E eines der Agentensysteme aus der Menge M besucht?
- Ist die Gattung von E gleich Y?
- Ist die Authority von E gleich Z?
- Ist der Implementierer von E gleich Z?

Da die Attribute Authority, Implementierer, Agentensystem selbst wieder vom Typ *Principal* sind, lassen sich auch rekursiv fortgesetzte Bedingungen über die Attribute formulieren:

- Hat E ein Agentensystem besucht, dessen Authority von Z signiert wurde?
- Ist die Authority von E eine Person?

Weiterhin ist die boolsche Verknüpfung solcher Bedingungen wünschenswert, um komplexere Regeln formulieren zu können:

- (Ist E von Typ Agenteninstanz ODER Ist E von Typ Person) UND wurde die Authority von E von X signiert?

Eine weitere Flexibilisierung in der Formulierung von Regeln wird dann erreicht, wenn Attribute des Eigentümers einer Policy über symbolische Konstanten für die Vergleichsoperationen zu Verfügung gestellt werden. Sei MEINE_AUTHORITY nun die symbolische Konstante für den Eigentümer der Entität, die die Policy auswertet, dann könnte eine Regel lauten:

- Ist E vom Typ Agentensystem und ist die Authority von E gleich MEINE_AUTHORITY?

Nachdem jetzt die Bedingungen formuliert werden können ist es möglich anhand dieser Bedingungen Erlaubnisse für die Benutzung einer Schnittstelle zu erteilen, indem man in Abhängigkeit von Bedingungen Aktionen erlaubt:

- Wenn E vom Typ Agentensystem ist UND die Authority von E gleich MEINE_AUTHORITY, dann erlaube alle Aktionen.
- Wenn E von der Gattung G ist und die Authority von E gleich MEINE_AUTHORITY, dann erlaube Aktion A.

Die hier informell vorgestellten Überprüfungen markieren das Minimum dessen, das benötigt wird, um Regeln formulieren zu können, die gleichzeitig

- unterschiedliche Typen von Entitäten behandeln,
- unterschiedliche Eigentumsverhältnisse von Entitäten reflektieren,
- die Historie betrachten,
- Charakteristika der Entität miteinbeziehen, die die Regeln auswertet.

Eine Einschränkung des Umfangs der Formulierungsmöglichkeiten ist zwar denkbar, bedeutet aber immer, daß Teilaspekte sicherheitsrelevanter Informationen nicht überprüft werden können oder die Flexibilität in der Formulierung von Regeln eingeschränkt wird.

Adressierung von Aktionen in Policies

Mit den bis hierhin vorgestellten Mitteln lassen sich die Entitäten flexibel adressieren. Gleiches gilt bis dato aber nicht für Aktionen, diese können nur nach ihrer Art benannt werden. Häufig genügt es aber nicht nur die Aktion ihrem Typus nach zu betrachten, auch die konkrete Belegung der Parameter einer Aktion können die Entscheidung über die Durchführung der Aktion beeinflussen.

Beispielsweise sei eine Aktion `ErzeugeAgenteninstanz` mit dem Parameter Agentengattung betrachtet. Soll nun nur die Erzeugung von Instanzen bestimmter Agentengattungen gestattet werden, so besteht keine Möglichkeit dies zu formulieren.

Betrachtet man parametrisierte Aktionen näher, so zerfallen diese in zwei Kategorien:

- Die Menge aller Parameterbelegungen ist endlich und aufzählbar.
- Die Menge aller Parameterbelegungen ist nicht endlich und aufzählbar, bzw. sie ist zwar endlich, aber sehr groß.

Die erste Kategorie läßt sich auf Aktionen ohne Parameter abbilden, indem man für jede Parameterbelegung eine eigene Aktion definiert.

Beispiel: Eine Aktion `Dateizugriff` mit dem Parameter Zugriffsart und dessen möglichen Belegungen `Lesen`, `Schreiben`, `LesenUndSchreiben` kann auf die Aktionen `DateiLesen`, `DateiSchreiben` und `DateiLesenSchreiben` abgebildet werden.

Ist die Menge der Parameterbelegungen aber nicht endlich[2], werden wiederum Entscheidungsregeln benötigt, die es ermöglichen eine konkrete Parameterbelegung zu analysieren.

Beispiel: Eine Aktion `SystemkommandoAusführen` mit dem Parameter `Kommando` vom Typ `String`, kann nicht in einzelne diskrete Aktionen zerlegt werden, da die Menge der möglichen Belegungen von `Kommando` praktisch nicht aufzählbar ist.

Entscheidungsregeln für Parameterbelegungen für den allgemeinen Fall, im voraus, auch nur informell, zu spezifizieren, ist aber nicht möglich, da hierfür die Semantik der einzelnen Parameter bekannt sein müßte. Eine Spezifikation solcher Entscheidungsregeln muß demnach immer individuell für den Einzelfall erstellt werden.

Realisierung von Policies

Faßt man nun die Erkenntnisse über die Adressierung von Aktionen in Policies mit jenen über die in Abschnitt 5.8.1 gemachte Analyse über eine sinnvolle Durchführung der Autorisierung zusammen, ergibt sich folgendes Bild:

In Agentensystemen ist eine sinnvolle Autorisierung von Aktionen nur im Folgenden Umfang möglich:

- Die Autorisierung der Schnittstellen von Agenten kann höchstens bis zur Ebene der an den Schnittstellen angebotenen Methoden erfolgen, ohne aber die Parameter der Methoden zu betrachten. Dies kann durch generische Policies erfolgen.
- Eigene Methoden des Agentensystems können individuell, inklusive Betrachtung der einzelnen Parameter behandelt werden.
- Zugriffe auf Schnittstellen des Endsystems können ebenfalls individuell im Agentensystem behandelt werden.

Im Rahmen der Entscheidungshierarchie kann dann durch Agenteninstanzen eine weitere, individuelle Autorisierung der eigenen Methoden erfolgen.

[2]Tatsächlich sind in realen Systemen die Mengen der Belegungen immer endlich, da nur eine begrenzte Menge physikalischen Speichers zu Verfügung steht, aber schon 1 kByte kann ca. $2*10^5$ verschiedene Belegungen annehmen

Generische Policies können also nur Aktionen behandeln, die entweder keine Parameter besitzen oder die Betrachtung der Parameter außen vorlassen.

Um eine solche Policy nun konkret formulieren zu können und damit auch eine plattformunabhängige Instanz zum Management von Policies zu schaffen, ist die Schaffung einer formale Sprache denkbar, die folgende Eigenschaften besitzt:

- Adressierung von Attributen der Entitättypen Agentensystem, Agenteninstanz und Person
- Vergleichsoperationen über Attribute
- Wenn / dann Konstrukte
- Boolsche Ausdrücke
- Symbolisch Konstanten
- Adressierung generischer Aktionen

Um auch die individuellen Semantiken der einzelnen Schnittstellen behandeln zu können, d.h. konkret Methoden inklusive ihrer Parameter zu betrachten, sollte die Policy-Sprache modular erweiterbar sein.

Die konkrete Spezifikation einer solchen Policy-Sprache, die all diese Eigenschaften unterstützt, muß im Rahmen dieser Arbeit unterbleiben und könnte Gegenstand einer weiteren Arbeit werden.

Konkrete Policies

Neben einer Policy für die Benutzung der einzelnen Schnittstellen werden aber weitere Entscheidungen notwendig, die in eigene Policies formuliert werden können:

- Von welchem Code Repository ist ein Agentengattung zu laden?
- Welche Zertifikatketten können als vertrauenswürdig angesehen werden?

Somit seien nun die folgenden Policies konkret definiert:

- Code Repository Policy: Entscheidet, welches zur Verfügung stehende Code Repository benutzt werden soll.
- Authentication Policy: Entscheidet, ob eine Zertifikatkette als vertrauenswürdig angesehen werden kann.
- Permission Policy: Entscheidet über den Zugriff auf Schnittstellen

Alle drei genannten Policy-Typen müssen durch ein Agentensystem implementiert werden, Authentication Policy und Permission Policy können optional auch zusätzlich durch Agenteninstanzen erweitert werden. Dann wird die Priorisierung der Entscheindungen nach Abschnitt 5.8.2 vorgenommen.

Die Permission Policy bestimmt dabei im Fall eines Agentensystems über die Erlaubnis zur Benutzung der eigenen Methode, der Methode der Schnittstelle zum Endsystem und über den Zugriff auf Methoden eines Agenten. Die Permission Policy einer Agenteninstanz dagegen ist nur für eigene Methoden zuständig, die an ihren Schnittstellen angeboten werden.

5.8.4 Der Entscheidungsprozeß

Die Eingangs im vorangegangenen Kapitel gestellte Frage

"Darf eine Entität E eine Aktion A durchführen?"

kann jetzt mit den vorgestellten Mitteln vollständig beantwortet werden.

Wird von einer Entität E eine Anfrage zur Durchführung einer Aktion A an eine Entität F gestellt, so authentisiert F die anfragende Entität mittels der Authorization Policy und den in Abschnitt 5.5.3 geschilderten Verfahren. Wenn E authentisiert werden konnte, dann überprüft F anhand der Permission Policy, ob die Aktion ausgeführt werden darf.

Wird von der Permission Policy die Ausführung der Aktion genehmigt, so wird die Aktion ausgeführt. Kann die anfragende Entität nicht authentisiert werden oder stellt die Permission Policy einen negativen Bescheid aus, so wird die Aktion nicht durchgeführt. Außerdem wird die Entität, die die Aktion ausführen wollte, darüber informiert, daß die Aktion nicht ausgeführt wurde. Ob und wie ausführlich eine Begründung für die Ablehnung beigefügt wird, ist davon abhängig, inwieweit man einem potentiellen Angreifer Hinweise für weitere Angriffe geben möchte, bzw. hängt auch mit der Benutzerfreundlichkeit des Systems ab; näheres hierzu soll an dieser Stelle nicht betrachtet werden.

Wird die Ausführung einer Aktion nicht gestattet, sind noch weitere Sanktionen denkbar. Begeht z.B. eine lokale Agenteninstanz einen "schwerwiegenden" Sicherheitsverstoß, so wäre beispielsweise denkbar, daß dieser terminiert wird. Die Art der Sanktionen kann optional wiederum durch eine Policy bestimmt werden.

5.9 Delegation

Mit Hilfe einer Policy ist nun die fallweise Erteilung eines Rechts für die Ausführung einer Aktion möglich. Für die in 2.5 geforderte Möglichkeit der Delegation, der Weitergabe von Rechten, ist es notwendig, diese persistent formulieren zu können. Dabei genügt es nicht, einfach nur das Recht zu betrachten, da dieses keine Beziehung zur berechtigten Entität besitzt.

Beispiel: Habe eine Agentengattung A gemäß der Permission Policy eines Agentensystems S das Recht Lesen der Datei F. Eine Agentengattung B solle dieses Recht gemäß der Permission Policy nicht haben.

Um einer Instanz von B dieses Recht für die Durchführung einer Teilaufgabe im Auftrag von einer Instanz A erteilen zu können, wäre ein naiver Ansatz jener, daß jede Agenteninstanz eine Liste von Rechten mit sich führt.

Trage eine Instanz von B also das Recht zum Lesen von F mit sich. Nur wie kann dann entschieden werden, ob B durch A ermächtigt wurde F zu lesen, oder B sich dieses Recht illegalerweise selbst eingeräumt hat?

5.9.1 Persistenz von Rechten: Capabilities

Anhand des Beispiels läßt sich erkennen, daß an ein Recht unmittelbar zwei weitere Attribute geknüpft sind:

- Der Eigentümer des Rechts.
- Der Aussteller des Rechts, nämlich die Instanz, die das Recht erteilt hat.

Dieses 3-Tupel aus Recht, Eigentümer und Aussteller soll fortwährend als Capability bezeichnet werden.

Um nun die Authentizität einer Capability sicherstellen zu können bietet sich wiederum das Verfahren der digitalen Signatur an: Das 2-Tupel bestehend aus Recht und Eigentümer wird vom Aussteller signiert. Damit kann eine beliebige Instanz die Authentizität der Capability überprüfen, indem es die Signatur über Recht und Eigentümer überprüft. Weiterhin tritt mit der Signatur der Aussteller als Bürge für die "Rechtmäßigkeit" des Besitzes des genannten Rechts auf.

Mit den Capabilities wurde somit eine Möglichkeit geschaffen, daß eine Agenteninstanz eine Liste von Rechten mit sich trägt und überprüft werden kann, ob die Agenteninstanz tatsächlich über die in der Liste verzeichneten Rechte verfügt.

Die Rechte, die durch eine Permission Policy erteilt werden lassen sich nun ebenfalls als Capability formulieren: Wird für eine konkrete Entität gemäß einer Permission Policy ein Recht eingeräumt, so stellt dies eine implizite Capability dar, mit der konkreten Entität als Eigentümer und der autorisierenden Entität, die die Permission Policy auswertet, als Aussteller. Damit lassen sich also Permission Policies benutzen, um konkrete Capabilities zu erstellen.

5.9.2 Autorisierung mittels Capabilities

Für die Autorisierung einer Aktion können nun neben den sich aus der Permission Policy ergebenden Rechten auch die von der Agenteninstanz mit sich getragenen, expliziten Capabilities herangezogen werden.

Wird nämlich ein benötigtes Recht nicht durch die Permission Policy erteilt, so könnte eine Agenteninstanz immer noch eine Capability des entsprechenden Rechts vorweisen.

Um nun die Autorisierung vornehmen zu können, müssen die folgenden Fragen geklärt werden:

1. Ist die Capability authentisch?
2. Ist die Agenteninstanz der Eigentümer der Capability?
3. Ist der Aussteller der Capability vertrauenswürdig?

Die Fragen 1 und 2 lassen sich nach dem vorangegangenen Abschnitt eindeutig klären.

Frage 3 ist nun wiederum eine individuelle Entscheidung, die eine autorisierende Instanz über die eigene Permission Policy klären kann, beispielsweise indem diese Regeln enthält, die bestimmte Entitäten als vertrauenswürdige Bürgen für Rechte bzw. Aussteller von Capabilities markiert.

Somit wird es möglich, daß eine autorisierende Entität Aktionen genehmigt, ohne daß sie selbst vorher explizit Kenntnis der hierfür benötigten Rechte hat, indem sie sich bei der Autorisierung auf einen vertrauenswürdigen Stellvertreter verläßt.

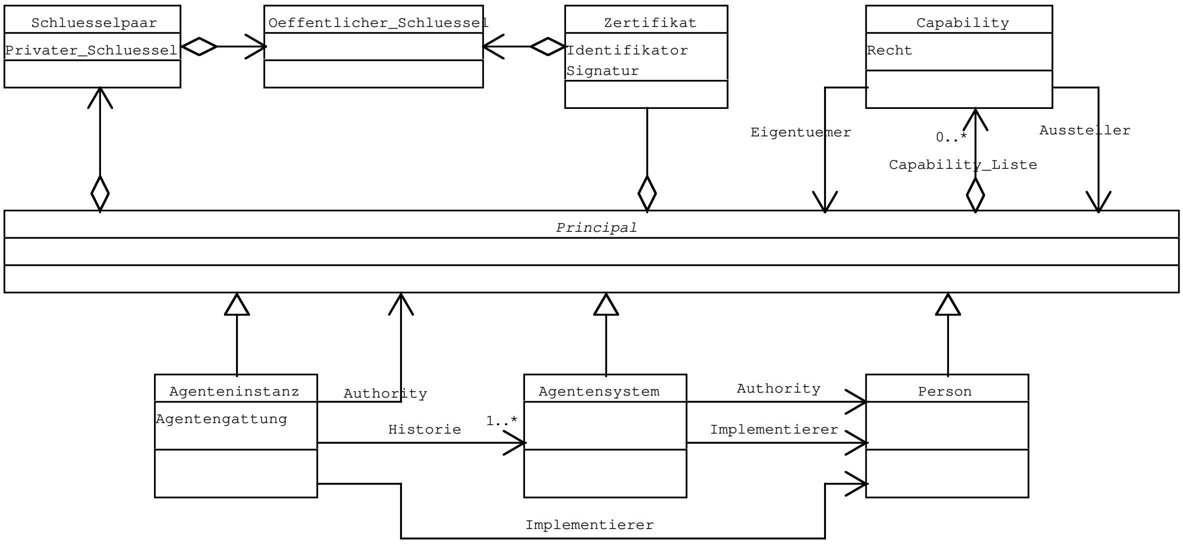

Abbildung 5.10: *Principals* mit Zertifikaten und Capabilities

5.9.3 Ketten von Capabilities: delegierte Rechte

Mit den vorgestellten Konzepten läßt sich nun auch die "Weitergabe" von Rechten vergleichsweise einfach realisieren. Die entsprechende Erweiterung des UML-Diagramms um Capabilities ist in Abbildung 5.10 dargestellt.

Eine Agenteninstanz, die über eine Capability verfügt, kann diese an eine andere Agenteninstanz weitergeben, indem sie selbst eine neue Capability des gleichen Rechts mit der anderen Agenteninstanz als Eigentümer ausstellt. Damit entsteht eine Kette von Capabilities gleichen Rechts, die über die Aussteller und Eigentümer der einzelnen Capabilities verknüpft ist.

Betrachtet man das eingangs vorgestellte Beispiel der beiden Agenteninstanzen A und B, so kann A das Recht F an B folgendermaßen weitergeben: A erzeugt eine neue Capability über `F_lesen`, in der B als Eigentümer eingetragen ist und A selbst als Aussteller fungiert. A übergibt die Capability an B. Will B nun tatsächlich die Datei `F_lesen`, führt S folgende Überprüfungen durch:

1. Hat B gemäß der Permission Policy das Recht F zu lesen: Nein (nach den Annahmen zum Beispiel).

2. Hat B eine Capability die Datei F zu lesen: Ja, explizit.

3. Ist A, der Aussteller der Capability, vertrauenswürdig: Nein

4. Besitzt A eine Capability die Datei F zu lesen: Ja, nämlich implizit über die Permission Policy.

5. Ist S, der Aussteller der impliziten Capability, vertrauenswürdig: Ja

Damit konnte die Aktion `F_lesen` für die Agenteninstanz B autorisiert werden, obwohl S diese Aktion selbst nicht direkt durch seine Permission Policy gestattet, indem über die Kette der Capabilities ein vertrauenswürdiger Aussteller (in diesem Beispiel S selbst) gefunden wurde.

Optional kann, ebenfalls über eine Policy gesteuert, auch noch die zusätzlich Abfrage gestellt werden, ob eine Instanz als berechtigt angesehen wird, überhaupt Capabilities weiterzugeben.

5.10 Domänenbildung

Nachdem nun die Mechanismen für Authentisierung und Autorisierung definiert sind, stellt sich die Frage, wie mit den angegebenen Mitteln ein für das SmA unabhängiger Domänenraum bezüglich der Sicherheit gebildet werden kann.

Eine solche Domänenbildung sollte unabhängig von anderen Domänenkonzepten erfolgen können, um eine maximal flexible Gestaltung der Domänenstruktur zu ermöglichen. Am Beispiel von Agenten für das Netzmanagement wäre es nicht sinnvoll Domänen im SmA an jene der Netzinfrastruktur zu binden, wie das Beispiel aus Kap. 1.2 zeigt.

Weiterhin ist eine flexible Handhabung von unterschiedlichen Rollen, die eine Entität, und dabei speziell der Fall von Personen, im SmA einnehmen kann, sinnvoll. So ist es denkbar, daß eine Person P sowohl als Administrator eines Agentensystems in der Organisation X, als auch als Implementierer für die Organisation Y tätig ist. Ein Domänenkonzept sollte nun die unterschiedlichen Rollen dieser Person handhaben können.

Die Lösung für all diese Anforderungen besteht nun in den für die Authentisierung gebildeten Zertifikatketten.

Bis jetzt wurde die Zertifikatkette einer Entität nur als Beleg der Echtheit für das Zertifikat betrachtet. Durch den strukturierten Aufbau der Zertifikatkette läßt sich hieraus aber auch ein Domänenbegriff ableiten.

So kann die Zugehörigkeit einer Entität zu einer bestimmten Domäne dadurch definiert werden, daß ihr Zertifikat unmittelbar oder mittelbar von einer bestimmten anderen Entität signiert wurde.

Weiterhin können Personen, die in unterschiedlichen Rollen agieren, unterschiedliche Zertifikate besitzen, wobei jedes Zertifikat eine Rolle repräsentiert in der die Person agieren kann. Für obiges Beispiel würde das bedeuten, daß die Person P jeweils ein Zertifikat besitzt, welches von der Organisation X bzw. Y unterschrieben ist. Die Semantik der Rolle, die mit diesen unterschiedlichen Zertifikaten verknüpft ist wird dann in den jeweiligen Policies hinterlegt.

Der größte Vorteil dieses Domänenkonzepts ist dabei die völlige Unabhängigkeit von bestehenden Domänensystemen. Die Semantik der Zertifikathierarchien läßt sich individuell für jede Anwendung frei definieren und beliebig erweitern. Bei Bedarf kann jedoch auch eine schon bestehende Organisationsstruktur übernommenen werden. Existiert diese bereits in Form einer vorgegeben Zertifizierungshierarchie, kann sie, wie in Abschnitt 5.5.5 angedacht, unmittelbar für das SmA übernommen werden.

Gleichzeitig resultiert aus der Unabhängigkeit des Domänenkonzepts aber auch sein größter Nachteil. Dieser besteht darin, daß die Semantik einer Domäne nicht fest mit den Domänenidentifikator (dem Zertifikat) der Domäne gekoppelt ist, sondern unabhängig vom Identifikator in den Policies der Agenteninstanzen und Agentensysteme hinterlegt wird.

5.11 Fallbeispiele

In diesem Abschnitt wird die Anwendung der in diesem Kapitel beschriebenen Verfahren anhand typischer Stationen des Lebenszyklus eines Agenten beschrieben.

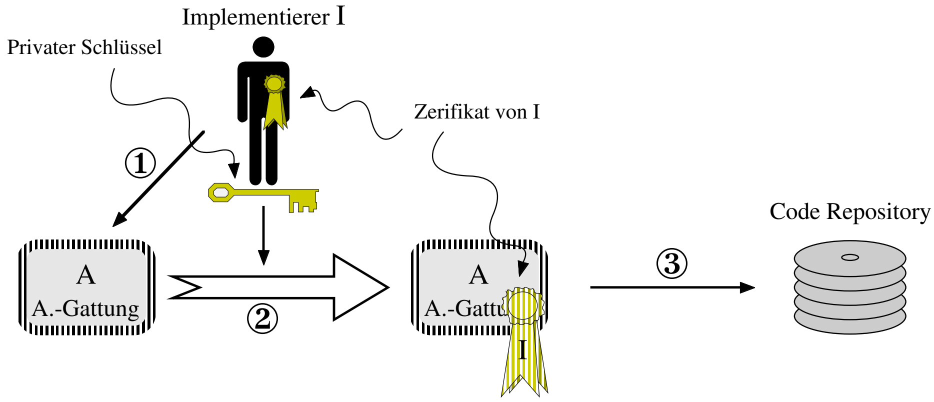

Abbildung 5.11: Skizze der Implementierung einer Agentengattung

5.11.1 Implementierung einer Agentengattung

Sei ein Implementierer nun Besitzer eines persönlichen Zertifikats und des zugehörigen privaten Schlüssels. Abb. 5.11 zeigt die folgenden Schritte:

1. Im Zuge der Implementierung einer Agentengattung A erstellt der Implementierer I den Code der Gattung und evtl. weitere für die Gattung spezifische statische Daten.

2. Code und statische Daten werden dann vom Implementierer I unter Benutzung seines Zertifikat signiert.

3. Die neue Agentengattung A wird in einem Code Repository hinterlegt.

Damit wird folgendes erreicht:

- Der Agentengattung ist ein Implementierer fest und nachvollziehbar zugeordnet.

- Alle Daten der Agentengattung können auf nachträgliche Veränderung überprüft werden.

- Die Gattungsdaten können unabhängig von Agenteninstanzen aus dem Code Repository geladen werden.

5.11.2 Authentisierung eines Anwenders am Agentensystem

1. Der Anwender meldet sich am Agentensystem an, indem er ein Zertifikat präsentiert.

2. Das Agentensystem überprüft, ob die Zertifikatkette gültig ist.

3. Das Agentensystem überprüft anhand der Authentication Policy, ob das Zertifikat als vertrauenswürdig angesehen werden kann, z.B. ob es von einer für das Agentensystem als vertrauenswürdig angesehenen Instanz signiert wurde.

4. Waren beide Überprüfungen positiv, so ist der Anwender erfolgreich vom Agentensystem authentisiert.

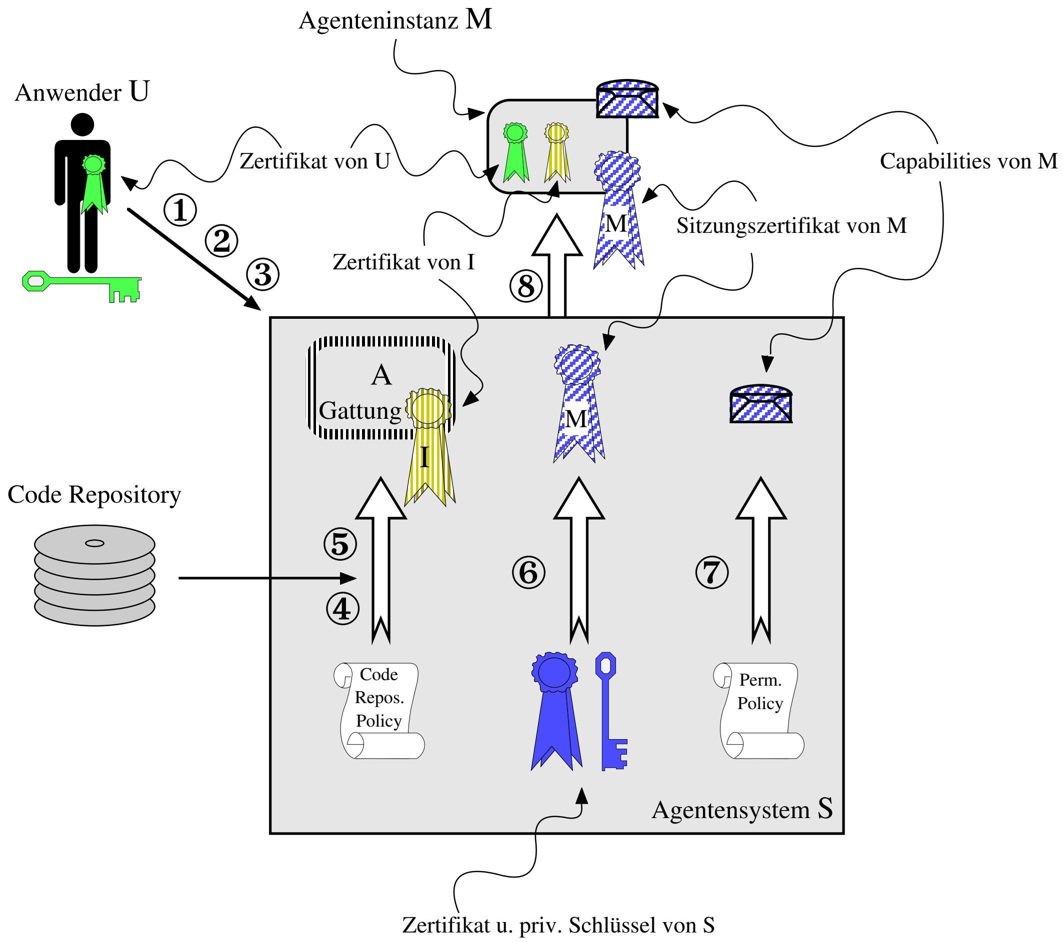

Abbildung 5.12: Skizze der Erzeugung einer Agenteninstanz

5.11.3 Erzeugung einer Agenteninstanz

Abb. 5.12 zeigt die einzelnen Schritte, wie eine neue Agenteninstanz, der in 5.11.1 implementierten Agentengattung A, erzeugt wird:

1. Anwender U meldet dich am Agentensystem S an (siehe Abschnitt 5.11.2).
2. Anwender U erbittet Erzeugung einer Instanz der Agentengattung A.
3. Das Agentensystem S überprüft, ob die eigene Permission Policy dem Anwender das Starten einer Agenteninstanz der Gattung A gestattet.
4. Das Agentensystem S wählt über die Code Repository Policy einen Ort aus, von dem die Daten der Agentengattung A zu laden sind und lädt diese.
5. Das Agentensystem S überprüft die Integrität der geladenen Daten.
6. Das Agentensystem erzeugt (gemäß Abschnitt 5.5.4) ein Sitzungszertifikat und signiert dieses.
7. Das Agentensystem erzeugt eine Menge von Capabilities für diese Agenteninstanz aus der Permission Policy.

8. Das Agentensystem erzeugt eine neue Agenteninstanz M mit den vorher erzeugten Capabilities, setzt dabei die Attribute Authority, Agentengattung, Implementierer und initialisiert die Historie mit der eigenen Identität als ersten Eintrag.

Damit ist die neue Agenteninstanz M der Gattung A, mit Anwender U als Authority und Implementierer I erzeugt und gestartet.

5.11.4 Migration einer Agenteninstanz

1. Die Agenteninstanz A meldet eine Migrations-Anforderung nach Agentensystem D an das Agentensystem S.
2. Das Agentensystem S überprüft, ob die Agenteninstanz im Besitz einer Capability ist, die die Migration auf D gestattet.
3. S kontaktiert D und fordert dessen Zertifikat an.
4. S überprüft die Zertifikatkette von D und entscheidet anhand der Authentication Policy, ob D als vertrauenswürdig angesehen werden kann.
5. S übergibt die Zertifikat-Kette von D an A und A entscheidet mit der eigenen Authentication Policy, ob D als vertrauenswürdig angesehen werden kann.
6. D fordert das Zertifikat von S an und überprüft anhand seiner Authentication Policy, ob S als vertrauenswürdig angesehen werden kann.
7. S serialisiert A und versendet A nach D.
8. D empfängt A und deserialisiert die empfangenen Daten.
9. D überprüft die Integrität der Daten von A. Insbesondere wird dabei die Korrektheit der Historie von A überprüft. Hierzu muß die Integrität der Liste gewährleistet sein und S muß als letzter Eintrag in der Liste stehen.
10. D überprüft anhand seiner Permission Policy, ob die Instanz A aufgrund ihrer Gattung, deren Implementierer, ihrer Authority und ihrer History gestartet werden darf.

Alle weiteren Schritte erfolgen analog zu den Schritten 4 bis 8 der Erzeugung einer neuen Agenteninstanz.

5.11.5 Lokale Agenteninstanz führt Aktion auf Endsystem aus

1. Die Agenteninstanz A möchte eine Aktion X auf dem Endsystem durchführen lassen.
2. Das Agentensystem S überprüft, ob A im Besitz einer Capability ist, die die Aktion X gestattet.
3. Liegt eine passende Capability vor, wird die Aktion ausgeführt, ansonsten verweigert.

5.12 Zusammenfassung

Mit dem vorgestellten Konzept werden die folgenden Forderungen aus der Anforderungsanalyse und der MASA-Risikoanalyse erfüllt:

- Aus der Anforderungsanalyse:
  - Mit den Verfahren aus Abschnitt 5.7 kann zumindest die Integrität von unveränderlichen Daten gewährleistet werden.
  - Den sich zur Laufzeit eines SmA ändernden Strukturen wird durch flexibel formulierbare Policies Rechnung getragen (vgl. Abschnitt 5.8.3).
  - Die Entscheidungsfindung ist feingranular durch die Entscheidungshierarchie, wodurch jede Entität, die für sie sinnvolle Semantik einer Aktion erfassen kann (vgl. Abschnitt 5.8.2).
  - Sicherheitsdomänen können unabhängig von bestehenden Domänenkonzepten erstellt werden (vgl. Abschnitt 5.10).
  - Die Kooperation im SmA wird mit Capability-Ketten zur Rechtedelegation unterstützt (vgl. Abschnitt 5.9).
- Aus der Risikoanalyse von MASA:
  - Alle handelnden Entitäten können sicher identifiziert und wechselseitig authentisiert werden. (vgl. Abschnitt 5.5)
  - Agentengattungen und Agenteninstanzen werden strikt unterschieden (vgl. Abschnitt 5.6).
  - Alle Kanäle im SmA werden gesichert (vgl. Abschnitt 5.4).
  - Die Benutzung aller Schnittstellen kann autorisiert werden (vgl. Abschnitt 5.8.1). Zusätzliche CORBA-Dienste wurden nicht betrachtet.
  - Agentensysteme setzten durch abgeschottete Ausführungsumgebungen einen wechselseitigen Schutz der lokalen Agenten durch und schützen sich selbst vor ihnen (vgl. Abschnitt 5.3).
  - Agentensysteme unterstützen Schutzmaßnahmen von Agenteninstanzen, indem sie die Policy einer Agenteninstanz in die Autorisierungsentscheidung einbeziehen (vgl. Abschnitt 5.8.2).

Auch die wesentlichen Forderungen aus der MASIF-Spezifikation werden erfüllt. Dabei stellen das Agentensystem den geforderten *"Security Service"* und die in Abbildung 5.10 dargestellten Attribute einer Agenteninstanz die *"Credentials"* dar. Den für die Authentisierung vorgeschlagenen "Authenticator" bildet ebenfalls das Agentensystem.

Kapitel 6

Implementierungskonzept

In diesem Kapitel werden zunächst konkrete Techniken vorgestellt, die eine Umsetzung des im vorangegangenen Kapitel entwickelten Modells ermöglichen, und die in der bestehenden MASA-Implementierung bereits nutzbar sind bzw. um die sich die Implementierung erweitern läßt. Anschließend wird die Anwendung dieser Techniken in einer möglichen Implementierung beschrieben.

6.1 Für MASA nutzbare Sicherheitsmechanismen

6.1.1 Java-Mechanismen

Da MASA in der Programmiersprache Java implementiert ist, muß zunächst ein Blick auf die in Java bereits vorhandenen Sicherheitsmechanismen geworfen werden.

Diese Mechanismen gliedern sich in jene, die durch

- die Programmiersprache Java selbst ([JLS]),
- das Java-API ([JDK1.1-SDK][JDK1.2-SDK])

angeboten werden.

Sicherheitskonzepte in der Sprache Java

Das Design der Sprache Java realisiert bereits einige Mechanismen, die geeignet sind, Sicherheitseigenschaften zu gewährleisten.

Die Basis aller Sicherheitseigenschaften in Java wird durch das Fehlen jeglicher Zeigerarithmetik in der Programmiersprache gebildet.

Es ist nicht möglich, einen Zeiger auf primitive Datentypen zu erzeugen (in C++ mit dem `&` Operator). Objektdatentypen werden immer über Referenzen angesprochen. Diese können nicht explizit dereferenziert werden (`* und ->` Operator in C++), und es sind keine arithmetische Operationen auf Referenzen definiert (in C++ können die Operatoren `+ - * /` auch auf Zeiger angewandt werden). Dadurch wird garantiert, daß willkürliche Speicherzugriffe, und damit auch jene in "fremde" Speicherbereiche innerhalb der JVM, ausgeschlossen sind.

Weiterhin garantiert Java eine strikte Typüberprüfung, die nicht nur zur Compilezeit, sondern auch zur Laufzeit durchgesetzt wird. Illegale Veränderungen von Werten durch Typumwandlungen (type casts) sind somit in Java ausgeschlossen.

Durch die fehlende Zeigerarithmetik und die stringente Typüberprüfung realisiert Java einen strikten Speicherschutz. In der Folge heißt dies, daß ein Objekt definitiv nur dann verändert werden kann, wenn eine Referenz auf dieses Objekt regulär erhalten wurde.

Eine erste fallweise Differenzierung des Zugriffs auf Daten wird durch Visibilitätsmodifikatoren realisiert, die anwendbar auf Klassen, ihre Attribute und ihre Methoden sind.

Um diese erläutern zu können muß zunächst noch eine weiteres Konzept der Java Sprache betrachtet werden. Alle Klassen der Sprache werden bei ihrer Deklaration in einen hierarchischen Namensraum, in sog. Packages, eingeordnet. Diese werden nach Konvention durch den komponentenweise rückwärts geschriebenen DNS-Namen der implemtierenden Instanz gebildet. Beispielsweise würde eine Klasse, die von der Firma Sun implementiert wird in das Package `com.sun.` eingeordnet. Damit wird eine hierarchische Gliederung aller in Java deklarierten Klassen erreicht.

Für Klassen stehen in Java zwei Visibilitätsmodifikatoren zu Verfügung:

public: Die Klasse ist von allen anderen Klassen aus sichtbar.

package: Die Klasse ist nur von Klassen im gleichen Package aus sichtbar.

Die Visibilitätsmodifikatoren von Attributen und Methoden sind:

public: Das Attribut/die Methode ist von allen Klassen aus sichtbar.

package: Das Attribut/die Methode ist nur von Klassen innerhalb des gleichen Package aus sichtbar.

protected: Das Attribut/die Methode ist nur von Tochter-Klassen aus sichtbar. Dabei spielt es keine Rolle, ob sich die Tochter-Klasse im gleichen oder einem anderen Package befindet als die Mutterklasse.

private: Das Attribut/die Methode ist ausschließlich in der deklarierenden Klasse sichtbar. Im Fall von Methoden bedeutet dies auch, daß eine Tochterklasse diese Methode nicht überladen kann.

Zur Kennzeichnung der Klassen, Attribute und Methoden sind in der Java-Syntax die Schlüsselworte `public`, `protected` und `private` definiert. Für "package" existiert kein eigenes Schlüsselwort. Wird keines der drei definierten Visibilitätsmodifikatoren explizit angegeben, wird implizit "package"-Sichtbarkeit angenommen.

Neben den Visibilitätsmodifikatoren gibt es noch einige weitere Modifikatoren für Klassen bzw. Attribute und Methoden, wovon hier noch der `final`-Modifikator beschrieben sei. Dieser Modifikator, angewendet auf Klassen und Methoden, garantiert, daß eine Klasse nicht abgeleitet, bzw. eine Methode nicht überladen werden kann. Auf Attribute angewandt bedeutet es, daß ein in einem Konstruktor initial zugewiesene Wert des Attributs nicht mehr verändert werden kann.

Die Einhaltung der durch die Modifikatoren zugesicherten Eigenschaften wird vom Java-Compiler bei der Übersetzung überprüft. D.h. beispielsweise, wenn in einer Klasse versucht wird, auf ein als `privat` deklariertes Attribut einer anderen Klasse zuzugreifen, so wird dies durch den Compiler als Fehler gemeldet und der Übersetzungsvorgang wird abgebrochen.

Nun mag man einwenden, daß es aber während der Laufzeit auch möglich wäre diesen Schutz zu umgehen, indem man das *Reflection*-API ([JDK1.2-SDK]) benutzt, um Zugriff auf ein solches Attribut zu erlangen. Das Attribut bzw. der Methodenzugriff wird jedoch während der Laufzeit durch die JVM überprüft, und gegebenenfalls eine Ausnahmebehandlung eingeleitet.

Mit vorgestellten Modifikatoren lassen sich also bereits auf Ebene der Programmiersprache Java statisch Zugriffseigenschaften sicherstellen. Diese Modifikatoren können in MASA nun dazu benutzt werden, um alle statischen Zusicherungen von Zugriffsstrukturen zu implementieren. Ihre Überwachung und Durchsetztung wird dabei vom Seiten des Compilers bzw. der JVM übernommen, weshalb es diesen von seiten des Implementierungskonzepts keiner weiteren Aufmerksamkeit bedarf.

Java Platform 2 Security Architecture

Für die Sicherung der Schnittstelle zum Endsystem, welche durch das Java-API dargestellt wird, bietet sich die Nutzung der bereits im Java-API vorgesehenen Sicherheitsmechanismen an.

An dieser Stelle wird nur ein kurzer Überblick geboten, ausführliche Informationen zum Sicherheitsmodell finden sich in [JCA 98] und [Gon 98], die Anwendung des Security-API wird [Oaks 98] detailliert beschreiben.

JDK 1.1 unterscheidet in seinem Sicherheitsmodell nur 2 Arten von Code:

- Vertrauenswürdigen Code, der ohne Einschränkungen ausgeführt wird
- Nicht vertrauenswürdigen Code, der in der sog. *Sandbox* ausgeführt wird und dessen Rechte damit eingeschränkt werden.

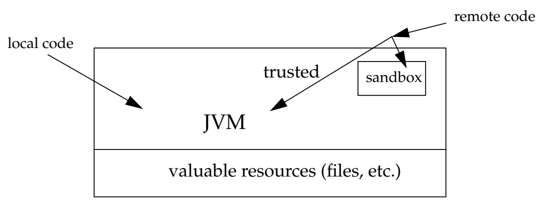

Abbildung 6.1: Sicherheitsmodell von JDK 1.1 (aus [Gon 98])

Dabei ist Code, der vom lokalen Dateisystem geladen wird, immer vertrauenswürdig, ebenso wie entfernter Code (z.B. von einem Webserver), der von einer beliebigen Person signiert wurde und dessen Zertifikat überprüft werden kann.

Die Durchsetzung der Sicherheitseigenschaften in der *Sandbox* wird dabei durch den sog. *Security Manager* realisiert. Dieser muß individuell vom Implementierer entsprechend der Bedürfnisse der Anwendung entwickelt werden und in Form einer Klasse, die von `java.lang.SecurityManager` abgeleitet wird, implementiert werden. Dabei werden durch das API keine Mechanismen bereitgestellt, die es ermöglichen eine weitere Differenzierung (z.B. nach dem Unterzeichner) des Codes vorzunehmen, um damit beispielsweise auf einfache Weise mehrere *Sandboxes* mit unterschiedlichen Rechten zu realisieren. Deshalb können die im Sicherheitsmodell aufgestellten Forderungen mit JDK 1.1 nicht realisiert werden.

Mit JDK 1.2 wird dieses starre Modell erweitert. Unabhängig von der Quelle des Codes (lokal oder entfernt) und ob dieser signiert wurde, kann Code einer von mehreren *Sandboxes* zugeteilt werden.

Jede *Sandbox* wird durch eine *Protection Domain* repräsentiert, der Identifikator einer Protection Domain ist die *Code Source*. Einer Protection Domain wiederum sind *Permissions* zugeordnet, die die Rechte einer Protection Domain bestimmen. Die Zuteilung des Codes zu einer Protection Domain wird durch eine *Security Policy* bestimmt, ebenso die zugehörigen Permissions.

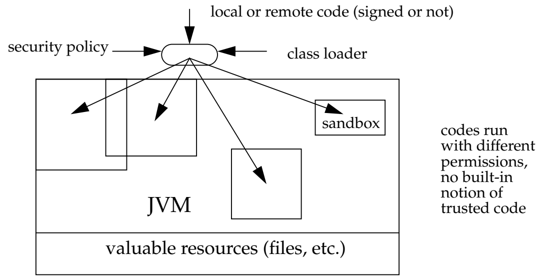

Abbildung 6.2: Sicherheitsmodell von JDK 1.2 (aus [Gon 98])

Die Durchsetzung der von den Permissions bestimmten Rechte wird dann vom Security Manager in Zusammenarbeit mit dem *AccessController* vorgenommen. Soll eine privilegierte Aktion ausgeführt werden, zu der ein bestimmtes Recht erforderlich ist, so überprüft der Security Manager mittels des Access Controllers, ob die Protection Domain, in der sich der Code befindet, welcher die privilegierte Aktion ausführen möchte, eine entsprechende Permission besitzt.

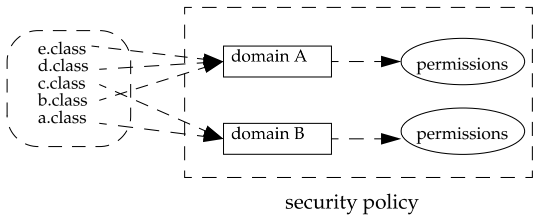

Abbildung 6.3: Zusammenhang von Code, Protection Domain und Permissions (aus [Gon 98])

Permissions sind dabei für alle Aktionen vordefiniert, die vormals in JDK 1.1 durch den Security Manager manuell überprüfbar waren. Weiterhin lassen sich anwendungsspezifische Permissions definieren, die ebenfalls in der Security Policy verwendet werden können.

Durch die Security Policy, die in einer Datei festgelegt wird und den (erweiterbaren) Permissions, ist es in JDK 1.2 möglich, ein vergleichsweise feingranulares und ohne Programmieraufwand konfigurierbares Sicherheitsmodell zu implementieren, welches auf die Bedürfnisse der Anwendung zugeschnitten ist.

Java Cryptography Extension (JCE)

Mit dem JCE-API ([JCE 99]) steht ein umfangreiches Werkzeug für die Nutzung kryptographischer Basistechniken zur Verfügung. Es enthält in Form von Klassen beispielsweise Datenstrukturen und Funktionen für:

- Schlüsselerzeugung, -Verwaltung und -Speicherung
- Zertifikaterzeugung, -Verwaltung und -Speicherung
- Digitale Signaturen
- Chiffrierung

Eine auch nur einführende Behandlung von JCE würde an dieser Stelle zu weit führen, hierfür sei auf [JCE 99], [JCA 98] und [Oaks 98] verwiesen. Zusammengefaßt stellt JCE eine standardisierte Schnittstelle für eine Reihe von Basisfunktionalitäten dar, wie sie für die Implementierung von kryptographiegestützter Anwendungen notwendig ist.

6.1.2 Secure Socket Layer (SSL) V3.0

Das in [FKK 96] definierte Protokoll SSL V3.0 realisiert einige grundlegende Mechanismen zur Kanalsicherung.

Obwohl [FKK 96] nie als offizieller Standard verabschiedet wurde, hat es sich zu einem de-facto Standard in Internet-Anwendungen entwickelt. Alle namhaften Webbrowser unterstützen dieses Protokoll. Im RFC-Dokument [DiAl 99] wird TLS, der Nachfolger von SSL V3.0 beschrieben. Da TLS unmittelbar auf [FKK 96] basiert, es im wesentlichen SSL V3.0 entspricht und noch nicht weit verbreitet ist, wird im Folgenden SSL V3.0 kurz betrachtet.

SSL bietet sichere Verbindungen zwischen zwei Kommunikationspartnern. Es baut auf ein beliebiges, verläßliches Transportprotokoll auf. Auf der anderen Seite agiert es transparent für die Anwendungsschicht. Beispielsweise läßt sich SSL zwischen die Protokolle HTTP (Anwendungsprotokoll) und TCP/IP (Transportprotokoll) "dazwischenschieben", ohne daß eines der beide Protokolle geändert werden muß. Trotzdem sind die durch HTTP übertragenen Daten ab der TCP/IP-Schicht und darunter gesichert.

Folgende sicherheitsrelevante Eigenschaften werden von SSL unterstützt:

Authentisierung: SSL unterstützt den sicheren Austausch von X.509-Zertifikaten ([X.500]) zur Authentisierung. Tauschen beide Kommunikationspartner X.509-Zertifikate aus, so können sie sich wechselseitig authentisieren.

Verschlüsselung und Authentizitätschutz: Über SSL verschickte Daten werden mit einem MAC (Message Authentication Code) versehen, womit ihre Authentizität sichergestellt wird, und dann verschlüsselt übertragen.

Die konkreten Parameter einer SSL-Verbindung, wie z.B. Schlüssellängen und die zu verwendende Verschlüsselungsalgorithmen, können individuell zwischen den beiden Kommunikationspartnern über ein Handshake-Protokoll vereinbart werden, das ebenfalls Teil der SSL-Spezifikation ist.

Eine ausführliche Untersuchung der Sicherheitseigenschaften von SSL findet sich in [WaSc 96]. Zusammenfassend läßt sich sagen, daß SSL auf einfachem Weg die Möglichkeit bietet, bestehende, auf TCP/IP basierende Protokolle gegen Angriffe zu schützen. Somit lassen sich auch die in MASA verwendeten CORBA/IIOP- und HTTP-Kanäle mit SSL gegen die in 2.3 skizzierten Angriffe absichern. Zusätzlich werden Kommunikationspartner und ausgetauschte Daten authentisiert.

6.2 Der ORB und die Konformität zum MASIF-Standard

Viele der in MASIF geforderten Eigenschaften sind nur realisierbar, wenn ein ORB zur Verfügung steht, der konform zu *Security Functionality Level 2* der *CORBA Sercurity Services Specification* ist. Zum Zeitpunkt dieser Arbeit ist, allerdings keine auch nur annähernd vollständige Implementierung der *Security Functionality Level 2* als Produkt verfügbar.

Die meisten Produkte, die am Markt erhältlich sind, implementieren mehr oder minder vollständig *Security Functionality Level 1*, indem sie die in [OMG 98-12-17] Kap 15.14 beschriebene Nutzung von IIOP über SLL implementieren.

Nun würde die Implementierung eines ORB nach *Security Functionality Level 2* den Rahmen dieser Arbeit bei Weitem sprengen. Somit stellt sich die Frage, wie unter Einhaltung der

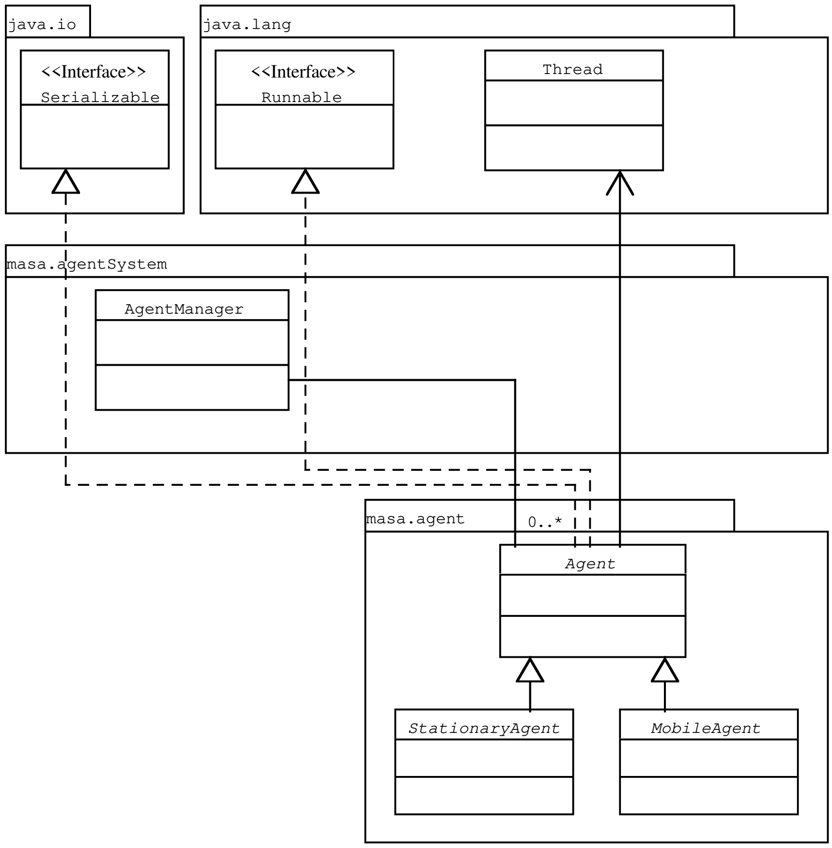

Abbildung 6.4: Klasse Agent – Vererbung und Package-Struktur der bisherigen Implementierung

MASIF-Spezifikation dennoch ein Implementierungskonzept einer Sicherheitsarchitektur für MASA gefunden werden kann.

Um einen möglichst großen Teil der Forderungen aus Kap. 4.1.1 konform zur *CORBA Sercurity Services Specification* nutzen zu können, ist deshalb zumindest die Verwendung eines ORBs mit SSL-Unterstützung sinnvoll. Weitere Anforderungen, die in den *MASIF Security Service Requirements* gestellt und durch den verwendeten ORB nicht geleistet werden, müssen dann abseits des Standards implementiert werden.

Durch SSL wird die Sicherung der Kanäle und die Authentisierung über Zertifikate unterstützt. Somit werden diese Teile bereits durch einen SSL-fähigen ORB bereitgestellt. Autorisierung, Überwachung von Schnittstellen sowie Delegation und Domänenkonzepte werden erst mit *Security Functionality Level 2* standardisiert und stehen deshalb nicht zur Verfügung. Darum sind diese Teile abseits des Standards zu implementieren.

Ordnet man das hieraus resultierende Implementierungskonzept in die Kategorien aus Kap. 4.1.1 ein, wäre dies sowohl in Kategorie 2 als auch in Kategorie 3 einzuordnen.

6.3 Ausführungsumgebungen von Agentensystem und Agenteninstanzen

6.3.1 Änderungen in Klassen `Agent`/ `AgentSystem`/`AgentManager`

Um den in Kap. 5.3 geforderten Speicherschutz von Agenteninstanzen (und des Agentensystems selbst) realisieren zu können, ist in Java nach Abschnitt 6.1.1 durch die fehlende Zeigerarithmetik bereits eine gute Basis vorhanden. Um nun auch tatsächlich unerwünschte Zugriffe auf Attribute und Methoden von Agentenklassen und Klassen des Agentensystems verhindern zu können, müssen nur die in Abschnitt 6.1.1 vorgestellten Visibilitätsmodifikatoren konsequent eingesetzt werden.

Die Grundvoraussetzung für diesen Ansatz, nämlich die jeweils getrennten Packages von Agentensystem und allen Agentengattungen ist bereits in der bestehenden MASA-Implementierung realisiert.

Für die Sicherung des Zugriffs auf Attribute sollte generell auf die Möglichkeit des direkten Zugriffs auf Attribute durch andere Klassen verzichtet werden. Stattdessen sollten Zugriffsmethoden angeboten werden, die es ermöglichen, lesend (`get`-Methode) bzw. schreibend (`set`-Methode) auf das Attribut zuzugreifen. Entsprechend sollte der Visibilitätsmodifikator eines Attributs immer `private` lauten. Neben der Möglichkeit der Unterscheidung zwischen lesenden/schreiben Zugriffen in der Autorisierung, was bei direkten Attributzugriffen nicht möglich ist, bietet dieses Vorgehen noch den Vorteil, daß bei schreibenden Zugriffen zusätzlich eine Überprüfung auf semantische Korrektheit des zu setzenden Wertes durchgeführt werden kann.

Für Klassen, die als interne Klassen des Agentensystemems bzw. einer Agentengattung fungieren, läßt sich dies einfach erreichen, indem die Klasse als Package deklariert wird. Damit können alle Attribute, unabhängig von ihrem Visibilitätsmodifikator nur noch aus dem eigenen Package angesprochen werden. Für Klassen, die `public` deklariert sein müssen, weil sie Methoden für Klassen aus anderen Packages bereitstellen (dies gilt insbesondere für jene Klassen, die CORBA-Methoden implementieren), muß der Zugriffsschutz durch die Visbilitätsmodifikatoren der einzelnen Attribute gewährleistet werden.

In diesem Zusammenhang müssen besonders die Basis-Klassen `agent.Agent` und deren Tochterklassen `agent.StationaryAgent` und `agent.MobileAgent` beachtet werden. Um den in Kap. 3.4.1 geschilderten Bedrohungen entgehen zu können muß eine Reorganisation dieser Klassen vorgenommen werden.

Da die Klasse `agent.Agent` kritische Attribute und Methoden enthält (vgl. Kap. 3.4.1), die u.a. vom Agentensystem für die Verwaltung einer Agenteninstanz benötigt werden und auf die eine Agenteninstanz aber höchstens lesend zugreifen darf, müssen diese in einer Klasse deklariert werden, die im Package `agentSystem` liegt. Diese neue Klasse wird `AgentEnvironment` genannt und soll in Zukunft alle Attribute und Methoden aufnehmen, die einer Agenteninstanz zuordenbar sind. Am Beispiel der Methode `setThread()`, die nur vom Agentensystem benötigt wird, betrachtet: Die Umdeklarierung der Methode zu `package` oder `private` ist nicht möglich. Zwar könnte die Agenteninstanz sie dann nicht mehr nutzen, das Agentensystem aber ebenfalls nicht. Wird `setThread()` aber in der Klasse `AgentEnvironment` deklariert, kann sie mit `package`-Sichtbarkeit deklariert werden, wodurch nur noch Klassen des Agentensystems Zugriff haben.

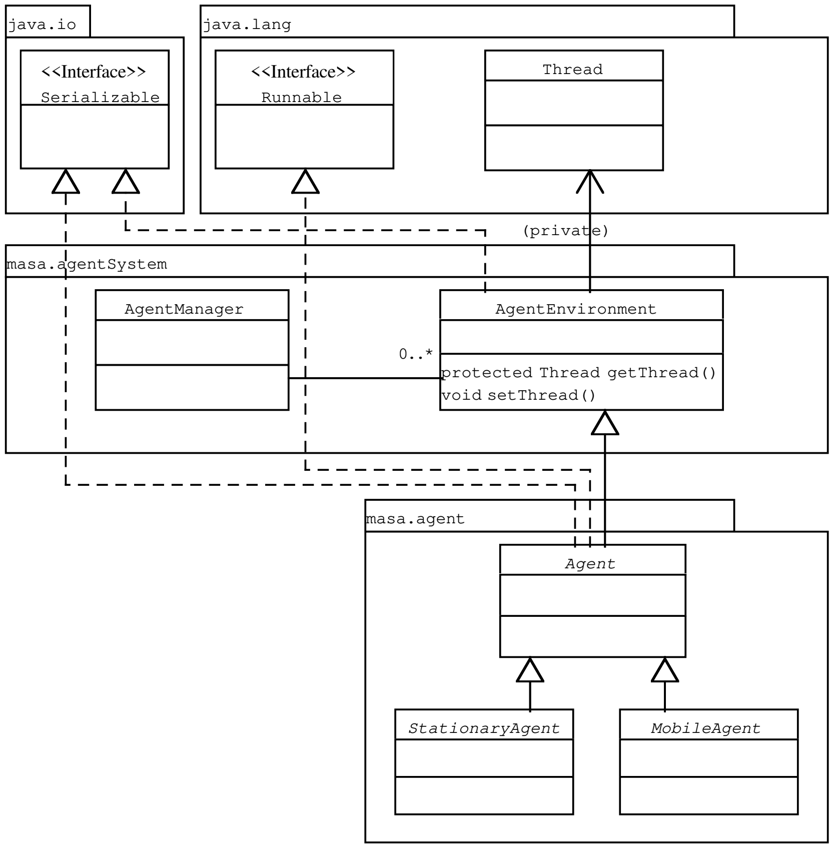

Abbildung 6.5: Klasse Agent – Neue Vererbung und Package-Struktur

Um den lesenden Zugriff auf Attribute durch die Agenteninstanz aber zu ermöglichen, wird die Klasse `agent.Agent` von `AgentEnvironment` abgeleitet. Wird dann die `get()`-Methode eines Attributs `protected` deklariert, kann auch die Agenteninstanz das Attribut lesen, da die `get()`-Methode von der Agenteninstanz erreicht werden kann.

Der Vorteil der in der bisherigen Implementierung gemachten Ansiedelung charakteristischer Attribute in der Basis-Klasse `agent.Agent` liegt darin, daß diese im Fall der Serialisierung einer Agenteninstanz (z.B. zur Migration) automatisch mitserialisiert werden, da jede Gattung indirekt immer von `agent.Agent` abgeleitet ist. Dieser Vorteil geht aber nicht verloren, da `agent.Agent` von `agentSystem.AgentEnvironment` abgeleitet ist. Die automatische Serialisierung der Attribute ist durch die Ableitung weiterhin gewährleistet, durch die unterschiedlichen Packages der Klassen ist der statische Zugriffsschutz aber weiterhin gewährt.

6.3.2 Laufzeitumgebung

Für die in Kap. 5.3 geforderten getrennten Laufzeitumgebungen für Agenteninstanzen bietet sich die Verwendung von Threads und ThreadGroups als Laufzeitmodell an.

In der bestehenden Fassung von MASA werden allerdings noch nicht alle Möglichkeiten genutzt, die Java bietet, um Threads gegeneinander abzuschotten. Zwar werden Threads, die zu Agenteninstanzen gehören in einer eigenen ThreadGroup zusammengefaßt und so von Threads des Agentensystems getrennt, aber es wird keine Unterscheidung zwischen Threads von verschiedenen Agenteninstanzen getroffen. Folglich besteht kein Schutz gegen die Einflußnahme auf Threads durch fremde Agenteninstanzen.

Weiterhin ist von Seiten des Agentensystems nur der Haupt-Thread, nämlich jener, der durch das Agentensystem bei Erzeugung der Instanz gestartet wird, identifizierbar. Sub-Threads, die eine Instanz selbst erzeugt hat, können nicht identifiziert werden. Dies hat zur folge, daß bei Terminierung einer Instanz die Sub-Threads nicht zwangsweise terminiert werden können, vielmehr ist das Agentensystem hier auf die Kooperation der Instanz angewiesen.

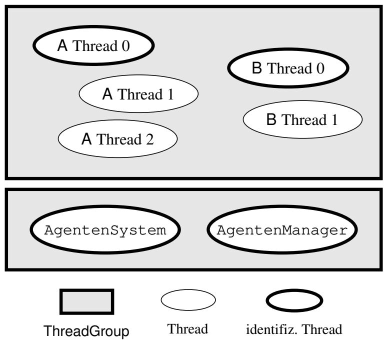

Abbildung 6.6: Threads der bestehenden Implementierung

Um eine solche Einflußnahme ausschließen zu können müssen demnach alle Threads, die zu einer Agenteninstanz gehören in einer eigenen ThreadGroup zusammengefaßt werden. Durch den SecurityManager ist es dann möglich, Zugriffe auf fremde Threads zu erkennen und zu verhindern.

Damit werden auch Sub-Threads einer Agenteninstanz identifizierbar, da diese immer in der ThreadGroup der Instanz erzeugt werden. Damit kann durch das Agentensystem alleine gewährleistet werden, daß bei Terminierung wirklich alle zur Instanz gehörenden Threads gestoppt werden.

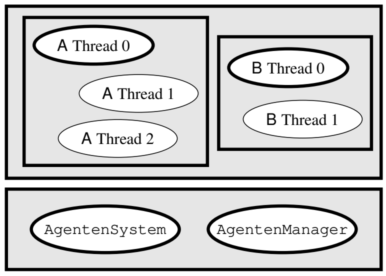

Abbildung 6.7: Eigene ThreadGroups für Agenteninstanzen

6.3.3 Namensräume: Der Java Classloader

Nachdem nun auch getrennte Laufzeitumgebungen für Agenteninstanzen geschaffen sind, ist nur noch dafür Sorge zu tragen, daß keine Namenskonflikte zwischen Agenteninstanzen und besonders zwischen einer Agenteninstanz und dem Agentensystem auftreten können.

Die Trennung von Namensräumen wird in Java durch sogenannte Classloader ([JDK1.2-SDK]) bewerkstelligt. Ein Classloader ist eine spezielle Klasse, die für das Laden von Code zuständig ist und vom runtime linker der JVM angesprochen wird.

Ein Namensraum besteht nun aus allen Klassen, die von der gleichen Instanz eines Classloader geladen wurden. Folglich ist es für die Trennung der Namensräume zwischen Agenteninstanzen notwendig, daß jeder Agenteninstanz eine eigene Instanz eines Classloaders vom Agentensystem zugewiesen wird.

Hierfür erzeugt das Agentensystem, bevor es die Hauptklasse der Agentengattung instanziiert, eine neue Classloader-Instanz. Durch diese lädt das Agentensystem dann den Code der Hauptklasse aus dem Code Repository. Erst dann wird die Hauptklasse instanziiert. Jede weitere Klasse, welche die Hauptklasse benötigt, wird dann automatisch durch diesen Classloader geladen. Damit werden alle Klassen, die die Agenteninstanz benutzt, vom gleichen Classloader geladen und die Instanz hat einen eigenen getrennten Namensraum.

6.4 Sichere Kanäle

Die in Kap. 5.4 aufgestellten Forderungen an eine Implementierung von sicheren Kanälen werden unter Verwendung des in Abschnitt 6.1.2 vorgestellten Protokolls SSL V3.0 erfüllt.

Konkret wird der HTTP-Kanal durch Verwendung von SSL V3.0 anstatt eines "blanken" TCP/IP-Stroms gesichert. Dieses Verfahren bezeichnet man als "HTTP über SLL" (kurz HTTPS) und wird von allen gängigen Webbrowsern unterstützt.

Der CORBA-Kanal wird mittels eines SSL-fähigen ORBs nach Abschnitt 6.2 gesichert.

Die Benutzung des Java-Kanals zur Kommunikation zwischen lokalen Agenten oder mit dem eigenen Agentensystem wird mit dem Implementierungskonzept untersagt, stattdessen ist immer der CORBA-Kanal zu verwenden. Die einzige Ausnahme besteht in der in Abschnitt 6.3.1 beschriebenen Basisklasse `agent.Agent`, da diese Methoden enthält, deren Rückgabewerte sich nicht über den CORBA-Kanal übertragen lassen. Beispielsweise kann kann das Thread-Objekt der Methode getThread() nicht über CORBA übertragen werden.

Die Möglichkeit der Verwendung des Java-Kanals anstatt des CORBA-Kanals wird auf Ebene der Klassen durch die getrennten Namensräume ausgeschlossen. Wollten beispielsweise zwei lokale Agenteninstanzen A und B über den Java-Kanal kommunizieren wird dies dadurch unterbunden, daß die hierfür notwendige Referenzierung einer Klasse von B im Namensraum von A verhindert wird, indem der Classloader von A keine Definition einer Klasse aus dem Namensraum vom B gestattet.

6.5 Code Repositories

Für Code Repositories muß eine neue Komponente implementiert werden, der Code Repository Manager. Er interpretiert die Code Repository Policy und realisiert den Zugriff auf Agentengattungen. Er bietet Schnittstellen für das Laden von Code bzw. anderer Daten, die zu einer Agentengattung gehören, an.

6.5.1 Agenteninstanzen in JAR-Dateien

Um Agentengattungen einfach in *Code Repositories* ablegen zu können, ist es sinnvoll alle statischen Daten in einer einzigen Einheit zusammengefaßt. Hierzu sind die sog. JAR-Dateien ([JDK1.2-SDK]) hervorragend geeignet.

JAR-Dateien bieten die Möglichkeit beliebige Dateien und Verzeichnisse in einer einzigen Datei zusammenzupacken. Im Fall einer MASA-Agentengattung sind dies konkret:

- Der Code des Agenten und seines Applets in Form von `.class`-Dateien.
- Dateien für die HTML-Seite des Applets, wie z.B. Bilder.
- Die Dokumentation der Agentengattung.
- Sonstige gattungsspezifische Dateien.

Die Erstellung solcher JAR-Dateien wird dabei mit dem *jar*-Tool (Teil des JDK) durchgeführt.

Mit JAR-Dateien wird somit eine Agentengattung durch genau eine Datei repräsentiert. Durch die kompakte Darstellung wird die Distribution von Agentengattungen und ihre Speicherung in Code Repositories erheblich vereinfacht.

Mit der Möglichkeit der digitalen Signierung einer JAR-Datei nach [JDK1.2-SDK] wird dann auch die in 5.5.1 geforderte Unveränderlichkeit der Agentengattung und die Zuordnung des Implementierers sichergestellt. Dies geschieht indem der Implementierer der Agentengattung sein Zertifikat der JAR-Datei hinzufügt und diese dann mit seinem privaten Schlüssel signiert. Dieser Vorgang wird komplett durch das *`jarsigner`*-Tool (Teil des JDK) ([JDK1.2-SDK]) unterstützt.

Für den Zugriff auf JAR-Dateien und deren Authentisierung bietet das Java-API im Package `java.util.jar` entsprechende Klassen an.

6.6 Authentisierung

Für die Authentisierung und Zertifikathandhabung muß in MASA eine neue Komponente geschaffen werden. Da jegliche Authentisierung in MASA über Zertifikate erfolgt, soll diese Komponente als *Certificate Manager* bezeichnet werden.

Für die Implementierung des *Certificate Manager* werden dabei die Datenstrukturen und Funktionen für den Umgang mit X.509-Zertifikaten komplett durch JCE (vgl. 6.1) bereitgestellt.

6.6.1 Zertifikaterteilung und Zertifikatüberprüfung

Der *Certificate Manager* ist für die Erstellung von Zertifikaten im Agentensystem nach Kap. 5.5.4 zuständig:

- Erstellung und Speicherung des eigenen Agentensystemzertifikats (vgl. Kap. 5.5.3). Für die Erstellung muß die Signatur durch das Zertifikat der Authority unterstützt werden.
- Erstellung von Sitzungszertifikaten für Agenteninstanzen.

Der *Certificate Manager* wird in MASA auch für die Überprüfung der Gültigkeit und Vertrauenswürdigkeit von Zertifikaten benutzt. Er führt die Integritätsüberprüfung von Zertifikatketten durch und bestimmt anhand der *Authentication Policy*, ob ein Zertifikat vertrauenswürdig ist. Hierzu verfügt er nach Kap. 5.5.5 optional über eine Schnittstelle zu einem Zertifikatserver.

6.6.2 Zertifikatübermittlung

Die für die Authentisierung notwendige Übermittlung von Zertifikaten wird bereits durch die Sicherung der Kanäle mittels SSL V3.0 unterstützt. Dadurch ist der Austausch von Zertifikaten zwischen zwei Entitäten gewährleistet. Können keine Zertifikate ausgetauscht werden, weil z.B. eine Entität kein Zertifikat vorweisen kann, so wird bereits durch das SLL-Protokoll dieser Fehler erkannt und eine Kommunikation kommt nicht zustande.

Wurden die Zertifikate erfolgreich ausgetauscht, muß durch den *Certificate Manager* bestimmt werden, ob das Zertifikat als vertrauenswürdig angesehen werden kann. Um hierfür an das Zertifikat zu gelangen werden für die jeweiligen Kanäle die folgenden Mechanismen benutzt:

- HTTPS: Die verwendete SSL-Implementierung muß einen entsprechenden Bibliotheksaufruf zur Verfügung stellen, der das Zertifikat, das durch das SSL-Handshake-Protokoll übermittelt worden ist, bereitstellt.
- CORBA/SSL: Nach [OMG 98-12-17] kann durch das CORBA-`Current`-Objekt das Zertifikat des Kommunikationspartners erhalten werden.

6.7 Autorisierung und Überwachung der Schnittstellen

Für das Fällen der Entscheidung, ob eine Aktion ausgeführt werden darf oder nicht, muß in MASA eine neue Komponente eingeführt werden, der *Permission Manager*.

Der *Permission Manager* implementiert dann die in Kap. 5.8.3 informell beschriebene Syntax für Policies in MASA. Weiterhin muß er die Möglichkeit bieten, dynamisch die *Permission Policies* von Agenteninstanzen zu integrieren, sowie die *Capabilities* einer Agenteninstanz miteinbeziehen. Des weiteren besitzt er eine Schnittstelle, mit der abgefragt werden kann, ob eine Aktion für eine konkrete Entität erlaubt ist.

6.7.1 Integration in die Java Platform 2 Security Architecture

Um die mit der Java Platform 2 Security bereits vorhandenen Mechanismen zur Autorisierung von Aufrufen des Java-API nutzen zu können muß der Permission Manager in die JDK 1.2 Sicherheitsarchitektur integriert werden. Dies geschieht, indem die Implementierung des Permission Managers von `java.security.Policy` abgeleitet wird und dann als globale Java-Policy nach [JDK1.2-SDK] in der JVM installiert wird.

Um im Sicherheitsmodell von JDK 1.2 die lokalen Agenteninstanzen repräsentieren zu können, ist mit den Protection Domains bereits ein genau passender Mechanismus vorhanden. Errichtet man für jede lokale Agenteninstanz eine eigene Protection Domain, so können durch die in JDK 1.2 bereits vorhandenen Sicherheitsmechanismen die einzelnen Agenteninstanzen unterschieden werden.

Wird nun durch einen Java-API Aufruf der Java Access Controller befragt, bezieht dieser indirekt den Permission Manager in die Entscheidung mit ein, da der Access Controller die Java System Policy benutzt, um zu entscheiden, ob die zum Aufrufer gehörige Protection Domain über ein entsprechendes Recht verfügt. Über die Protection Domain kann dann der als System Policy Objekt installierte Permission Manager herausfinden, um welche Agenteninstanz es sich bei dem Aufrufer handelt und entsprechend die Regeln der Permission Policy anwenden.

6.7.2 Überwachung der Schnittstellen

Für die Überwachung der Schnittstellen muß zwischen zwei Arten unterschieden werden:

- "eigene" Schnittstellen, dies sind jene, die von der überwachenden Entität selbst implementiert werden:
  - CORBA-Schnittstellen des Agentensystems werden von diesem selbst überwacht.
  - CORBA-Schnittstellen einer Agenteninstanz, die diese selbst überwacht.

- "fremde" Schnittstellen, also jene, die von einer anderen als der überwachenden Entität implementiert werden:
  - CORBA-Schnittstellen einer Agenteninstanz, die nach der Entscheidungshierarchie durch das Agentensystem überwacht werden.
  - Schnittstellen des Endsystems, die durch das Agentensystem überwacht werden.

"Eigene" CORBA-Schnittstellen

Die Überwachung eigener CORBA-Schnittstellen wird einfach dadurch realisiert, daß die Methoden, die die einzelnen Funktionen der Schnittstelle implementieren, überwacht werden.

Konkret führt man in der Implementierung einer solchen Methode als erste Aktion die Entscheidung durch, ob der Funktionsaufruf autorisiert werden kann oder nicht. Dies geschieht dadurch, daß man über den Permission Manager eine Abfrage der Permission Policy durchführt.

"Fremde" CORBA-Schnittstellen

Für den Fall, daß eine fremde CORBA-Schnittstelle überwacht werden soll, ist die im vorangegangenen Abschnitt beschriebene Technik nicht einsetzbar, da kein Zugriff auf den Quelltext der zu überwachenden Methoden besteht.

Wird an einer CORBA-Schnittstelle einer Agenteninstanz eine Funktion aufgerufen, so geschieht dies konkret dadurch, daß die ORB-Instanz die entsprechende Methode in der Agenteninstanz ausführt. Damit nun das Agentensystem diese Schnittstelle überwachen kann, müßte das Agentensystem durch den ORB zur Autorisierung befragt werden, bevor dieser die Methode der Agenteninstanz aufruft. Das Agentensystem ist also für diesen Fall auf die Zusammenarbeit mit dem ORB angewiesen und kann von sich aus keine Maßnahmen ergreifen.

Nur wenn der verwendete ORB die sog. *Access Decision* Objekte (Teil der Security Functionality Level 2; [OMG 98-12-17]) unterstützt, ist eine standardisierte Schnittstelle für die Kooperation mit dem ORB gegeben. Da aber kein z. Zt. erhältlicher ORB die Security Funcionality Level 2 implementiert, besteht im Rahmen dieses Implementierungskonzepts keine direkte Möglichkeit, daß ein Agentensystem die Schnittstellen einer Agenteninstanz überwacht.

Schnittstellen des Endsystems

Für die Schnittstellen des Endsystems, die durch das Java-API repräsentiert werden, bietet die Java 2 Security Architecture bereits die entsprechenden Möglichkeiten. Da der Permission Manager mit Abschnitt 6.7.1 in die Sicherheitsarchitektur eingebunden ist, werden diese Mechanismen automatisch benutzt.

Da bei Verwendung einer Methode aus dem Java-API, die kritische Aktionen durchführt, und somit bestimmte Rechte benötigt, automatisch der Access Controller befragt wird, wird mit der in Abschnitt 6.7.1 angegebenen Technik über den Permission Manager auch die Permission Policy ausgewertet.

Durch den Access Controller wird dann überprüft, ob die Agenteninstanz, aus der der Aufruf durchgeführt wurde, über ein entsprechendes Recht verfügt, das den Aufruf der Methode gestattet.

Die Verwendung von zusätzlichen Bibliotheken, die keine Überwachung durch den Access Controller implementieren, kann nicht bis auf die Ebene der einzelnen Methoden kontrolliert werden. Allerdings kann durch die Trennung der Namensräume eine Kontrolle bis auf die Ebene der Klassen durchgeführt werden.

Soll beispielsweise die Benutzung einer SNMP-Bibliothek, die im Package `com.one_company.snmp` implementiert ist, exklusiv der Gattung com.other_company.SNMP_Agent erlaubt werden. Dann kann der Zugriff auf die Bibliothek eingeschränkt werden, indem nur die Classloader von Agenteninstanzen der Gattung com.other_company.SNMP_Agent das Laden von Code aus dem Package `com.one_company.snmp` gestatten. Die Classloader anderer Agenten dagegen verweigern das Laden von Klassen aus diesem Package. Zur Entscheidung, ob eine Klasse geladen werden darf, befragt der Classloader den Permission Manager.

6.8 Klassen für gesicherte Attribute

Um die in Kap. 5.7 beschriebenen Attributtypen für unveränderliche Daten lassen sich unmittelbar durch das Java-API realisieren:

- öffentliche, unveränderliche Daten: `java.security.SignedObject`
- private, unveränderliche Daten: `javax.crypto.SealedObject`

Für die KAG-Protokolle müssen mit Hilfe von `javax.crypto.SealedObject` bzw. `java.security.SignedObject` eigene Klassen nach [KAG 98] implementiert werden.

6.9 Darstellung von Rechten und Capabilities

Die in Kap. 5.8.3 definierten Rechte lassen sich durch die `java.security.Permission` Objekte der Java 2 Security Architecture darstellen.

Rechte, die sich auf die Schnittstelle des Endsystems beziehen werden dabei bereits durch das API definiert. So können beispielsweise Zugriffsrechte auf Dateien durch `java.io.FilePermission` dargestellt werden.

Für MASA-Spezifische Rechte können neue Permission Objekte erzeugt werden, die dann von `java.security.Permission` abgeleitet werden. Beispielsweise könnte das Recht zur Erzeugung einer Agenteninstanz der bestimmten Gattung durch eine eigene Klasse

```
public class createAgentPermission extends java.security.permission
```

dargestellt werden. Dabei könnten dann die in der permission-Klasse deklarierten Attribute `name` den erlaubten Gattungsidentifikatoren aufnehmen.

Da das Konzept der Capabilities aus Kap. 5.9.1 nicht in der Java 2 Security Architecture vorgesehen ist, müssen diese erst implementiert werden. Für die Attribute Recht, Eigentümer und Aussteller existieren mit `java.security.Permission` und `java.security.cert.X509Certificate` bereits die benötigten Klassen. Die Signatur des Tupels Recht und Eigentümer durch den Ersteller kann mit der Klasse `java.security.Signature` implementiert werden.

6.10 CORBA Services

Da CORBA Services häufig durch ein Fremdprodukt, das nicht im Quelltext vorliegt, realisiert werden, ist der in Abschnitt 6.7.2 benutzte Weg des direkten Einbaus der Autorisierung in die Java-Methoden, welche die CORBA-Funktionen implementieren, ausgeschlossen. Außerdem verhindern Quelltextmodifikationen die generelle Austauschbarkeit verschiedener Implementierungen, wie sie durch die CORBA-Standards eigentlich möglich wären.

Da gängige Produkte entsprechende Schnittstellen zur Autorisierung nicht bereitstellen, müssen für diese Fälle dann spezielle Security-Proxy-Objekte implementiert werden. Diese werden dann den CORBA-Schnittstellen des ursprünglichen Services "vorgeschalten" und ersetzen so dessen CORBA-Funktionen durch eigene, in denen Authentisierung und Autorisierung vorgenommen werden, um schließlich, falls beide Schritte erfolgreich waren, die korrespondierenden Funktionen des eigentlichen CORBA-Services aufzurufen.

Ebenso ist häufig die zu verwendende ORB-Instanz vorgegeben, die dann nicht SSL-fähig ist. Durch den Einsatz von Security-Proxy-Objekten wird diesem Problem ebenfalls begegnet, indem diese eine SSL-fähige ORB Instanz benutzen, die dann wiederum die nicht-SSL fähige Implementierung des Services benutzen. Dabei muß beachtet werden, daß der Kanal zwischen Security-Proxy-Objekt und CORBA-Service sicher ist, was sich dadurch realisieren läßt, daß beide in einer gemeinsamen JVM ablaufen, und somit der Kanal ein Java-Methodenaufruf ist, und damit sicher ist.

Kapitel 7

Realisierung

Zur Demonstration der Umsetzbarkeit, der in den beiden vorangegangenen Kapiteln entwickelten Konzepte, wurde die bestehende MASA-Implementierung erweitert.

Die Beschreibung der Implementierung erfolgt hier nur im Überblick, eine detaillierte Dokumentation der einzelnen Klassen, sowie ihrer Attribute und Methoden entnehme man der javadoc-Dokumentation des Agentensystems bzw. der einzelnen Agentengattungen. Zur Vereinfachung der Darstellung wurden der Präfix `de.unimuenchen.informatik.mnm.masa.` bei den jeweiligen Klassen weggelassen.

7.1 Neue Produkte für die Realisierung

Für die Realisierung war zunächst eine CORBA-Entwicklungsumgebung zu wählen, die folgende Rahmenbedingungen zu erfüllen hatte:

- Konformität zu CORBA V2.0 oder höher.
- Unterstützung von Java JDK 1.2.
- ORB mit SSL-Unterstützung.
- Unterstützte Plattformen: Linux, Sun Solaris, HP-UX, Microsoft Windows NT.
- Vorhandene CORBA-Dienste: *Naming Service*, *Event Service*.

Da das Unix-Derivat Linux nicht nur im akademischen Umfeld zunehmend an Bedeutung gewinnt und auch am Lehrstuhl, an dem MASA entwickelt wird, Linux massiv Einsatz findet, wurde auf dessen Unterstützung als Entwicklungs- und Laufzeitplattform besonderen Wert gelegt.

Zwar bietet die Firma Inprise, der Hersteller der bisher verwendeten Umgebung Visibroker, eine SSL-Erweiterung an, allerdings existierte keine Unterstützung von Linux als Laufzeitplattform. Konkret verwendet Visibroker eine als betriebssystem-native Applikation vorliegende Komponente (den sog. *osagent*), die Voraussetzung für den Einsatz von Visibroker ist und nicht für Linux verfügbar ist.

7.1.1 Die CORBA-Entwicklungsumgebung Orbacus 3.1.2

Das Produkt Orbacus 3.1.2 des Herstellers Object Oriented Concepts (OOC) ([OOC Site]) erfüllt alle Anforderungen an eine neue CORBA-Umgebung:

- Orbacus ist konform zu CORBA V2.0 und besitzt bereits einige Erweiterungen der CORBA V2.2 Spezifikation.
- JDK 1.2 wird explizit unterstützt, daneben auch weiterhin die Vorgängerversion JDK V1.1.x.
- Mit dem Produkt OrbacusSSL existiert eine passende SSL-Erweiterung.
- Orbacus ist für die Sprachen C++ und Java erhältlich. In der Java-Version ist die gesamte Laufzeitumgebung selbst in Java implementiert, damit werden alle Plattformen unterstützt, die kompatibel zu JDK 1.1.x/1.2 sind [1]. Der für den Entwicklungsprozeß notwendige IDL-nach-Java-Übersetzer ist Teil der C++ Version, die folgende Plattformen unterstützt: SGI IRIX, Sun Solaris, HP-UX, IBM AIX, Linux, DEC OSF/1 und Microsoft Windows 95/98/NT. [2]
- Neben *Event Service* und *Naming Service* ist auch noch eine Implementierung des *Property Service* Teil der C++ und der Java-Version. In der C++ Version ist auch noch ein *Interface Repository* vorhanden.

Weitere Merkmale sind:

- Frei für nicht-kommerzielle Nutzung.
- Quelltext öffentlich verfügbar.

Bewertung und Erfahrungsbericht

Neben den aufgezählten Leistungsmerkmalen erwies sich die Unterstützung seitens des Herstellers durch eine Mailingliste als hervorragend. Gestellte Fragen wurden prompt und kompetent durch die Entwickler selbst beantwortet. Die bereits im Rahmen von [Roel 98] gemachten positiven Erfahrungen mit Orbacus bestätigen dabei die Alltagstauglichkeit des Produkts.

Als weiterer Vorteil des Produkts ist die Tatsache zu werten, daß sowohl die C++ als auch die Java-Variante, anders als bei Visibroker, komplett im Quelltext vorliegen. Probleme durch einfache Bugs, wie sie beispielsweise in [Bran 99] mit Visibroker aufgetreten sind, lassen damit nicht nur einfacher nachvollziehen, sondern in eiligen Fällen auch selbst beheben, bis der Hersteller eine korrigierte Version liefert.

Neben diesen technischen Eigenschaften besitzt Orbacus noch einen positiven wirtschaftlichen Aspekt: Die gesamte Laufzeitumgebung ist für die Verwendung in nicht kommerziellen Projekten frei, was dem MASA-Projekt zugute kam. Dies gilt auch für die SSL-Erweiterung, für die im Fall von Visibroker ein merklicher Betrag zu entrichten gewesen wäre.

Da mit Orbacus ein vollwertiger Ersatz für das bisher verwendete Visibroker gefunden wurde, ist dessen Unterstützung für die erweiterte MASA-Version eingestellt worden. Zu den Zukunftsperspektiven von Orbacus bleibt zu erwähnen, daß gerade die Nachfolgeversion 4.0 zum

[1] Am Lehrstuhl bereits getestete Plattformen: Linux, Sun Solaris, HP-UX (nur JDK V1.1.x), Apple MacOS (nur JDK V1.1.x)

[2] Details und eine Liste der unterstützten C++ Compiler siehe [OOC Site].

Alpha-Test durch OOC freigegeben wurde, welche die komplette CORBA V2.3 Spezifikation implementieren wird.

7.1.2 Die SSL-Erweiterung OrbacusSSL 1.0.1

Mit OrbacusSSL 1.0.1 bietet OOC eine SSL-Erweiterung für Orbacus 3.1.x an. Ebenso wie das Hauptprodukt werden für die SSL-Erweiterung die Sprachen C++ und Java unterstützt, wiederum sind beide Versionen im Quelltext verfügbar und die nicht-kommerzielle Nutzung frei.

Allerdings beinhaltet OrbacusSSL selbst keine eigene Implementierung von SSL-Strömen und keine kryptographischen Algorithmen, hierfür werden zusätzliche Fremdprodukte benötigt:

- SSLeay ([SSLEAY Site]) für die C++ Version.
- IAIK-JCE und IAIK-iSaSiLk für die Java Version.

OrbacusSSL 1.0.1 erfüllt die *Security Functionality Level 1* aus [OMG 98-03-09] nur teilweise. Zwar werden durch die Verwendung von SSL die Forderungen hinsichtlich von Authentisierung und sicherer Kommunikation erfüllt, allerdings werden nicht alle in der CORBA Security Specification geforderten Schnittstellen implementiert.

Bewertung und Erfahrungsbericht

Leider bestätigten sich im Laufe der Arbeiten an der Implementierung die positiven Eindrücke des Hauptprodukts für die SSL-Erweiterung nur eingeschränkt:

- Insgesamt ist das Handbuch zu OrbacusSSL ([OrbSSL 98]) als zu knapp zu bewerten.
- Zum Beginn der Implementierungsarbeiten war eine Unterstützung von JDK V1.2 nicht offiziell vorgesehen. Dank des Quellcodes war es jedoch möglich in wenigen Tagen eine entsprechend modifizierte Version für JDK 1.2 zu erstellen.
- Die Implementierung selbst zeigte im Laufe der Arbeiten ebenfalls einige Schwächen, die in Abschnitt 7.2.5 beschrieben werden.

7.1.3 Die Kryptographie-Bibliothek IAIK-JCE 2.5.1

IAIK-JCE 2.5.1 vom Institut für angewandte Informationsverarbeitung und Kommunikationstechnologie der Technischen Universität Graz (IAIK) ist eine unabhängige Neuimplementierung der Java Cryptography Extension V1.2 (JCE) ([JCE 99]), die aufgrund von US-Exportbeschränkungen nicht außerhalb der USA vertrieben werden darf, und stellt eine Java-Erweiterung im Rahmen der Java Cryptographic Architecture (JCA) ([JCA 98]) dar.

Die folgenden Algorithmen werden u.a. unterstützt, für eine komplette Übersicht siehe [IAIK Site]:

- Symmetrische Schlüsselverfahren: DES, TripleDES, IDEA, Blowfish, GOST, CAST128
- Asymmetrische Schlüsselverfahren: RSA, DSA, DH, ECDSA
- Sichere Hash-Algorithmen: MD2, MD5, SHA-1, RipeMd128, RipeMd160

- Spezifikationssprache: Abstract Syntax Notation One (ASN.1)
- PKCSS-Familie: PKCS#1, PKCS#5, PKCS#7, PKCS#8, PKCS#10, PKCS#12
- X.509-Zertifikate: Alle X.509v3 Standard Extensions, einige X.509v2 CRL Extensions, sowie Netscape Certificate Extensions

IAIK-JCE ist frei für Lehrzwecke und wird in Form von fertig übersetzten JAR-Dateien geliefert.

Neben der Standardausgabe ist noch eine sog. Applet-Edition erhältlich, die den gleichen Leistungsumfang besitzt, die Klassen jedoch in einem anderen Package implementiert als in [JCA 98] vorgesehen, um Probleme mit der internen Sicherheitsarchitektur mancher Webbrowser zu vermeiden (siehe auch [IAIK Site]).

Bewertung und Erfahrungsbericht

Nachdem mit der Version 2.5.1 die Probleme der Vorgängerversion 2.5 mit JDK 1.2 behoben wurden, hat sich IAIK-JCE als zuverlässige Bibliothek für alle gängigen kryptologischen Verfahren bewährt. Besonders die Unterstützung der proprietären Netscape Erweiterungen für X.509-Zertifikate hat sich im praktischen Betrieb zur Nutzung der weitverbreiteten Netscape Produkte als äußerst hilfreich erwiesen.

Die API-Dokumentation [IAIK-JCE API] ist als gut zu bewerten, allerdings fehlt ein Benutzerhandbuch. Dieses Manko wird durch eine Mailingliste etwas aufgefangen, wo Anwender und Entwickler Fragen beantworten.

7.1.4 Die SSL-Bibliothek IAIK-iSaSiLk 2.5.1

Mit IAIK-iSaSiLk 2.5.1 liefert IAIK eine vollständige Implementierung des SSL V3.0 Protokolls ([FKK 96], vgl. 6.1.2), die komplett in Java verfaßt ist und auf die kryptographischen Funktionen von IAIK-JCE zurückgreift.

IAIK-iSaSiLk 2.5.1 umfaßt Socket-Klassen für den Aufbau von SSL-Strömen über TCP/IP. Dabei sind diese Socket-Klassen und jene des Java-API (vgl. [JDK1.2-SDK]) angelehnt. Eine Übersicht dieser Klassen und ihrer Anwendung, sowie eine Liste der unterstützten Verschlüsselungsalgorithmen findet sich in [iSaSiLk].

Analog zu IAIK-JCE ist IAIK-iSaSiLk 2.5.1 frei für Lehrzwecke und wird in JAR-Dateien geliefert, ebenso ist eine Applet-Edition verfügbar.

Bewertung und Erfahrungsbericht

Während der Implementierung erwiesen sich die Klassen von IAIK-iSaSiLk 2.5.1 als äußert einfach und problemlos in der Handhabung. In der Zusammenarbeit mit SSL-fähigen Webbrowsern traten keine nennenswerten Probleme auf.

Mit [iSaSiLk] und [iSaSiLk API] steht eine gute Dokumentation des Produkts zur Verfügung.

7.2 Vorarbeiten

7.2.1 Entflechtung der Quellen

Zu Beginn der Implementierung wurden zunächst die Quelldateien des bisherigen MASA entflochten, da sich im gleichen Zweig des CVS-Repositories sowohl Klassen des Agentensystems, als auch Klassen spezifischer Agentengattungen befanden. Notwendig wurde dieser Schritt, um getrennte JAR-Distributionen der einzelnen Agenten nach Kap. 6.5.1 erstellen zu können.

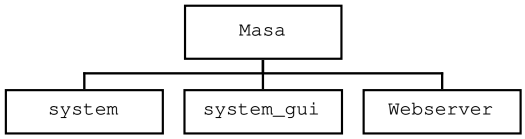

Abbildung 7.1: Oberstruktur im CVS-Repository

Die sich daraus ergebende Oberstruktur im CVS-Repository ist in Abbildung 7.1 zu sehen:

`system/`: Das Agentensystem mit Basis- und allgemeinen Hilfsklassen zur Agentenentwicklung

`system_gui/`: Das AgentSystemApplet zur Steuerung des Agentensystems, sowie der ASManagementAgent, einem Hilfsagenten für Applets (siehe [Gerb 99]).

`Webserver/`: Der Webserver Agent

Gleichzeitig wurde sowohl für das Agentensystem als auch für Agenten eine neue Produktionsumgebung entwickelt, die in Anhang A beschrieben wird. Für die Agenten Webserver und ASManagementAgent kommt diese neue Produktionsumgebung bereits zum Einsatz.

7.2.2 Portierung auf JDK 1.2 und Orbacus 3.1.2

Da für die Umsetzung des Implementierungskonzepts JDK 1.2 eine Grundvoraussetzung ist, wurden zunächst das Basissystem, sowie die Agenten Webserver und ASManagementAgent auf JDK 1.2 portiert. Dabei traten, durch die identische Sprachsyntax zum bisher verwendeten JDK 1.1.x, mit einer Ausnahme, keine größeren Probleme auf.

In JDK 1.2 ist die Methode `stop()` der Klasse `java.lang.Thread` als *deprecated* bezeichnet, d.h. sie sollte nicht mehr verwendet werden. Allerdings ist diese Methode essentiell für die Terminierung einer Agenteninstanz, konkret für das (zwangsweise) Anhalten eines Threads. Sun stellt momentan keine für MASA akzeptable Alternative für `stop()` bereit, weshalb diese Methode trotzdem weiterhin verwendet wird, bis ein entsprechender Ersatz durch eine spätere JDK-Version verfügbar ist.

Weiterhin wurde in diesem Schritt die alte CORBA-Entwicklungsumgebung Visibroker gegen Orbacus 3.1.2 ausgetauscht. Erheblich erleichtert wurde die gesamte Umstellung dabei durch die neue Produktionsumgebung.

7.2.3 Klassen für symbolische Konstanten

Während der Entflechtungsarbeiten fiel auf, daß einige Konstanten "hart" in die Quelltexte von Agenten bzw. des Agentensystems einkodiert waren. Um die Wartbarkeit der Quellen zu

Abbildung 7.2: GUI des LoggerConfigAgent

verbessern wurden deshalb neue Klassen angelegt, die ausschließlich Konstantendefinitionen[3] enthalten.

Die Klassen

- Klasse `tools.AgentSystemType`
- Klasse `tools.LanguageID`
- Klasse `tools.SerializationID`

enthalten Konstantendefinitionen des MASIF-Standards (vgl. [OMG 98-03-09]).

In der Klasse `tools.GlobalConstants` befinden sich MASA-spezifische Konstantendefinitionen wie z.B. der MASA-Versionsstring.

Für die CVS-Verzeichnisse `system/` und `system_gui/` wurden alle Quellen nach "'hartkodierten" Konstanten durchsucht, und diese durch entsprechende symbolische Konstanten aus den beschriebenen Klassen ersetzt.

7.2.4 Hilfsmittel zur Fehlerausgabe

Um eine Basis für einen generischen Logging-Mechanismus zu schaffen und die Fehlersuche während der Entwicklung zu erleichtern, wurde die sehr einfache Möglichkeit zu Fehlerausgabe in `tools.Debug` durch einen leistungsfähigeren Logging-Mechanismus ersetzt.

Dieser unterstützt nun, in Anlehnung an den `syslog`-Mechanismus von Unix-Systemen, verschiedene Quellen (*Facilities*) von Fehlerausgaben und unterschiedliche "Dringlichkeiten" (*Levels*). Weiterhin wird die automatische Ausgabe von `java.lang.Trowable`-Objekten unterstützt.

Die zur Verfügung stehenden *Facilities* sind in der Klasse `tools.Debug` in den Konstanten mit dem Präfix `FACILITY_`, die *Levels* in den Konstanten mit dem Präfix `LEVEL_` definiert.

[3]Es existieren zwar keine "echten" Konstanten in Java, eine Attributdeklaration mit `public static final` kann aber als äquivalent angesehen werden.

Zur Ausgabe von Fehler- bzw. Logmeldungen sind folgende Methoden definiert:

- `logMessage( int inFacility, int inLevel, String[] inMessage)`
- `logMessage( int inFacility, int inLevel, String inMessage)`
- `logMessage( int inFacility, int inLevel, String[] inMessage, Throwable inException)`
- `logMessage( int inFacility, int inLevel, String inMessage, Throwable inException)`

Daneben stehen, für Debug-Zwecke, die folgenden Methoden bereit:

- `logEnvProps( int inFacility, int inLevel)`

  Gibt die Belegung einiger Standard Java-Properties aus.

- `logThreadGroup( int inFacility, int inLevel, ThreadGroup inTG)`

  Gibt eine Übersicht aller Threads aus, die zur *Thread Group* `inTG` gehören.

Neben diesen Funktionen besteht mit dem neuen Logging-Mechanismus noch die Möglichkeit, die Ausgabe von Meldungen nach *Facility* und *Level* zu unterdrücken bzw. zu ermöglichen. Die Konfiguration erfolgt dabei über den LoggerConfigAgent. Am Applet des LoggerConfigAgent (Abbildung 7.2) läßt sich die Ausgabe von *Facility*/*Level*-Kombinationen einzeln steuern, sowie der Kopf, der zu jeder Meldung ausgegeben wird, konfigurieren.

7.2.5 Korrekturen an Orbacus/OrbacusSSL

An den Produkten Orbacus und OrbacusSSL waren einige Änderungen notwendig, um diese für die neue MASA-Implementierung verwenden zu können. Ursache hierfür waren Implementierungsmängel der jeweiligen Produkte, die sich erst im Laufe der Implementierung der erweiterten MASA-Version zeigten.

Die an Orbacus bzw. OrbacusSSL durchgeführten Änderungen liegen der MASA-Distribution in Form von Unix `patch`-Dateien bei.

Orbacus

Um den *Naming Service* von Orbacus über eine SSL-Verbindung ansprechen zu können, mußte die Quelldatei `com/ooc/CosNaming/Server.java` geringfügig modifiziert werden, damit der *Naming Service* selbst ein SSL-fähiges ORB-Objekt benutzt. Für den *Event Service* war eine entsprechende Modifikation nicht notwendig, da die zu verwendende ORB-Instanz nicht durch die Implementierung des Service selbst vorgeschrieben wird.

OrbacusSSL

An OrbacusSSL waren umfangreichere Änderungen notwendig. Da OrbacusSSL nur ein Zertifikat pro ORB-Instanz unterstützt, mußte in der MASA-Implementierung jeder Agenteninstanz, sowie dem Agentensystem selbst, eine eigene ORB-Instanz zugeteilt werden, damit diese ihre individuellen Zertifikate für die CORBA/SSL-Kommunikation benutzen können.

Dabei zeigte sich jedoch, daß das Attribut in OrbacusSSL, welches das zu verwendende Zertifikat referenziert, in der Quelldatei `com/ooc/SSL/impl/SSLimpl.java` als `static` deklariert war. Die Folge war, daß *alle* ORB-Instanzen das Zertifikat des zuletzt instanziierten ORB zur SSL-Kommunikation benutzen, womit OrbacusSSL in der vorliegenden Form für MASA unbrauchbar gewesen wäre. Durch Entfernen des `static` Modifikators und einigen weiteren Änderungen, die durch Seiteneffekte aus dem geänderten Attributmodifikator notwendig wurden, konnte dieses Manko jedoch behoben werden.

Weiterhin wurde im `SSLCurrent`-Objekt die Zertifikatkette des Kommunikationspartners in Form eines `com.ooc.SSL.X509CertificateChain`-Objekts geliefert. Dieses enthielt allerdings nur eine DN-Repräsentation der einzelnen Zertifikate und nicht die kompletten Zertifikate inklusive öffentlichem Schlüssel und X.509 Extensions. Somit wären Überprüfungen der Zertifikatkette auf ihre Integrität durch MASA ausgeschlossen gewesen. Deshalb wurde die Klasse `com.ooc.SSL.IAIK.X509Certificate` und die zugrundeliegende IDL-Definition um das Attribut `rawIAIKCert` erweitert, welches eine serialisierte Form des Java-Objekts `com.IAIK.X509Certificate` enthält.

7.3 Eindeutige Identifikatoren

Zur Beseitigung der in Kap. 3.2.2 geschilderten Unzulänglichkeiten im Umgang mit String-Repräsentationen der Klasse `CfMAF.Name`, wurde die zugehörige Hilfsklasse `tools.NameWrapper` angepaßt, so daß alle drei Komponenten von `CfMAF.Name` signifikant für die String-Repräsentation sind. Die Umsetzung erfolgt nun in der Methode `toString()` nach folgendem Schema:

`name_string :== [`<*identity*>`!`<*authority*>`!`<*type*>`]`

Entsprechend dieser Definition wurden auch der Konstruktor `NameWrapper( String inNameStr)` angepaßt, so daß dieser die inverse Funktion zu `toString()` implementiert. Damit liefert jetzt das Fragment

```
NameWrapper a, b;
String      aStr;
a    = new NameWrapper(\ldots);
aStr = a.toString();
b    = new NameWrapper( aStr);
```

zwei identische Objekte `a` und `b`.

Die Änderung der `toString()`-Methode führte zu vielfachen Seiteneffekten, da in der alten Implementierung häufig direkte Vergleiche mit dem Ergebnis von `toString()` und einem `identity`-Strukturelement gemacht wurden, anstatt die entsprechende `equals()`-Methode zu benutzen. Alle hieraus resultierenden Fehler wurden in den CVS-Verzeichnissen `system/` und `system_gui/` behoben.

7.4 Code Repositories

Der *Code Repository Manager* wurde durch die Klasse `agentSystem.AgentCodeRepository` implementiert. Sie enthält folgende `static` deklarierte Methoden:

`getCodeRepositoryEntry(...)`: Sucht eine Agentengattung gemäß der Code Repository Policy.

`getAvailableAgents(...)`: Listet die in alle Code Repositories zur Instanziierung verfügbaren Agentengattungen auf und ersetzt damit die in [Gerb 99] eingeführte Datei `ImplementedAgents.txt`.

Wird mittels `getCodeRepositoryEntry(...)` eine Agentengattung gefunden, so wird ein `agentSystem.AgentCodeRepositoryEntry`-Objekt zurückgegeben. Über dieses Objekt ist dann der ortstransparente Zugriff auf die Daten der Agentengattung möglich.

Um konkrete Zugriffstechniken (z.B. via ftp, aus einer Datenbank, etc.) auf Agentengattungen zu implementieren, muß eine neue Klasse von `agentSystem.AgentCodeRepository` abgeleitet werden und die dort `abstract` deklarierten Methoden implementiert werden. Ebenso muß eine passende Subklasse von `agentSystem.AgentCodeRepositoryEntry` erstellt werden. Für den Zugriff auf das lokale Dateisystem des Endsystems wurden die Klassen `agentSystem.LocalFilesystemCodeRepository` und `agentSystem.LocalFilesystemCodeRepositoryEntry` realisiert.

Eine explizite Repräsentation der Code Repository Policy existiert nicht, ihre Implementierung ist aber in `agentSystem.AgentCodeRepository` vorbereitet.

7.4.1 jar files von Agenten

Die neue Produktionsumgebung für Agenten unterstützt die Erstellung von JAR-Dateien und deren Signierung nach Kap. 6.5.1.

7.5 Authentisierung, Schlüssel- und Zertifikathandhabung

7.5.1 Standalone Tool `masa_keytool`

Um in der Entwicklungs- und Testphase auf einfache Weise, ohne komplette Zertifizierungshierarchie, Zertifikatketten für Authorities (vgl. 5.5.4) "out of band" erstellen zu können, wurde das *masa_keytool* implementiert. Mit diesem Werkzeug ist es möglich, mittels einer GUI (Abbildung 7.3), interaktiv Zertifikatketten zu erstellen.

Dabei werden, neben einigen X.509v3 Standard Extensions, auch die proprietären Netscape Extensions unterstützt, die benötigt werden, um die generierten Zertifikate mit Netscape Webbrowsern verwenden zu können[4]. Die Unterstützung der diversen Extensions ermöglicht *masa_keytool* die Generierung von Root CA, Sublevel CA und User Zertifikaten.

Abgespeichert werden die generierten Zertifikate im PKCS#12 Format, das von allen gängigen Browsern importiert werden kann.

7.5.2 Klasse `AgentSystemCertManager`

Die Klasse `agentSystem.AgentSystemCertManager` realisiert die in Kap. 6.5 beschriebenen *Certificate Manager* des Agentensystems. Konkret sind in dieser Klasse Methoden für die folgenden Aufgaben enthalten:

[4]Erfolgreich getestete Versionen: Navigator 4.06 und 4.07, Communicator 4.5x und 4.6x

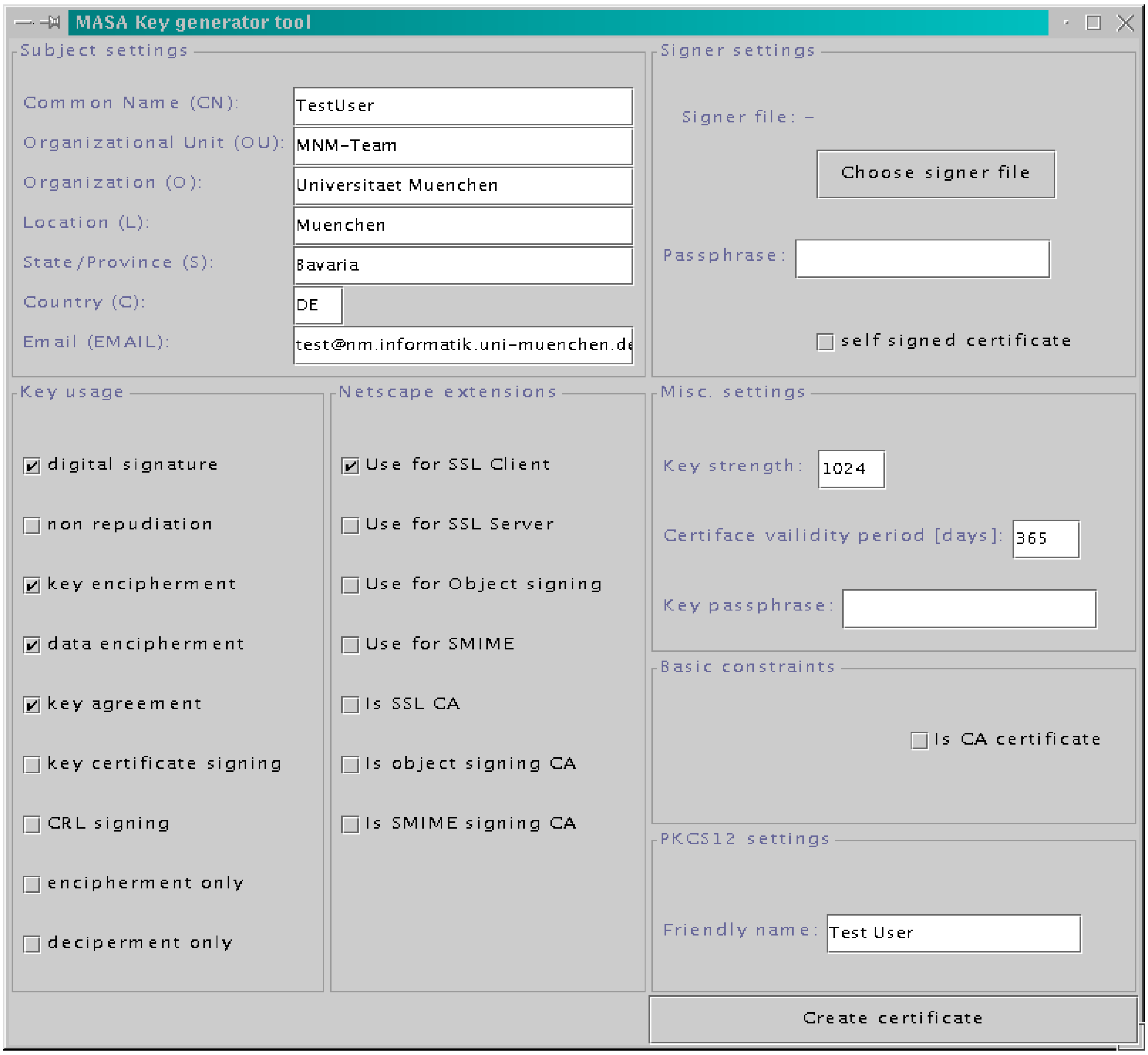

Abbildung 7.3: GUI von masa_keytool

- Erzeugung eines neuen Zertifikats für das Agentensystem, basierend auf dem Zertifikat der Authority (vgl. Kap. 5.5.4), und dessen paßwortgeschützte Speicherung.

- Paßwortabfrage beim Start des Agentensystems zur dechiffrierung des Agentensystemzertifikats.

- Generierung der Sitzungszertifikate von Agenteninstanzen (vgl. Kap. 5.5.4) und Erstellung einer ORB-Instanz, welche das Sitzungszertifikat benutzt.

- Erzeugung eines `SSLContext`-Objekts für die HTTPS-Kommunikation des Webserver Agenten.

- Erzeugung von Appletzertifikaten (vgl. Abschnitt 5.5.4).

- Überprüfung von Zertifikaten auf Vertrauenswürdigkeit.

Für die Zertifikatüberprüfung ist momentan noch keine *Authentication Policy* implementiert.

7.5.3 Asynchrone Schlüsselerzeugung

Bei der Erzeugung eines neuen Zertifikats ist die Generierung des Schlüsselpaares ein sehr rechenintensiver und damit auch zeitraubender Vorgang. Um eine Verzögerung bei der Erstellung einer neuen Agenteninstanz, verursacht durch die Erzeugung des Sitzungszertifikats, zu vermeiden, wurde die Klasse `agentSystem.AgentSystemKeyCache` implementiert. Sie erzeugt mit dem Start des Agentensystems neue Schlüsselpaare auf Vorrat.

Hierzu wird ein niedrig priorisierter Thread gestartet, der asynchron zur Klasse `agentSystem.AgentSystemCertManager` solange neue Schlüsselpaare generiert, bis eine (konfigurierbare) Anzahl von Schlüsseln auf Vorrat in einem Cache bereitstehen. Die Klasse `agentSystem.AgentSystemCertManager` holt dann bei Bedarf ein neues Schlüsselpaar von `agentSystem.AgentSystemKeyCache` ab und kann damit unmittelbar ein neues Sitzungszertifikat ausstellen. Nachdem ein Schlüsselpaar aus dem Cache entnommen wurde, wird dieser durch den Berechnungsthread wieder aufgefüllt. Stehen keine Schlüsselpaare mehr im Cache zur Verfügung, blockiert der Aufruf zum Abholen eines neuen Schlüsselpaares, bis ein neues erzeugt worden ist.

7.6 Laufzeitumgebung für Agenteninstanzen

Die Einführung der neuen Klasse `agentSystem.AgentEnvironment` und die Umgestaltung der Klassen `agent.Agent`, `agent.MobileAgent` und `agent.StationaryAgent` wurde exakt nach dem in Kap. 6.3.1 vorgestellten Schema durchgeführt. Weiterhin wurden die in Kap. 6.3.2 beschriebenen *Thread Groups* für Agenteninstanzen realisiert.

Der Classloader für Agenteninstanzen wurde mit der Klasse `agentSystem.AgentClassLoader` realisiert. Der Klassen-Code wird mittels `agentSystem.AgentCodeRepository` geladen, dabei wird der geladene Code mittels der Signatur des Implementierers auf Veränderungen überprüft. Um das in Kap. 6.4 und Kap. 6.7.2 beschriebene Laden "fremder" bzw. unerlaubter Klassen verhindern zu können, fragt `agentSystem.AgentClassLoader` vor jedem Ladevorgang beim *Access Controller* an, ob die Definition bzw. Benutzung der entsprechenden Klasse gestattet ist (vgl. Abschnitt 7.7.1). Als zusätzliches Sicherheitsmerkmal führt der Classloader beim Laden der Hauptklasse einer Agenteninstanz einfache Plausibilitätsüberprüfungen über das Reflection-API durch. Beispielsweise wird nachgeprüft, ob die Hauptklasse von einer der Basisklassen `agent.MobileAgent` oder `agent.StationaryAgent` abgeleitet ist.

Für die Basisklassen von Applets wurde das eigene Subpackage `agentApplet` angelegt. Die Klasse `agentApplet.AgentApplet` wurde völlig überarbeitet und unterstützt jetzt:

- Einen SSL-fähigen ORB
- Zertifikaterteilung nach Abschnitt 7.8.3
- EventChannels nach [Gerb 99]

Um die dynamische Erzeugung der zu den Applets gehörenden HTML-Seiten zu ermöglichen wurde die Klasse `agentApplet.AgentAppletConstants` implementiert, die die Maße (Höhe/Breite) des Applets enthält. Jede Agentengattung muß eine Subklasse davon ableiten.

7.7 Autorisierung

In der vorliegenden Implementierung konnte der *Permission Manager* nicht vollständig implementiert werden, da noch keine Spezifikation der konkreten Policy Sprache für die Permission Policy vorliegt. Entsprechend können momentan keine dynamischen Entscheidungen anhand der Identität oder anderer Attribute der zu autorisierenden Entität durchgeführt werden.

Stattdessen wurde die Standard-Implementierung der Java System Policy benutzt. Somit konnte nur die Autorisierung anhand der Gattung einer Agenteninstanz realisiert werden.

7.7.1 Erweiterung von Java Permissions

Zur Demonstration der Möglichkeit der Implementierung eigener Permission-Klassen wurden vier Klassen implementiert. Diese bestimmen, ob eine Agenteninstanz berechtigt ist, Klassen zu definieren bzw. sie zu benutzen. Diese Permissions werden durch `agentSystem.AgentClassloader` ausgewertet.

`agentSystem.ClassDenyUsePermission`: Legt fest, daß eine Klasse bzw. eine ganze Package-Hierarchie nicht benutzt werden darf.

`agentSystem.ClassAllowUsePermission`: Erlaubt die Benutzung einer Klasse bzw. einer ganzen Package-Hierarchie

`agentSystem.ClassDenyDefinePermission`: Legt fest, daß eine Klasse bzw. eine ganze Package-Hierarchie nicht definiert werden darf.

`agentSystem.ClassAllowDefinePermission`: Erlaubt die Definition einer Klasse bzw. einer ganzen Package-Hierarchie

Der Einsatz dieser Permission-Klassen soll nun anhand des Beispiels der SNMP-Bibliothek aus 6.7.2 erläutert werden. Hier werden zwei Fälle unterschieden:

1. Die SNMP-Bibliothek `com.one_company.snmp` ist Teil der Agentengattung com.other_company.SNMP_Agent, d. h. die Agenteninstanz muß die Bibliothek in ihrem Namensraum definieren dürfen:

   ```
   grant codeBase "systemresource://com/other_company/SNMP_Agent"
   {
       permission ClassAllowDefinePermission "com.one_company.snmp.-"
       permission ClassAllowUsePermission "com.other_company.snmp.-"
   }
   ```

2. Die SNMP-Bibliothek `com.one_company.snmp` ist Teil der Installation des Agentensystems, d. h. die Agenteninstanz darf die Bibliothek zwar benutzen, diese aber nicht neu definieren:

   ```
   grant codeBase "systemresource://com/other_company/SNMP_Agent"
   {
       permission ClassDenyDefinePermission "com.one_company.snmp.-"
       permission ClassAllowUsePermission "com.other_company.snmp.-"
   }
   ```

7.7.2 Überwachung der Endsystemschnittstelle

Jeder Agenteninstanz wird eine eigene Protection Domain zugewiesen. Die Code Source der Protection Domain, mit der diese im System Policy File adressiert wird lautet:

`systemresource://`<*gattungsname*>

Dabei ist <*gattungsname*> das Package der Agentengattung.

Über die Java System Policy können nun Rechte über beliebige Permission-Objekte vergeben werden. Soll nun beispielsweise allen Agentengattungen aus dem Package `de.unimuenchen.informatik.mnm.masa.agent` das Recht eingeräumt werden, die Property `testProperty` auszulesen, ist folgende Zeile in das System Policy File einzufügen:

```
grant codeBase "systemresource://de/unimuenchen/informatik/mnm/masa/agent/-"
{
    permission java.util.PropertyPermission "testProperty", "read";
}
```

7.7.3 Überwachung der CORBA-Methoden

Die CORBA-Methoden des Agentensystems werden überwacht, indem zu Beginn jeder Methode der Permission Manager befragt wird, ob die Aktion erlaubt werden darf oder nicht (vgl. 6.7.2).

Eine Überwachung der Schnittstellen der Agenteninstanzen konnte nicht implementiert werden, da Orbacus hierzu keine Schnittstellen bereitstellt (vgl. 7.1.2, 6.7.2).

7.8 Webserver Agent

Um den HTTP-Kanal sichern zu können wurde der Webserver Agent komplett neu implementiert. Dabei wurde die Protokollabwicklung und die Anfragebearbeitung auf verschiedene Klassen aufgeteilt (vgl. Abbildung 7.4).

Die neue Implementierung des Webserver Agenten unterstützt das HTTP Protokoll sowohl über "blanke" TCP-Ströme als auch über SSL-Ströme. Die Klassen HttpServer und HttpsServer implementieren die jeweiligen Varianten, die Klasse RequestHandler ist dann für die eigentliche Bearbeitung der Anfrage zuständig.

7.8.1 HTTP über TCP/IP und SSL

Die beiden Protokolle HTTP und HTTPS[5] werden von den beiden Klassen `HttpServer` und `HttpsServer` bereitgestellt. In Abhängigkeit von Betriebsmodus des Agentensystems (vgl. Abschnitt 7.11) wird beim Start des Agentensystems eine Instanz der jeweiligen Klasse erzeugt.

Beide Klassen erzeugen eigene Threads, in denen für das jeweilige Protokoll ein `ServerSocket` Objekt erstellt wird, welches auf eingehende Verbindungen wartet.

[5]Streng genommen ist HTTPS kein eigenständiges Protokoll, sondern nur die Kombination von HTTP mit SSL-Strömen

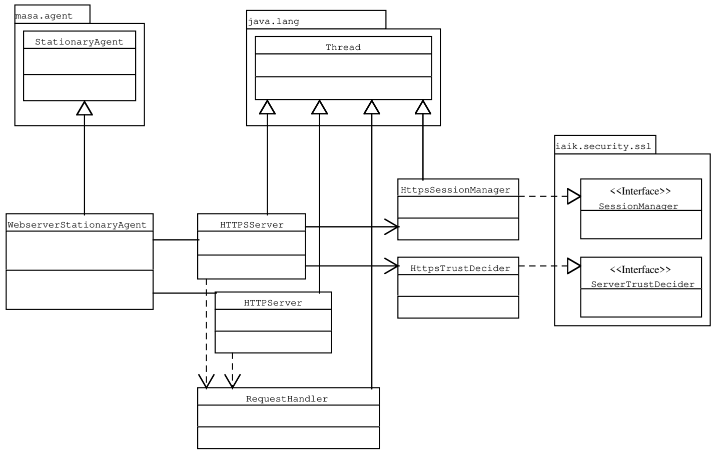

Abbildung 7.4: Klassendiagramm des Webserver Agenten

Wird eine eingehende Verbindung erkannt, so wird eine neuer TCP-Strom zum Anfragenden aufgebaut. Das weitere Vorgehen unterscheidet sich zwischen den beiden Protokollen.

HTTP

Für das HTTP-Protokoll wird einfach ein `java.net.Socket`-Objekt erzeugt, und eine neue Instanz von `RequestHandler` übergeben.

HTTPS

Beim HTTPS Protokoll erfolgt dann der Zertifikataustausch. Das dabei benutzte Server-Zertifikat erhält der Webserver Agent vom `AgentSystemCertManager`. Sind die Zertifikate ausgetauscht und bis dahin kein Protokollfehler aufgetreten, dann wird das Client-Zertifikat mit der Klasse `HttpsTrustDecider` auf seine Gültigkeit überprüft. War dieser Schritt ebenfalls erfolgreich, so wird ein neues `RequestHandler` erzeugt und daran ein `SSLSocket`-Objekt übergeben.

Um nicht bei jeder HTTPS-Anfrage von neuem die gesamte Prozedur des Zertifikataustauschs und der Zertifikatüberprüfung durchführen zu müssen, werden mit der Klasse `HttpsSessionManager` einfache SSL-Sessions unterstützt.

7.8.2 Anfragebearbeitung

Objekte von Typ RequestHandler übernehmen die protokollunabhängige Anfragebearbeitung des Webserver Agenten.

Um die dezentrale Verwaltung von Agenten-Applets und agentenspezifischen HTML-Seiten deutlich zu machen wurde das alte URL Schema des Webserver Agenten durch das folgende abgelöst:

- `GET /` Anfrage nach dem AgentSystemApplet.
- `GET /ImplementedAgents` Anfrage nach einer Liste der zur Verfügung stehenden Agentengattungen ([Gerb 99]).

- `GET /`<*`filename`*> Anfrage nach einer Datei aus dem `HTGUI/`-Verzeichnis des Agentensystems.
- `GET /`<*`agentname`*>`/agentapplet` Anfrage nach dem Applet eines Agenten. Die HTML-Seite, die das Applet einbettet, wird dabei dynamisch erzeugt.
- `GET /`<*`agentname`*>`/`<*`certID`*>`.appletcert` Anfrage nach einem Applet-Zertifikat (vgl. Abschnitt 7.8.3).
- `GET /`<*`agentname`*>`/ior` Anfrage nach der IOR einer Agenteninstanz (vgl. [Gerb 99]).
- `GET /`<*`agentname`*>`/url` Anfrage nach der URL einer Agenteninstanz (vgl. [Gerb 99]).
- `GET /`<*`agentname`*>`/.../`<*`filename`*> Anfrage nach einer Datei aus dem `HTGUI/`-Verzeichnis der Agentengattung.

7.8.3 Applet-Zertifikate

Für die CORBA/IIOP-Kommunikation zwischen dem Applet und der Agenteninstanz wird ein ORB-Objekt im Applet benötigt. Damit dieses ORB-Objekt SSL-Verbindungen benutzen kann, benötigt es wiederum ein Zertifikat. Nach dem Sicherheitsmodell muß dies das Zertifikat des Anwenders sein, der das Applet bedient.

Nun ist es dem Applet allerdings nicht möglich, das im Browser vorliegende Anwenderzertifikat auszulesen, da dies einer Kompromittierung des Zertifikats gleichkäme. Deshalb wurde folgender Mechanismus implementiert, mit dem das Applet mit einem eigenen Zertifikat versorgt wird, welches aus dem Anwenderzertifikat abgeleitet ist:

1. Der Anwender fordert am Webbrowser über eine HTTPS-Verbindung ein Applet an. Dabei wird das Zertifikat des Anwenders übermittelt.
2. Der Webserver Agent fordert vom `AgentSystemCertManager` die Erstellung eines Applet-Zertifikats aus dem Anwenderzertifikat an. Dabei werden die Daten des Anwenderzertifikats in ein neues Zertifikat übertragen und dieses dann vom Agentensystem signiert. Das neue Zertifikat wird, mit einer ID-Nummer versehen, vom `AgentSystemCertManager` zwischengespeichert und die ID-Nummer an den Webserver Agenten zurückgegeben.
3. Der Webserver Agent erstellt eine HTML-Seite, in der ein Verweis auf den Applet-Code und die ID-Nummer des Zertifikats eingebettet sind. Hierzu werden die HTML-Tags `APPLET` und `PARAM` benutzt. Die HTML-Seite wird an den Webbrowser über die HTTPS-Verbindung geschickt.
4. Der Webbrowser lädt über die HTTPS-Verbindung den Applet-Code und startet das Applet.
5. Das Applet fordert das Zertifikat über die in der HTML-Seite eingebetteten ID-Nummer vom Weberserveragenten an.
6. Der Webserver Agent fordert das Zertifikat vom `AgentSystemCertManager` an und sendet es an das Applet.
7. Das Applet erzeugt mit dem empfangenen Zertifikat eine neue SSL-fähige ORB-Instanz.

Damit verfügt dann das Applet über ein eigenes Zertifikat, das es für die SSL-Verbindungen benutzt und aus dem der Anwender des Applets ersichtlich ist.

7.9 CORBA Services

Die CORBA-Kanäle zu den in 3.5 bereits angesprochenen und von MASA benötigten CORBA-Services sind durch SSL gesichert. Sowohl der Naming Service, als auch der Event Service verwenden jeweils eigene Zertifikate. Diese sind im `certs/` Verzeichnis des MASA-Installtionsverzeichnisses abgelegt (vgl. Anhang A).

Auf eine Implementierung der in Kap. 6.10 vorgeschlagenen Security-Proxy-Objekte wurden momentan verzichtet. Von seiten des MASA-Basissystems werden beide Services z. Zt. nicht authentisiert. Weiterhin verwendet der eigene Webserver zur Bootstrapping-Unterstützung des Naming Service ungeschützte HTTP-Verbindungen.

7.10 Kompatibilität zu alten MASA Version

Wegen der Umfangreichen Änderungen an den Basisklassen für Agenten ist die neue MASA-Implementierung nicht voll kompatibel zur alten Version. Agenten, die für die alte MASA-Version erstellt wurden können nicht unmittelbar auf der Version ausgeführt werden.

Die Portierung auf die neue MASA-Version erfordert in der Regel aber nur einige kleine Änderungen an den Quelltexten. Außerdem sollte die neue Produktionsumgebung verwendet werden, mit der die Erstellung von signierten JAR-Dateien unterstützt wird.

Außerdem können Agentensysteme, die auf der neuen MASA-Implementierung beruhen nicht mit alten Agentensystemen in einem SmA kooperieren, da die alten Agentensysteme keine Kanalsicherung besitzten, und diese zwingend erforderlich ist.

7.11 Betriebsmodi des Agentensystems

Die Implementierung so gestaltet, daß die Teile der Sicherheitsimplementierung die auf JCE basieren, d.h. sich auf "harte" Kryptographie abstützen, abschaltbar sind. Die im Standard-Java-API vorhandenen Sicherheitsmechanismen bleiben jedoch intakt. Dadurch können im "unsicheren" Modus die folgenden Sicherheitsmerkmale noch verwendet werden:

- Getrennte Ausführungsumgebungen
- Signatur der JAR-Dateien der Agentengattungen, also Identifizierung des Implementierer.
- Überprüfung der Integrität der JAR-Dateien.
- Adressierung der Gattungen in Policies.
- Autorisierung der Endsystemschnittstelle auf Gattungsbasis
- Autorisierung der Agentensystemschnittstelle auf Gattungsbasis

Die Sicherung der Kanäle, die Autorisierung von Agentensystemen, Agenteninstanzen und Personen kann wegen der fehlenden Unterstützung für Zertifikate nicht erfolgen.

7.12 Zusammenfassung

Mit der neuen MASA-Implementierung sind wesentliche Sicherheitsmerkmale realisiert worden:

- Trennung der Ausführungsumgebungen der Agenteninstanzen, abschottung vom Agentensystem
- Sicherung aller Kanäle
- Code Repositories
- Durchgängige Authentisierung von
  - Personen
  - Agentensystemen
  - Agentengattungen
- Autorisierung der Schnittstellen von Agentensystem und Endsystem auf Gattungsbasis

Mit der Portierung der Implementierung auf JDK 1.2 und einen SSL-fähigen ORB wurde eine solide Basis für weitere Entwicklungen geschaffen.

Die folgenden Punkte des Sicherheitsmodells sind noch nicht verwirklicht worden:

- Einbindung einer PublicKey-Infrastruktur mit Key-Server und Revocation-Listen
- Eigene Policy-Sprache
- Capabilities zur Formulierung persistenter Rechte
- Delegation von Capabilities
- Autorisierung der CORBA-Methoden von Agenten durch das Agentensystem

Kapitel 8

Richtlinien zur Agentenimplementierung

In diesem Kapitel sollen einige Hinweise gegeben werden, die für die Implementierung einer Agentengattung beachtet werden sollten.

Ein guter Ausgangspunkt für jede Implementierung ist dabei die FOO-Agentengattung. Diese Gattung wurde speziell zu Demonstrationszwecken erstellt und bietet eine gute Basis, da hilfreiche Details zur Programmierung und dem Umgang mit der Produktionsumgebung in der Distribution enthalten sind.

Den Applets muß bei der Implementierung besondere Aufmerksamkeit geschenkt werden. Nach Kap. 5.5.1 können Applets durch die Agenteninstanz nicht authentisiert werden. Wird nun durch ein Applet eine Aktion angefordert, so kann nicht überprüft werden, ob die Anforderung tatsächlich vom "eigenen" Applet kommt oder von einem anderen, feindliche Absichten hegenden Applet versendet wurde. Für den naheliegenden Gedanken, Autorisierungsentscheidungen auf einem Applet zu treffen, heißt das, daß nicht gewährleistet werden kann, daß die Autorisierung tatsächlich durchgeführt wurde.

Einem Beispiel: Eine Agenteninstanz biete zwei Methoden M und X an, die beide durch das Applet dargestellt werden. Dabei solle X nur von bestimmten Anwendern genutzt werden dürfen. Wird die Überprüfung, ob X vom aktuellen Anwender ausgeführt werden darf, nur auf dem Applet durchgeführt, so entsteht ein Sicherheitslücke. Ein Angreifer kann nämlich ein anderes Applet benutzen, das die Überprüfung nicht durchführt, ohne daß dies von der Agenteninstanz bemerkt werden könnte.

Deshalb: Jegliche Autorisierungen immer durch die Agenteninstanz durchführen, niemals (alleinig) durch das Applet.

Nachfolgend einige weitere Richtlinien, die bei der Implementierung zu beachten sind:

- Kritische Berechnungen möglichst nur auf "sicheren" Agentensystemen durchführen.
- Kritische Daten nach Möglichkeit nur verschlüsselt, zumindest aber signiert ablegen.
- Eine genau Spezifikation anfertigen, welche Aktionen auf dem Endsystem ausgeführt werden, welche Klassen hierfür benötigt werden und mit welchen anderen Agenten wie kooperiert wird.

- Benutzung der eigenen CORBA-Schnittstelle immer autorisieren.
- Keine öffentlich zugreifbaren Java-Schnittstellen schaffen, besonders nicht in Klassen, die `public` deklariert sein müssen, da sie CORBA-Funktionen implementieren.
- Agentengattungen sollten nach Möglichkeit in genau einem Java-Package implementiert werden, das ermöglicht die optimale Nutzung von Visibilitätsmodifikatoren:
- Klassen, die nicht abgeleitet werden sollen, `final` deklarieren.
- Klassen, die nur von der eigenen Agentengattung benötigt werden, `package` deklarieren.
- Methoden, die nicht überladen werden sollen, `final` deklarieren.
- Methoden, außer jene, die CORBA-Funktionen implementieren, möglichst nie `public` deklarieren.
- Methoden, die nur von der eigenen Agentengattung benötigt werden, `package` deklarieren.
- Methoden, die nur in der deklarierenden Klasse benötigt werden, `private` deklarieren.
- Attribute, deren Werte nur einmalig im Constructor der Klasse gesetzt werden sollen, `final` deklarieren.
- Attribute immer `private` deklarieren.
- Wenn auf Attribute außerhalb einer Klasse zugegriffen werden muß, Zugriffsmethoden (`get()`/`set()`-Methoden) bereitstellen, dabei auf Visibilitätsmodifikatoren achten. Überprüfungen auf semantische Korrektheit in der `set()`-Methode durchführen.

Kapitel 9

Zusammenfassung und Ausblick

Mit dem vorgestellten Sicherheitsmodell für MASA werden Mechanismen zur Authentisierung und Autorisierung in Systemen mobiler Agenten geschaffen, sowie die Rahmenbedingungen erläutert, die diese sinnvoll ermöglichen. Damit konnten die primären Sicherheitslücken von MASA geschlossen werden:

- Getrennte Ausführungsumgebungen der Entitäten werden festgelegt und implementiert.
- Alle Kommunikationskanäle in MASA werden gesichert.
- Mechanismen zur Authentisierung aller wesentlichen Entitäten von MASA werden spezifiziert und implementiert. Diese basieren durchgängig auf Public-Key-Verfahren und den zugehörigen Zertifikathierarchien.
- Ein Konzept zur Unterscheidung von statischen und dynamischen Informationen eines Agenten wird angegeben und implementiert.
- Charakteristische Attribute von Agenteninstanzen werden gesichert und somit vor Veränderung (im Rahmen eines Angriffs) geschützt.
- Die Benutzung jeglicher Schnittstellen wird in einem hierarchischen Entscheidungsprozeß autorisiert, dabei können die hierfür notwendigen Regeln flexibel und feingranular formuliert werden.
- Ein Konzept zu Delegation von Rechteinformationen wird angegeben.

Die Erweiterung der vorhergehenden MASA-Implementierung stützt sich dabei auf etablierte Techniken wie SSL, X.509-Zertifikate und Java 1.2. Damit ist ein Fortbestand der Implementierungsbasis als höchstwahrscheinlich anzusehen. Durch die Weiterentwicklung der diesen Techniken zugrundeliegende Standards wird auch MASA davon profitieren. Weiterhin besteht die Möglichkeit MASA in vorhandene Strukturen einzubinden (z.B. CA-Strukturen) und verfügbare Werkzeuge zu Nutzen.

Ausblick

Mit dem vorgestellten Sicherheitsmodell ist eine Basis für eine umfassende Sicherheitsarchitektur gelegt worden. Für zukünftige Arbeiten lassen sich drei Kategorien identifizieren:

1. Ergänzungen an der Implementierung:
    - Portierung bestehender Agenten auf die neue MASA-Implementierung.

- Vervollständigung der Implementierung des Sicherheitsmodells im MASA-Basissystem, darunter:
  - Einbindung einer Public-Key-Infrastruktur mit Key-Server und Revocation-Listen. Hierzu wird gerade Systementwicklungsprojekt durchgeführt.
  - Capabilities, nach Kap. 6.9.
- Einbeziehung des Naming Service und des Event Service in die sichere Implementierung. Hierzu sollte ein generisches Framework für Security-Proxy-Objekte nach dem Konzept aus Kap. 6.10 erstellt werden und anschließend für beide Services entsprechende Proxy-Objekte implementiert werden. Außerdem sollte der Bootstrapping-Webserver des Naming Service auf die Verwendung von SSL-Strömen umgestellt werden.

2. Für eine Fortschreibung des vorgestellten Konzepts lassen sich die folgende Schritte ausmachen:
   - Konkrete Spezifikation einer Policy-Sprache nach den in Kap. 5.8.3 gestellten Anforderungen.
   - Realisierung der Policy-Sprache und Einbindung in die Java-Sicherheitsarchitektur nach Kap. 6.7.1.
   - Erstellung einer GUI zur benutzerfreundlichen Wartung und Spezifikation von Security Policies.
   - Konkrete Spezifikation eines Domänen/SecurityPolicy-Managements. Dabei ist eine Untersuchung der folgenden Punkte denkbar:
     - Gruppenbildung über den in Kap. 5.10 entwickelten Domänenbegriff, daraus Erweiterung der Policy-Sprache um Gruppenidentifikatoren zur Adressierung der Gruppen als Erweiterung der in Kap. 5.8.3 vorgestellten Techniken zur Adressierung von Entitäten.
     - Möglichkeiten der Einbeziehung der in [Radi 98] entwickelten Konzepte und Agenten zur Durchführung des Domänen/SecurityPolicy-Managements
     - Erweiterung der MASA-Implementierung um *places* ([OMG 98-03-09]) als Repräsentation von Sicherheitsdomänen auf dem Agentensystem.
3. Als Fernziel für die Komplettierung des Modells ist die Lösung der Problematik der sicheren und nachvollziehbaren Ausführung der Agenten zu nennen.

Anhang A

Die neue MASA Produktions-Umgebung

Mit der neuen MASA Version wurde auch eine neue Produktionsumgebung eingeführt, die im folgenden kurz beschrieben werden soll.

A.1 Motivation

Im Umgang mit der vorherigen MASA Produktionsumgebung zeigten sich einige Nachteile:

- Einstellungen der zu verwendenden Werkzeuge (Compiler, Libraries, etc.) mußten für das Agentensystem und jeden Agenten getrennt von neuem getroffen werden. Neben dem fehlenden Komfort führte dies zu der Gefahr, daß beispielsweise Agentensystem und Agenten mit unterschiedlichen Compilern übersetzt wurden, was dann zur Laufzeit des Systems zu undurchsichtigen Fehlern führte.
- Keine automatische Unterstützung der verschieden am Lehrstuhl verfügbaren Betriebssystemplattformen. Es war nicht möglich, zur Übersetzung des Systems von einer Betriebssystemplattform (z.B. Solaris) auf eine andere (z.B. Linux) zu wechseln, ohne eine große Zahl von Pfadeinstellungen in allen Makefiles von Agentensystem und Agenten neu durchführen zu müssen.
- Keine Möglichkeit der konditionalen Übersetzung der Quelltexte. Das fallweise Ein-/Ausblenden von Quelltextteilen zur Übersetzung, beispielsweise für Debug-Code, war nur durch manuelles Ein-/Auskommentieren des entsprechenden Blocks möglich.

Ziel der neuen Produktionsumgebung war es, diese Unflexibilitäten zu beseitigen und eine komfortable Umgebung zu schaffen, die Entwickler von Agenten in die Lage versetzt, möglichst auf bereits für die Übersetzung des Agentensystem getroffene Einstellungen zurückgreifen zu können.

Erreicht wurde dies durch die Schaffung einer Reihe von Makefiles, deren Inhalte nach den folgenden Kriterien aufgeteilt wurde:

- Betriebssystemplattform-spezifische Einstellungen
- Tool-spezifische Einstellungen
- MASA-spezifische Einstellungen

Diese Makefiles werden gemeinsam von den Produktionsumgebungen von Agentensystem und Agenten benutzt.

Weiterhin sollte eine Möglichkeit geschaffen werden, mit der die konditionale Übersetzung unterschiedlicher Varianten aus einem Quelltext heraus möglich wird. Hierzu wurde das in [Roel 98] vorgestellte Konzept der Verwendung eines Präprozessors angewendet. Dort wird die damit verbundene Problematik anhand der Verwendung zweier unterschiedlicher ORB-Implementierungen, welche unterschiedliche Signaturen für Methoden gleichen Namens verwenden, erläutert:

> “ Für die flexible Verwendung des Quelltexts für beide ORBs ist die konditionale Übersetzung durch Steuerung über globale Konstanten (wie in Java üblich, siehe [Flan 97], Seite 22) nicht möglich. Der Grund hierfür liegt darin, daß bei dieser Methode die syntaktische Korrektheit aller Varianten überprüft werden muß, was durch die unterschiedlichen Signaturen zentraler Methoden und Klassennamen nicht möglich ist.
>
> Die Verwendung eines Präprozessors, wie er aus der Programmiersprache C bekannt ist, stellt hier einen Ausweg dar, da dieser die jeweiligen, nicht zur Übersetzung anstehenden Teile ausfiltert, bevor der Quelltext an den Übersetzer übergeben wird. Leider ist ein Präprozessor in Java nicht vorgesehen.
>
> Um dennoch einen Präprozessor nutzen zu können, mußte eine Eigenschaft des verwendeten Java Entwicklungssystems beachtet werden. Im Gegensatz zu C wird in Java eine automatische Auflösung von syntaktischen Abhängigkeiten auf Java-Ebene zwischen unterschiedlichen Quelldateien durch den Übersetzer vorgenommen und diese bei Bedarf auch sofort übersetzt. Somit ist eine, unter Unix gängige, “pipe”-Verkettung von Präprozessor und Java Compiler ausgeschlossen. Vielmehr müssen alle Quelldateien vor dem ersten Aufruf des Java Compilers den Präprozessor durchlaufen haben, da im Zuge des Übersetzungsvorgangs nicht nur die angegebene Datei betrachtet wird, sondern auch all jene, von denen diese Datei abhängig ist. Dies wiederum führt zu dem Problem, daß nun die Ergebnisse der Präprozessor-Läufe explizit in Form von Dateien vorliegen müssen, die, durch den Suchalgorithmus von Java bedingt, mit identischen Dateinamen in korrekter Hierarchie zu den Quelldateien vorliegen müssen.
>
> Wegen dieser Randbedingungen entstand eine Konstruktion, die zwischen den universellen Quelldateien und den Präprozessor-Ergebnissen unterscheidet. Die Präprozessor-Ergebnisse werden in einem sogenannten “Produktionsverzeichnis” abgelegt, in dem dann auch die eigentliche Übersetzung durch den Java Compiler durchgeführt wird und sich schließlich auch die übersetzten Programme befinden. ”

In [Roel 98] wird außerdem ein weiterer Vorteil der Verwendung eines Präprozessors beschrieben:

> “ Durch die Verwendung eines Präprozessors wurde außerdem eine weitere Automatisierungsstufe im Entwicklungsprozeß möglich. Gewöhnlich muß sich die Package-Hierarchie der Java-Klassen in der die Klassen implementierenden Datei-Hierarchie widerspiegeln, damit die automatische Auflösung von Abhängigkeiten durch den Java Compiler funktioniert. Durch den Präprozessor sind nun auch Textersetzungen möglich, was es erlaubt, die Package-Hierarchie zentral als Parameter festzulegen. Die Quelldateien selbst müssen deshalb weder die konkrete Spezifika-

> tion der Package-Hierarchie enthalten, noch ist eine Anordnung der Quelldateien entsprechend Package-Hierarchie notwendig. "

Diese Technik wird ebenfalls in der neuen Produktionsumgebung eingesetzt.

A.2 Die Umgebung des Agentensystems

A.2.1 Makefiles

Im Wurzelverzeichnis der Agentensystems befindet sich das Master-Makefile, das alle Aktionen initiert und koordiniert. Daneben werden in `Makefile.DEF` einige prinzipielle Einstellungen bezüglich einiger benötigter Verzeichnisse gemacht (siehe Tabelle A.1). Beide Dateien enthalten Einstellungen, die unspezifisch für Tools oder Betriebssystemplattform sind und ausschließlich für das Basis-System verwendet werden.

Im Verzeichnis `config/` befinden sich nun u.a. jene Makefiles, die werkzeug- oder plattformspezifische Einstellungen enthalten. Diese werden automatisch vom Master-Makefile eingebunden:

`Makefile.masaconf`: Enthält Basiseinstellungen des gesamten MASA Systems. Darunter auch in der Variable `MASA_PACKAGE` das Java Package, welches das Basis-Package des gesamten Agentensystems darstellt.

`Makefile.buildtools`: Definiert die für die Übersetzung und den Start des Agentensystem benötigten Tools. Betriebssystemplattform-spezifische Einstellungen werden aus den entsprechenden Unterverzeichnissen von `config/` eingebunden.

`Makefile.jcesetup`: Enthält Einstellungen für das verwendete JCE Toolkit.

`Makefile.orbsetup`: Tools und Einstellungen, die spezifisch für die verwendete CORBA Entwicklungsumgebung sind.

`Makefile.toolconfig`: Definiert die zu verwendenden Optionen aller Tools, die benötigt werden um MASA korrekt zu übersetzen und zu starten.

Dabei enthalten jene Dateien mit der Endung `.MNM` vordefinierte Einstellungen für die Arbeit am Lehrstuhl-Cluster. Diese werden automatisch benutzt, wenn an einem Rechner am Lehrstuhl gearbeitet wird. Für die Arbeit an Rechnern, die nicht im Lehrstuhl-Cluster eingebunden sind, werden die Pendants dieser Makefiles benutzt. Diese sind dann gegebenenfalls an die lokalen Gegebenheiten anzupassen.

Alle Makefiles in `config/` werden nicht nur für die Übersetzung des Basis-Systems benutzt, sie werden auch von den Produktionsumgebungen der Agenten eingebunden. D. h. daß alle Einstellungen bezüglich Compiler, Toolkits, etc. nur einmalig und zentral getroffen werden müssen, womit eine konsistente Übersetzungs- und Laufzeitumgebung für Basis-System und Agenten gewährleistet wird.

Neben den Makefile in `config/` befindet sich noch ein weiteres im Verzeichnis `prod.java/`, welches den eigentlichen Übersetzungsvorgang steuert.

Eine Aufstellung konkreter Kommandos, die das Master-Makefile unterstützt, ist in der Datei `INSTALL` enthalten.

A.2.2 Verzeichnisstruktur

Nachfolgend soll die zum Agentensystem gehörende Verzeichnisstruktur kurz beschrieben werden. Dabei sind Namen und Ort der einzelnen Unterverzeichnisse teilweise nicht fest eingestellt, sie werden vielmehr über `Makefile.DEF` festgelegt. Die Namen der Variablen, die dort belegt und wodurch die Datei- und Verzeichnisnamen festgelegt werden, sind Tabelle A.1 zu entnehmen.

Verzeichnisname	Makefile-Variable
`bin/`	-
`PRODUCTION.default/`	-
`src/`	`BUILD_SOURCE_PATH`
`config/`	`MASA_MAKE_CONFIG`
`prod.java/`	`PROD_DIR`
`install/`	`MASA_INSTALL_PATH`
`tmp/`	`BUILD_TMP_PATH`

Tabelle A.1: Makefile-Variablen für Verzeichnisnamen in `system/`

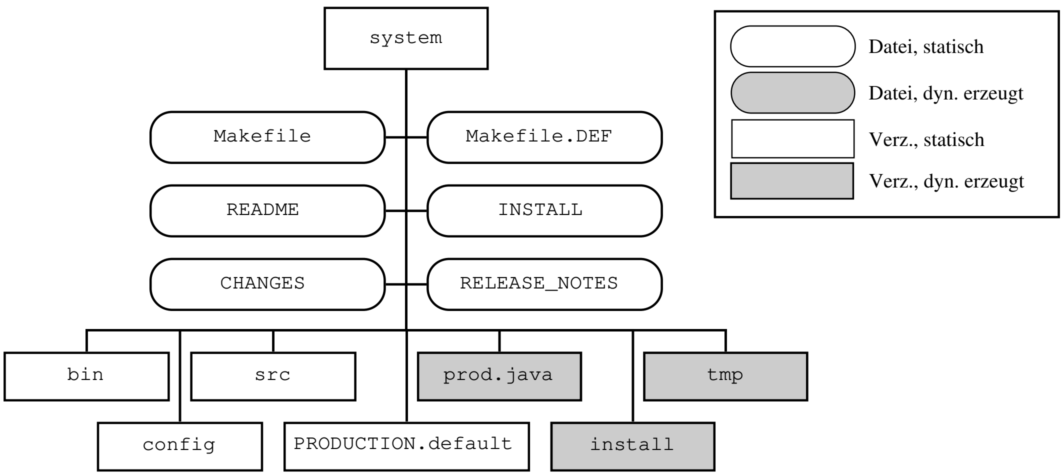

Abbildung A.1: Unterverzeichnisse der Agentensystem-Produktionsumgebung

Die Struktur, wie sie in A.1 dargestellt ist, entspricht den Voreinstellungen aus `Makefile.DEF`.

A.2.2.1 Dateien im Wurzelverzeichnis `system/`

`README`: Allgemeine Hinweise zum Basis-System.

`INSTALL`: Anleitung und Hinweise zur Übersetzung und Installation.

`RELEASE_NOTES`: Aufstellung über bekannte Probleme und versionsspezifische Anmerkungen

`CHANGES`: Informelle Liste der Änderungen zwischen verschiedenen Versionen.

`Makefile`: siehe Kapitel A.2.1.

`Makefile.DEF`: siehe Kapitel A.2.1.

A.2.2.2 Verzeichnis `bin/`

Hier sind Hilfsprogramme enthalten, die zur Übersetzung des Agentensystems, oder zur Erstellung spezieller Dateien benutzt werden. Diese sind meist als Shell-Skripten realisiert und greifen selbst wiederum auf Einstellungen zurück, die in den Makefiles getroffen wurden. Im einzelnen:

- `signAgentJar`: Hilfsprogramm zum signieren eine JAR Datei.
- `genPropertiesFile`: Generiert die Properties-Datei des Agentensystems.
- `genRunScripts`: Generiert die plattformspezifischen Start-Skripten für den Naming-Service, EventChannel-Service und das Agentensystem.
- `genMasterRunScript`: Generiert das plattformübergreifende Start-Skript, welches auf die mittels ***`genRunScripts`*** erzeugten Skripte zurückgreift.
- `getHostname.sh`: Hilfsprogramm zum Auflösen des symbolischen DNS-Namen `localhost` in den entsprechenden FQDN.
- `agentCompatCheck`: Führt eine grobe Überprüfung von Quelltexten auf häufig gemachte Fehler durch. Beispielsweise wird überprüft, ob der Standard-Package-Präfix "de.unimuenchen.informatik.mnm" als Klartext im Code enthalten ist. Dieser sollte durch die in Abschnitt A.2.3 vorgestellten Macros ersetzt werden.
- `genMasaBootPolicyFile`: Generiert die Java-Policy Datei der JVM des Agentensystems.

Die Vorlagen für die zu erzeugenden Dateien sind jeweils in den ***`gen...`***-Scripts selbst enthalten.

A.2.2.3 Verzeichnis `src/`

Dieses Verzeichnis enthält in einer Reihe von Unterverzeichnissen ausschließlich jene Quelltexte, die von Hand (also nicht durch Tools wie den IDL-nach-Java Übersetzer) erstellt wurden. Die Struktur in diesem Verzeichnis entspricht der relativen Java package-Hierarchie der Klassen ohne den gemeinsamen Präfix (vgl. Abbildung A.3).

Hier werden Änderungen an den Quelltexten vorgenommen.

A.2.2.4 Verzeichnis `config/`

Enthält eine Reihe von Makefiles, die in Kapitel A.2.1 näher erläutert werden.

A.2.2.5 Verzeichnis `PRODUCTION.default/`

Enthält Dateien die im CVS-Repository gespeichert werden sollen, aber noch nicht an ihrem endgültigen Bestimmungsort abgelegt werden können, da dieser erst später erzeugt wird. Beispiel: statische HTML-Seiten, die erst später in das `install/` Verzeichnis kopiert werden.

A.2.2.6 Das Produktionsverzeichnis `prod.java/`

In diesem Verzeichnis befinden sich alle Quelltexte, die durch den Präprozessor generiert oder durch Tools automatisch erstellt wurden, sowie die daraus übersetzten `.class`-Dateien. Dieses Verzeichnis wird ausschließlich zur Übersetzung des Agentensystems benutzt. Änderungen an den Quelltexten werden immer im `src/` Verzeichnis vorgenommen, ausgeführt wird das Agentensystem im `install/` Verzeichnis.

A.2.2.7 Verzeichnis `install/`

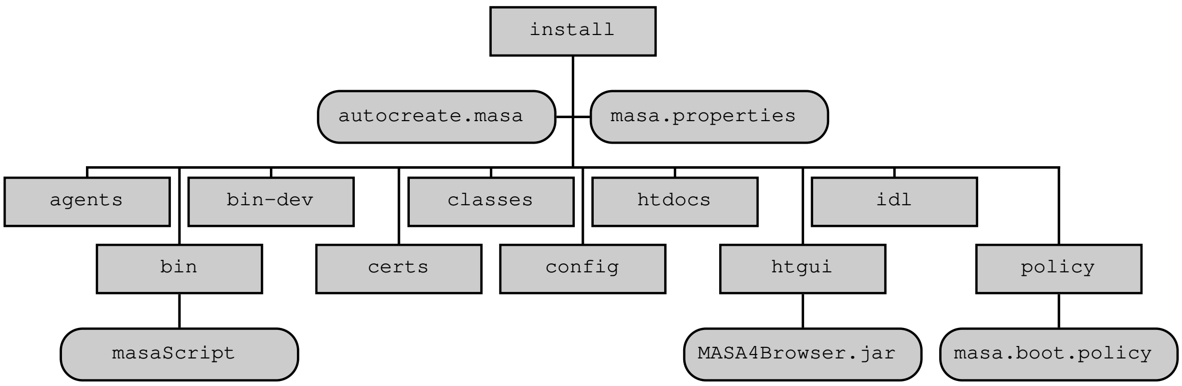

Abbildung A.2: Das `install/` Verzeichnis des Agentensystems

Das Verzeichnis `install/` enthält alle zur Ausführung des Agentensystems notwendigen Komponenten, mit Ausnahme der zum ORB, JCE und Swing gehörenden `.jar`-Dateien:

`masa.properties`: Enthält alle Agentensystem-spezifischen Java Properties, wurde durch `system/bin/`*`genPropertiesFile`* erzeugt.

`autocreate.masa`: Beschreibt, welche Agenten beim Start des Agentensystems automatisch erzeugt werden sollen, Kopie der Datei `autocreate.masa` dem Verzeichnis `system/PRODUCTION.default/install/`. Eine Beschreibung der Syntax dieser Datei entnehme man der *`javadoc`*-Dokumentation der Klasse `AutoCreateDescription`.

`config/`: Makefiles und Konfigurationen, die zur Implementierung und Übersetzung von Agenten benötigt werden. Kopie des Verzeichnisses `config/` aus `system/`.

`agents/`: Lokal installierte Agenten in Form von `.jar`-Dateien. Die in 7.4 eingeführte Klasse `LocalFilesystemCodeRepository` liest aus diesem Verzeichnis.

`bin/`: Ausführbare Dateien zum Starten des Agentensystems und zugehöriger Services. Erzeugt durch `system/bin/`*`genRunScripts`* und `system/bin/`*`genMasterRunScript`*.

`bin-dev/`: Ausführbare Dateien, die zur Implementierung und Übersetzung von Agenten benötigt werden.

`certs/`: Paßwortgeschützte Zertifikate und Schlüsselpaare die durch das Agentensystem zur Laufzeit erzeugt werden.

`classes/`: Java Klassen des Agentensystems.

`htdocs/`: Durch *javadoc* erzeugte Quelltext-Dokumentationen des Agentensystems und von Agenten.

`htgui/`: Dateien, die der Webserver Agent benötigt und dem Agentensystem zugeordnet sind. Besonders zu erwähnen ist hier die Datei `MASA4Browser.jar`, die alle Agentensystem-spezifischen Java Klassen enthält, die der Webbrowser benötigt, um das Applet eines Agenten ausführen zu können.

`idl/`: Agentensystem-spezifische `.idl` Dateien, die zur Implementierung und Übersetzung von Agenten benötigt werden.

`policy/`: Datei `masa.boot.policy`, die Policy-Datei der JVM des Agentensystems.

Alle Datei und Verzeichnisnamen sind nicht fest vergeben und können in `system/config/Makefile.masaconf` eingestellt werden. Die hierzu benutzten Variablen sind der Tabelle A.2 zu entnehmen. Weiterhin gibt die Tabelle Aufschluß darüber, in welchen Java Properties die entsprechenden Dateien und Verzeichnisse dem Agentensystem bekannt gegeben werden.

Dateiname	Makefile-Variable	Property (ohne Präfix)
masa.boot.policy	MASA_INSTALL_BOOT_POLICY_FILE	via *java* Option: -Djava.security.policy==[1]
masa.properties	MASA_INSTALL_PROPERTYFILE	via *java* Option: -Dmasa.propfile=
autocreate.masa	MASA_INSTALL_AUTOCREATEFILE	autocreateFile
MASA4Browser.jar	MASA_INSTALL_BROWSERBASEJARFILE	browserBaseJarFile
Verzeichnisname	Makefile-Variable	Property (ohne Präfix)
config/	-	-
agents/	MASA_INSTALL_AGENTS	agentCodeBase
bin/	MASA_INSTALL_BIN	-
bin-dev/	MASA_INSTALL_BINDEV	-
certs/	MASA_INSTALL_CERTS	certsPath
classes/	MASA_INSTALL_CLASSES	agentSystemCodeBase
htdocs/	MASA_INSTALL_HTDOCS	-
htgui/	MASA_INSTALL_HTGUI	htmlCodeBase
idl/	MASA_INSTALL_IDL	-
policy/	MASA_INSTALL_POLICY	-

Tabelle A.2: Konfigurierbare Dateinamen und zugehörige Makefile-Variablen

A.2.3 Produktionsprozeß

Soll das Agentensystem übersetzt werden, so werden zunächst alle im Verzeichnis `src/idl/` enthaltenen IDL-Quelltexte durch den IDL-nach-Java Übersetzer in die entsprechenden Java-Quelltexte übersetzt, welche im Verzeichnis `prod.java/` abgelegt werden.

Anschließend werden die Java-Quelldateien aus `src/`, gesteuert von den sich dort ebenfalls befindenden Makefiles, durch den Präprozessor gefiltert. Die erzeugten Dateien werden dann im Verzeichnis `prod.java/` entsprechend der kompletten package-Hierarchie abgelegt. Diese setzt sich dabei aus dem in `Makefile.masaconf/`, in der Variablen `__MASA_PACKAGE__` gesetzen

Basis-package und der relativen Hierarchie, wie sie durch die Anordnung in src gegeben ist, zusammen. Abbildung A.3 stellt diesen Vorgang dar.

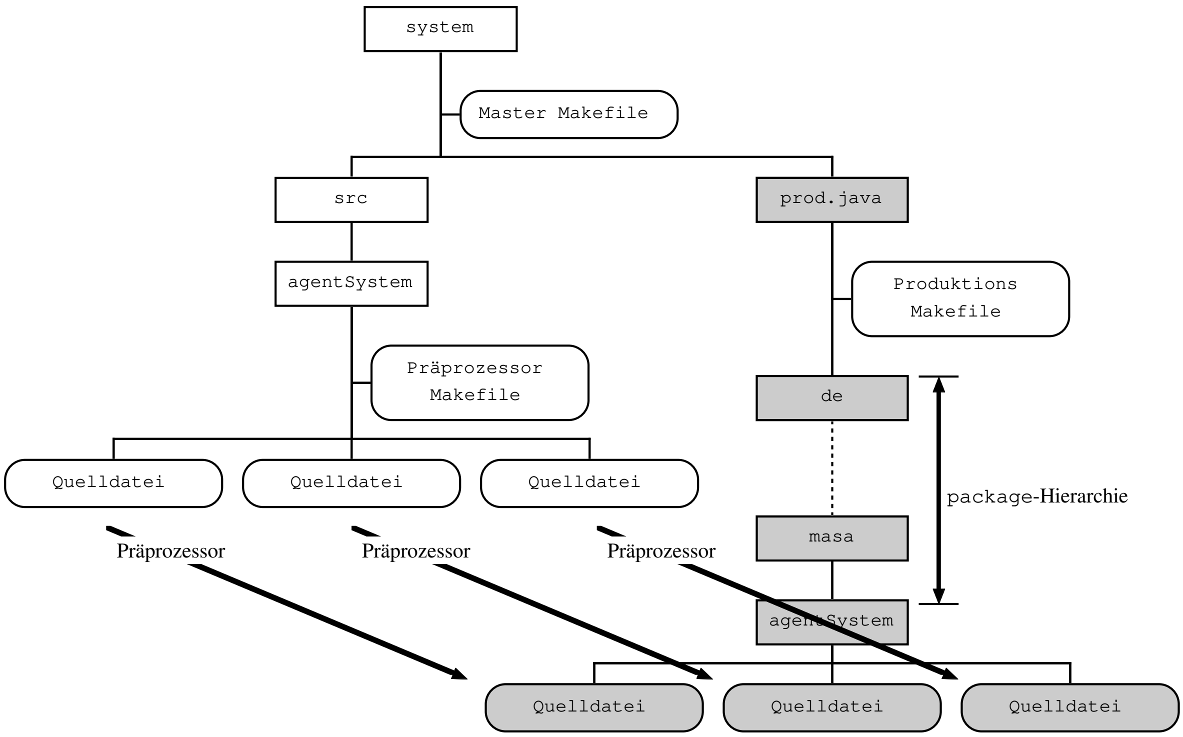

Abbildung A.3: Dynamische Anordnung entsprechend der package-Hierarchie (nach [Roel 98])

Bei der Bearbeitung durch den Präprozessor werden im Fall von MASA hauptsächlich zwei Schritte durchgeführt:

- Konditionales Einfügen von Quelltextteilen in die zu übersetzende Datei:

  Quelltextblöcke, die durch ein `#ifdef <ident>/#endif` Konstrukt gekapselt sind, werden nur dann eingefügt, wenn ein Wert `<ident>` gesetzt ist.

  So wird beispielsweise das Fragment

  ```
  #ifdef ORBACUS311
      agentSystemService = new _AgentSystemServiceImplBase_tie( agentSystem);
      boa.obj_is_ready( agentSystemService, null);
  #endif
  ```

  nur dann ausgegeben, wenn in `Makefile.masaconf` durch die Zeile

  ```
  export ORB_TYPE := ORBACUS311
  ```

  und der Definition

  ```
  export CPP_OPT := -C -P -D$(ORB_TYPE)
  ```

  in `Makefile.toolconfig` dieser Wert für den Präprozessor definiert wurde.

- Textersetzung von sog. Macros, die durch Makefiles vordefiniert werden:

  Quelltextteile, die im Präprozessor als Macro definiert sind, werden durch den im Macro definierten Wert ersetzt.

Beispielsweise wird mittels der Macro-Definition (aus `Makefile.toolconfig`)

```
export CPP_PACKAGE_MACROS := ...
                             "-D__MASA_PACKAGE__(x)=$(MASA_PACKAGE).\#\#x"
                             ...
```

und der Definition (aus `Makefile.masaconf`)

```
export MASA_PACKAGE := de.unimuenchen.informatik.mnm.masa
```

das Quelltextstück

```
package __MASA_PACKAGE__(tools);
```

als

```
package de.unimuenchen.informatik.mnm.masa.tools;
```

ausgegeben, womit die oben angesprochene dynamische Festlegung des Java Basis-Packages im Java Quelltext erfolgt.

Abschließend werden, gesteuert durch das Makefile in `prod.java/`, alle dort enthaltenen Quelldateien übersetzt.

Abkürzungsverzeichnis

API Application Programm Interface

CORBA Common Object Request Broker Architecture

CRL Certificate Revocation List

CSA Client/Server Architektur

CVS Concurent Versioning System

DN Distinguished Name

DNS Domain Name Service

GUI Graphical User Interface

HTTPS HTTP über SSL

IDL Interface Definition Language

JAR Java ARchive

JCA Java Cryptographic Architecture

JCE Java Cryptography Extension

LAN Local Area Network

NAT Network Address Translation

OOC Object Oriented Concepts

ORB Object Request Broker

QoS Quality of Service

SLA Service Level Agreement

SmA System mobiler Agenten

SNMP Simple Network Management Protocol

UML Unified Modelling Language

URL Universal Resource Locator

VPN Virtual Private Network

Literaturverzeichnis

[Aglet Site] *http://www.trl.ibm.co.jp/aglets/*.

[Baue 97] Bauer, F. L.: *Entzifferte Geheimnisse — Methoden und Maximen der Kryptologie*. Springer, Zweite Auflage, 1997.

[Bran 99] Brandt, R.: *Evaluierung der Interoperabilität von CORBA ORBs am Beispiel eines Gateways für die Mobilen Agentensysteme MASA und Voyager*. Systementwicklungsprojekt, Technische Universität München, 1999.

[Ches 98] Chess, D. M.: *Security Issues in Mobile Code Systems*. In: Vigna, G. [Vign 98], Seiten 1–14.

[CORBA 2.2] *The Common Object Request Broker: Architecture and Specification*. OMG Specification Revision 2.2, Object Management Group, Februar 1998.

[DiAl 99] Dierks, 0. T. und C. Allen: *RFC 2246: The TLS Protocol Version 1*. RFC, IETF, Januar 1999.

[FKK 96] Freier, A. O., P. Karlton und P. C. Kocher: *The SSL Protocol Version 3.0*. Spezifikation, Netscape Communications Corporation, März 1996.

[Flan 97] Flanagan, D.: *Java in a Nutshell*. O'Reilly, Zweite Auflage, Mai 1997.

[Gerb 99] Gerber, S.: *Entwicklung einer Benutzerschnittstelle mit Java/CORBA für die Mobile Agent System Architecture (MASA)*. Systementwicklungsprojekt, Technische Universität München, 1999.

[Gh Basics] *Grasshopper Development System – Light Edition Release 1.2 — Basics and Concepts*. Technischer Bericht IKV++ GmbH, Februar 1999. Revision 1.1.

[Gh Site] *http://www.ikv.de/products/grasshopper/*.

[Gh Tech] *Grasshopper – The Agent Platform — Technical Overview*. Technischer Bericht IKV++ GmbH, Februar 1999.

[GHR 99] Gruschke, B., S. Heilbronner und H. Reiser: *Mobile Agent System Architecture — Eine Plattform für flexibles IT–Management*. Technischer Bericht 9902, Ludwig-Maximilians-Universität München, Institut für Informatik, München, 1999.

[Gon 98] Gong, L.: *Java Security Architecture (JDK1.2), Version 1.0*. Spezifikation, Sun Microsystems, Oktober 1998.

[HAN 99a] Hegering, H.-G., S. Abeck und B. Neumair: *Integriertes Management vernetzter Systeme — Konzepte, Architekturen und deren betrieblicher Einsatz*. dpunkt-Verlag, 1999.

[HaRe 99] HAUCK, R. und H. REISER: *Monitoring of Service Level Agreements with flexible and extensible Agents.* Technischer Bericht 1999.

[Hohl 98] HOHL, F.: *Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts.* In: VIGNA, G. [Vign 98], Seiten 92–113.

[IAIK-JCE API] *IAIK-JCE 2.5 — Reference Manual.* Technischer Bericht Institut für angewandte Informationsverarbeitung und Kommunikationstechnologie der Technischen Universität Graz, 1999.

[IAIK Site] *http://jcewww.iaik.tu-graz.ac.at/.*

[iSaSiLk API] *iSaSiLk 2.5 — Reference Manual.* Technischer Bericht Institut für angewandte Informationsverarbeitung und Kommunikationstechnologie der Technischen Universität Graz, 1999.

[iSaSiLk] *IAIK-JCE 2.5 — User Manual.* Online Dokumentation, Institut für angewandte Informationsverarbeitung und Kommunikationstechnologie der Technischen Universität Graz, 1999.

[JCA 98] *Java Cryptography Architecture — API Specification & Reference.* Online Dokumentation, Sun Microsystems, Oktober 1998.

[JCE 99] *JavaTM Cryptography Extension 1.2 API Specification & Reference.* Online Dokumentation, Sun Microsystems, 1999.

[JDK1.1-SDK] *Java Development Kit - JDK 1.1.8 Documentation.* Online Dokumentation, Sun Microsystems, 1999.

[JDK1.2-SDK] *Java 2 SDK Standard Edition Documentation Version 1.2.1.* Online Dokumentation, Sun Microsystems, März 1999.

[JLS] GOSLING, J., B. JOY und G. STEELE: *The Java Language Specification.* Addison–Wesley, 1996.

[KAG 98] KARJOTH, G., ASOKAN N. und C. GÜLCÜ: *Protecting the Computation Results of Free-Roaming Agents.* Technischer Bericht IBM Research Division, 1998.

[Kemp 98] KEMPTER, B.: *Entwurf eines Java/CORBA-basierten Mobilen Agenten.* Diplomarbeit, Technische Universität München, August 1998.

[KLM 97] KARJOTH, G., D. B. LANGE und M. OSHIMA: *A Security Model For Aglets.* IEEE Internet Computing, August 1997.

[Moun 97] MOUNTZIA, M.-A.: *Flexible Agents in Integrated Network and Systems Management.* Dissertation, Technische Universität München, Dezember 1997.

[NeLe 98] NECULA, G. C. und P. LEE: *Safe, Untrusted Agents Using Proof-Carrying Code.* In: VIGNA, G. [Vign 98], Seiten 61–91.

[Newm 97] NEWMAN, A. U.A.: *Java: Referenz & Anwendung.* Que, Special Edition Auflage, 1997.

[Oaks 98] OAKES, S.: *Java Security.* The Java Series. O'Reilly, Erste Auflage, Mai 1998.

[OKO 98] OSHIMA, M., G. KARJOTH und K. ONO: *Aglets Specification 1.1 Draft.* Spezifikation, IBM Research Division, Tokyo Research Laboratory, September 1998. Draft 0.65.

[OMG 98-03-09] *MASIF Revision.* TC Document orbos/98-03-09, Object Management Group, März 1998.

[OMG 98-04-03] *Java to IDL mapping, merged version.* TC Document orbos/98-04-03, Object Management Group, April 1998.

[OMG 98-12-17] *CORBAServices Security Service v1.2 specification.* OMG Specification formal/98-12-17, Object Management Group, Dezember 1998.

[OOC Site] *http://www.ooc.de/.*

[Orb 99] *ORBacus for C++ and Java, Version 3.1.2.* Technischer Bericht Object-Oriented Concepts, 1999.

[OrbSSL 98] *ORBacus SSL for C++ and Java, Version 1.0.1.* Technischer Bericht Object-Oriented Concepts, 1998.

[OsKa 97] OSHIMA, M. und G. KARJOTH: *Aglets Specification (1.0).* Spezifikation, IBM Research Division, Tokyo Research Laboratory, Mai 1997. Draft 0.30.

[PhKa98] PHAM, A. und A. KARMOUCH: *Mobile Software Agents: An Overview.* IEEE Communications Magazine, 36(7):26–37, Juli 1998.

[Radi 98] RADISIC, I.: *Konzeption eines policy-basierten Konfigurationsmanagements für nomadische Systeme in Intranets.* Diplomarbeit, Ludwig-Maximilians-Universität München, August 1998.

[RJB 98] RUMBAUGH, J., I. JACOBSON und G. BOOCH: *Unified Modeling Language — Reference Manual.* Addison-Wesley, 1998.

[Roel 98] RÖLLE, H.: *Prototypische Implementierung eines CORBA Topology Service.* Fortgeschrittenenpraktikum, Technische Universität München, 1998.

[SaTs 98] SANDER, T. und C. F. TSCHUDIN: *Protecting Mobile Agents Against Malicious Hosts.* In: VIGNA, G. [Vign 98], Seiten 44–60.

[Schn 96] SCHNEIER, BRUCE: *Applied Cryptography.* Wiley & Sons, Zweite Auflage, 1996.

[SSLEAY Site] *http://www.ssleay.org/.*

[Tane 98a] TANENBAUM, ANDREW S.: *Computernetzwerke.* Prentice Hall, Dritte Auflage, 1998.

[Tane 98b] TANENBAUM, ANDREW S.: *Die Verarbeitungsschicht - Netzsicherheit*, Kapitel 7.1, Seiten 613–657. In: [Tane 98a], Dritte Auflage, 1998.

[Vign 98] VIGNA, G. (Herausgeber): *Mobile Agents and Security.* Lecture Notes in Computer Science. Springer, 1998.

[Vign 98a] VIGNA, G.: *Cryptographic Traces for Mobile Agents.* In: VIGNA, G. [Vign 98], Seiten 137–153.

[WaSc 96] WAGNER, D. und B. SCHNEIER: *Analysis of the SSL 3.0 protocol.* Technischer Bericht University of California und Berkeley Counterpane Systems, 1996.

[X.500] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Information Processing Systems – Open Systems Interconnection – The Directory – Overview of Concepts, Models, and Service.* Technischer Bericht International Organization for Standardization and International Electrotechnical Committee, Dezember 1988. International Standard 9594-1.