Technische Universität
Darmstadt
Fachbereich Informatik

Fraunhofer-Institut für
Graphische Datenverarbeitung

Diplomarbeit

Transaktionssicherheit für die Migration von mobilen Agenten

von

Thu Trang Pham Thi
Matrikelnummer 921189

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Graphisch-Interaktive Systeme
Fraunhoferstraße 5
64283 Darmstadt

Betreuer: Dipl.-Inform. Jan Peters

Prüfer: Prof. Dr.-Ing. J.L. Encarnação

Aufgabenstellung für die Diplomarbeit des
Frau cand.-Inform. Thu Trang Pham Thi
Matrikel-Nr. 921189

Thema: "Transaktionssicherheit für die Migration von mobilen Agenten"

Am Fraunhofer IGD wird im Projekt SeMoA (Secure Mobile Agents) an einer sicherheitsorientierten Plattform für mobile Agenten geforscht. Eine wichtige Eigenschaft der Agenten stellt dabei dessen Mobilität, also die Migrationsfähigkeit von Programm, Daten und aktuellem Zustand von einem Wirtssystem auf ein anderes, dar. Der entsprechende Forschungszweig hat sich in der wissenschaftlichen Gemeinschaft bereits etabliert und behandelt im Kontext der Agenten-Technologie grundlegende Fragen zur Kategorisierung, Optimierung und Absicherung der Agentenmigration.

Das Projekt SeMoA setzt im Hinblick auf diese Fragestellungen bereits einen Großteil der formulierten Anforderungen um. Vor allem im Bereich Sicherheit sind die entsprechenden Konzepte und Umsetzungen fortschrittlich. Im Rahmen einer juristischen Simulationsstudie in Zusammenarbeit mit der Universität Kassel wurde allerdings deutlich, dass die derzeitige technische Umsetzung der Agentenmigration keine Nachweisbarkeit der Zustellung gewährleistet. Der Nachweis der Zustellung eines Agenten durch den Absender ist aber notwendig für die Rechtsverbindlichkeit von mobilen Agenten.

Im Rahmen der Diplomarbeit soll aus diesem Grund ein Protokoll entwickelt und implementiert werden, welches den Vorgang der Agentenmigration als Transaktion realisiert und dadurch das bestehende Konzept erweitert. Es soll sowohl möglich sein, den Vorgang der Agentenmigration zwischen zwei Agentenpattform nachweisen, als auch im Fall einer Störung ein Rollback durchführen zu können. Der Entwurf eines solchen Protokolls für Softwareagenten ist nicht trivial, da der Empfänger des Agenten die Kommunikation während der Übertragung jederzeit abbrechen oder die Bestätigung des abgeschlossenen Migrationsvorgangs verweigern kann.

Das zu entwickelnde Framework soll darüber hinaus die Verhandlung von Parametern zwischen der sendenden und der empfangenden Agentenplattform im Vorfeld der eigentlichen Migration erlauben. Interessante Aspekte sind hierbei z.B. die zu erwartende Größe und die (Besitzer-)Identität der zu migrierenden Komponente, sowie die für die Ausführung ihrer Aufgabe benötigten Dienste und Rechte auf der Zielplattform. Die Parameter werden im Hinblick auf die Sicherheitspolitiken der beiden Plattformen verhandelt. Durch entsprechende Ergebnisse solcher Verhandlungen lassen sich gegebenenfalls unnötige Agentenmigrationen vermeiden.

Der modulare Aufbau der SeMoA-Architektur erleichtert die Integration des neuen Migrationsprotokolls in die Plattform. Es besteht allerdings eine Abhängigkeit der Module für die Agentenlokalisierung und der darauf basierenden ortstransparenten Kommunikation zwischen mobilen Agenten von dem eingesetzten Migrationsparadigma. Dies ist bei der Entwicklung des transaktionsbasierten Migrationsprotokolls zu beachten. Zudem soll die implizite Nutzung des entwickelten Frameworks für den Agentenentwickler möglichst transparent geschehen.

Die Diplomarbeit soll folgende Aspekte berücksichtigen:

Literaturrecherche Es soll der State-of-the-Art in den Bereichen Agentenmigration und Rechtsverbindlichkeit von mobilen Agenten zusammengetragen werden.

Generelle Probleme Es sind die generellen Schwierigkeiten aufzuzeigen und zu bewerten, die sich durch herkömmliche Migrationsverfahren für Agenten auch im Kontext Rechtssicherheit ergeben.

Bedrohungs- und Anforderungsanalyse Auf Basis einer Bedrohungsanalyse sind anschließend Anforderungen für das transaktionsbasierte Migrationsprotokoll zu erstellen.

Protokollspezifikation Entwicklung des transaktionsbasierten Migrationsprotokolls als Erweiterung des momentan realisierten Konzepts.

Evaluierung Es ist zu prüfen, ob und in welchem Maß das neue Migrationsprotokoll Einfluss auf die Performance einer einzelnen Migration bzw. die Leistungsfähigkeit des Gesamtsystems hat.

Related Work Einordnung und Diskussion des entwickelten Migrationsprotokolls im Vergleich zu existierenden Konzepten.

Darmstadt, den 29. Oktober 2004

Betreuer:
Dipl.-Inform. Jan Peters

Prof. Dr.-Ing. J. Encarnação

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Juni 2005 Thu Trang Pham Thi

Danksagung

Danken möchte ich bei dieser Gelegenheit meinen Eltern und meinem Onkel, dass sie mir das Informatik-Studium an der TU-Darmstadt ermöglicht haben. Dank schulde ich auch den Professoren der TU-Darmstadt, die mir ein fundiertes Wissen mit auf den Weg gegeben haben, was mir bei dieser Diplomarbeit sehr geholfen hat. Weiterer Dank gebührt den Mitarbeitern des IGD Darmstadt, besonders meinem Betreuer Herrn Dipl.-Informatiker Jan Peters, der mir immer mit Rat und Tat zur Seite stand. Schließlich gilt der besondere Dank meinem Kommilitonen Herrn cand. Päd. Jürgen Lugsch, der die mühevolle Aufgabe des Korrekturlesens auf sich nahm.

Darmstadt, Juni 2005 — Thu Trang Pham Thi

Inhaltsverzeichnis

1 Grundlagen **1**

1.1 Einleitung 1

1.1.1 Motivation 1

1.1.2 Zielsetzung 1

1.1.3 Überblick 2

1.2 Agententechnologie 2

1.2.1 Einleitung 2

1.2.2 Mobile Agenten 3

1.2.3 Motive für den Einsatz mobiler Agenten 4

1.2.4 Hauptanwendungsfelder von mobilen Agenten 4

1.2.5 Migration 5

1.3 Rechtssicherheit 6

1.3.1 Aufgabendelegation 6

1.3.2 Unabstreitbarkeit 7

1.3.3 Gestaltung 7

1.4 Vergleichbare Arbeiten 7

1.4.1 Basis-Verfahrensweise der Infrastruktur für mobile Agenten 7

1.4.2 Annäherung an Sicherheitsaspekte für mobile Agenten 8

1.4.3 Für mobile Agenten Transaktionsmodell 10

1.5 **SeMoA** 11

1.5.1 Systemarchitektur 12

1.5.2 Migrationskomponenten 13

1.5.3 Sicherheitsarchitektur 13

2 Protokollentwicklung: Transaktionsbasierte Migration **19**

2.1 Anforderungsanalyse 19

2.1.1 Sicherheitsaspekte 19

2.1.2 Rechtliche Aspekte 23

2.2 Protokollspezifikation 25

2.2.1 Aushandlung von Migrationsparametern 25

2.2.2 Migration als Transaktion 26

2.2.3 Protokollentwurf 27

2.3 Implementierung 31

2.3.1 Klassen 31

2.3.2 Integration in **SeMoA** 36
2.4 Evaluation 37
2.4.1 Testumgebung / Testszenario 37
2.4.2 Ergebnisse und Bewertung 39

3 Resümee **41**
3.1 Zusammenfassung und Ausblick 41
3.1.1 Zusammenfassung 41
3.1.2 Evaluation 41
3.1.3 Ausblick 42

A Programmcode **43**
A.1 Programmcode von `ChangeTicket` 43
A.2 Programmcode von `SendBack` 44
A.3 Programmcode von `Timer` 51
A.4 eingefügte Programmcode von `RawOutGate` 56
A.5 eingefügte Programmcode von `RawInGate` 59
A.6 eingefügte Programmcode von `Signals` 61
A.7 eingefügte Programmcode von `OutGate` 62
A.8 eingefügte Programmcode von `InGate` 63
A.9 Programmcode von `LSClient` 70
A.10 Programmcode von `AgentGateway` 72

B Akronyme **75**

Tabellenverzeichnis

2.1 Test für die alte Migration 38
2.2 Test für die neue erfolgreiche Migration 38
2.3 Test für die neue erfolglose Migration 38
2.4 Test für die neue Migration bei gleichzeitigem Erfolg und Timeout 39
2.5 Test für die neue Migration bei Erfolglosigkeit und Timeout 39

Abbildungsverzeichnis

1.1 Policy-based Infrastruktur 8
1.2 Transport 14
1.3 Dynamic Proxys 16
1.4 **SeMoA**-Security 17

2.1 Diagramm Senden 28
2.2 Diagramm Empfangen 29
2.3 Diagramm Protokoll 30
2.4 changeTicket 33
2.5 SendBack 34
2.6 timeout 35

Kapitel 1

Grundlagen

1.1 Einleitung

1.1.1 Motivation

Heutzutage ist die Informationstechnologie eine der wichtigsten Techniken für die Menschheit, welche die Forscher schon seit den 50er Jahren des letzten Jahrhunderts primär interessiert. Die Informationstechnologie wird auch in vielen anderen Wissenschaften wie z.B. Maschinenbau und Biologie verwendet, um das Leben der Menschen nach Möglichkeit besser zu gestalten. Weil die Anforderungen der Menschen in der Welt nach und nach steigen, werden auch die neuen Informationstechnologien ständig weiterentwickelt, um diesem Bedarf der Menschen gerecht zu werden. Die Agententechnologie ist eine von diesen neuen Informationstechnologien, in die im Moment investiert wird [28].

Das Fraunhofer Institut für Graphische Datenverarbeitung in Darmstadt ist eine der Institutionen, die an der relativ neuen Agententechnologie forscht. Das Projekt besteht aus mehreren Teilen, die notwendigerweise für die Entwicklung der Agententechnologie gebraucht werden: Eine von diesen wichtigen Teilen ist die Transaktionssicherheit für die Migration von mobilen Agenten, die in dieser Diplomarbeit dargestellt wird.

1.1.2 Zielsetzung

Es erhebt sich die Frage, warum die Transaktionssicherheit für die Migration von mobilen Agenten erforscht und entwickelt wird: Um das zu erläutern, muss weiter ausgeholt werden.

Die Informationstechnologie, die heutzutage nach und nach entwickelt wird, macht die Menschen immer stärker abhängig von den Maschinen, weil die Maschinen die Menschen in der Arbeit unverzichtbar unterstützen. Zum Beispiel können Menschen zu Hause mit einem Computer im Internet sein und dadurch Informationen aus der ganzen Welt erfahren. Nach Eingabe von ein paar Schlüsselwörtern in eine Suchmaschine wird meist eine Reihe von Internetadressen aufgelistet, die auf den gesuchten Inhalt verweisen. Wenn diese Suchmaschinen auf andere Server zugreifen, um die benötigte Informationen zu liefern, werden nur “normale” Texte transportieren. Folgend dem erhöhten Bedarf der Menschen heutzutage ist das noch nicht genug, weil die Benutzer nicht nur einfache Texte, sondern auch Bilder, Ordner oder Dokumente suchen wollen. Aus diesem Grund ist die Migration von mobilen Agenten für die IT-Branche von

großem Interesse. Wenn mobile Agenten migrieren, enthalten sie nicht nur einfache Texte, sondern können auch Code und Ressourcen zum Zielort bringen. Diese Idee wurde schon vor dieser Arbeit verwirklicht und funktionierte im Prinzip auch. Aber es fehlten noch Sicherheiten für die Benutzer bei der Migration von mobilen Agenten. Deswegen wird die Transaktionssicherheit für die Migration von mobilen Agenten in dieser Arbeit besonders herausgestellt und erweitert, um mehr Benutzerfreundlichkeit garantieren zu können.

1.1.3 Überblick

Diese Arbeit wird in folgende drei Teile aufgeteilt:

- **1. Teil: Grundlagen**: In diesem Abschnitt wird zuerst allgemein die Entwicklung der Agententechnologie vorgestellt. Er besteht aus einer Einführung in die Mobile-Agenten-Technologie und einer Erläuterung von notwendigen Begriffen für die Migration. Zweitens werden Aspekte der Rechtssicherheit vorgestellt. In diesem Abschnitt wird auch die Aufgabendelegation und Unabstreitbarkeit der Rechtssicherheit für die Benutzer vorgestellt, wenn die Agenten migrieren. Im nächsten Schritt folgt das Vergleichen dieser Arbeit mit anderen Projekten, bei denen die Migration von mobilen Agenten erforscht wird. Der letzten Abschnitt stellt die allgemeine Architektur der Agentenplattform-**SeMoA** (Secure Mobile Agent) dar, die durch das Fraunhofer-Institut für Graphische Datenverarbeitung in Darmstadt entwickelt wird.

- **2. Teil: Protokollentwicklung (Transaktionsbasierte Migration)**: In diesem Abschnitt wird ein Konzept für die Implementierung der Transaktionssicherheit für die Migration von mobilen Agenten vorgestellt. Um sie zu implementieren, werden zuerst die Anforderungen für Sicherheitsaspekte und rechtliche Aspekte für das Programm analysiert. Dann wird das Protokoll für die Migration spezifiziert (Migrationsparameter und Migration als Transaktion), und danach werden diese Protokolle entworfen. Nächster Schritt ist die Implementierung, die aus neuen Klassen und der Integration in **SeMoA** besteht. Am Ende folgt ein Benchmark-Test der Erweitungen und eine abschließende Bewertung der Ergebnisse.

- **3. Teil: Resümee**: Der letzte Abschnitt fasst zuerst noch einmal diese Arbeit zusammen, um diese anschließend Arbeit zu evaluieren. Am Ende dieser Arbeit steht ein Ausblick, mit dessen Hilfe und Tipps nachfolgende Entwickler das Programm möglicherweise weiter verbessern können.

1.2 Agententechnologie

1.2.1 Einleitung

Heutzutage sind mobile Agenten ein Forschungsgebiet der Informatik. Mobile Agenten sind autonome Programme, die meistens durch die Sprache JAVA implementiert werden. Diese Programme können von einem Rechner zum anderen bzw. von einem System zum anderen mit unterschiedlichen Prozessoren und Betriebsystemen migrieren. Das Agentensystem verpackt, verschlüsselt und signiert die Daten, den gesamten Code und den Ausführungszustand in einem mobilem Agent, um danach alles an den Bestimmungsort zu verschicken. Außerdem kann der mobile Agent mit anderen Agenten auf anderen Rechnern kommunizieren, bei dem er Gast ist.

Er kann Dienste bei Rechnern anfordern oder selbst Dienste anbieten.

Mobile Agenten bieten viele Aspekte für Forschungsgebiete in der Informatik an. In der künstlichen Intelligenz stellen mobile Agenten die Aspekte der Planung und Kooperation (Multi-Agenten Systemen) dar. In der Softwaretechnik wird ein "Mobiler Agent" als ein aktives Objekt bezeichnet, das neue Sichten auf die Architektur eines Programms erzeugt. In verteilten Systemen sind mobile Agenten ein neues Paradigma zur Kommunikation im Rechnernetzwerk.

1.2.2 Mobile Agenten

In der Informatik ist der Begriff der mobilen Agenten heute noch nicht eindeutig definiert. In der künstlichen Intelligenz werden Agenten als Programme verstanden, die Kenntnisse sowie menschliche Eigenschaften haben. Mobilität ist ein Aspekt, der den Agenten ermöglicht, im Netzwerk von Rechner zu Rechner migrieren. Für Nwane [26] ist die Tatsache, dass sowohl die Ausdrücke "mobil" und "Agenten" nicht innerhalb der Informatik oder anderen technischer Wissenschaften definiert wurden, sondern umgangsprachlich schon mehrfach besetzt sind, der Hauptgrund für die Verwirrung, die um die Begriffe existiert [4].

Das Wort "Agent" kommt aus dem Lateinischen. Es hat mehrere, verschiedene Bedeutungen in vielen Bereichen. Auf Deutsch bedeutet es "Spion", einfach "Person" oder "Künstlermanager". Auf Englisch wird es verwendet für "house agent" (Häusermakler) oder "travel agency" (Reisebüroinhaber und Reisebürokaufmann). In der Informatik entstanden "Agenten" in dem Forschungsgebiet der künstlichen Intelligenz seit Ende der 70er Jahre.

Wenn Agenten als Software-Komponenten eingesetzt werden, sind mindestens die folgenden Eigenschaften zu erfüllen:

- **Autonomie**: Der Agent kann auch selbstständig Pläne erstellen, d.h. er ist nicht nur abhängig von vorgegebenen Befehlen eines menschlichen Planers, sondern kann auch alleine die Sachlage analysieren und daraufhin agieren.
- **Soziales Verhalten**: Die ACL (Agent Communication Language) ermöglicht z.B die Kommunikation zwischen Agenten oder Menschen über das Netzwerk.
- **Reaktivität**: Agenten nehmen ihre Umwelt wahr und reagieren darauf.
- **Proaktivität**: Agenten organisieren zielgerichtetes Planen und Handeln. Sie ergreifen die Initiative.

In der künstlichen Intelligenz werden mobile Agenten über die menschlichen Eigenschaften wie Wissen, Glauben, Absicht und Verpflichtung zusammengefasst. Bei mobilen Agenten ist die Migration der wichtigste Aspekt der Forschung, wie Gilbert et al [12] schreibt. Es gibt drei Arten von mobile Agenten: Statische Agenten verbleiben an einem Ort, mobile Skripte werden auf einem entfernten Rechner übertragen und dort gestartet, und mobile Objekte können während ihrer Ausführung unterbrochen und auf einen anderen Rechner transportiert werden.

Mobile Agenten sind nicht nur in der Forschung und der Entwicklung populär, sondern auch in Projekten der Industrie. Um mobile Agenten zu unterscheiden, wird die Klassifikation der Migration und das Problem der Konjunktion der optimalen Migration im nächsten Abschnitt aufgezeigt.

1.2.3 Motive für den Einsatz mobiler Agenten

Ausführlichere Betrachtungen zu diesem Punkt finden sich in [5], [19], [20], [21], [22] und [39].

1.2.3.1 Übertragungsgeschwindigkeit

Das Backbone des Internets wird in naher Zukunft gestützt auf Glasfaserkabel und rein optische Signalverarbeitung Übertragungsraten der Größenordnung von 10^6 bit/s erreichen. Dagegen wird besonders der private Endnutzer in der Regel nur über Leitungen mit Übertragungsraten in der Größenordnung von 10^6 bit/s mit dem Internet verbunden sein und bei WLAN ist die Übertragungsrate noch niedriger. Die Entwicklung der letzten Jahre zeigt, dass dieser Unterschied in den nächsten Jahren noch größer sein wird. Insbesondere im Bereich e-commerce wird ein nicht geringer Teil dieser Bandbreite noch vom sicherheitsbedingten "Protokoll-Overhead" verbraucht. Deshalb wird sich notwendigerweise im Bereich e-commerce der Trend zu Technologien, die diese Entwicklung berücksichtigen, im besonderen Maße verstärken: Mobile Agenten sind eine solche neue Technologie.

1.2.3.2 Mobile Nutzer (PDA's)

Der am stärksten wachsende Bereich im Hardwaregeschäft ist der der transportablen Geräte. Es ist den mobilen Geräten inhärent, dass sie nicht ständig mit einem Netzwerk verbunden sein können bzw. dass diese Verbindungen, insbesondere im drahtlosen Bereich, sehr fehleranfällig sind. Eine effiziente Nutzung verteilter Ressourcen über das Internet ist in diesem Bereich nur mit Technologien machbar, die keine ständige Anbindung des Nutzers benötigen bzw. im Großen und Ganzen immun gegen Störungen dieser Verbindung sind, wie es bei mobilen Agenten der Fall ist.

1.2.3.3 Kundenrelevanz

Betrachtet man klassische Suchmaschinen, so sieht man Internetapplikationen, die in der Regel für alle Nutzer die gleichen Optionen anbieten. Häufig wird versucht, über gewisse Verfeinerungseinstellungen dem User die Möglichkeit einzuräumen, das Verhalten der Suchmaschine an seine eigenen Bedürfnisse anzupassen. Leider zeigt die Erfahrung, dass häufig gerade die Einstellung, die man zur Zeit am dringendsten benötigte, um einige tausend Angebote auf die vielleicht dreißig wesentlichen zu reduzieren, nicht berücksichtigt wurde. Das bedeutet, dass die Möglichkeiten der aktuelleren Technologien, auf die individuellen Bedürfnisse von Kunden einzugehen, recht eingeschränkt sind.

1.2.4 Hauptanwendungsfelder von mobilen Agenten

1.2.4.1 e-commerce

Beschreibungen von Applikationen in diesem Bereich finden sich u.a. in [3] und [5]. Es gibt Ansätze für die Architektur von e-commerce Systemen: Die Vorteile gegenüber existierenden Architekturen wie z.B. elektronischen Warenhäusern werden vor allem darin gesehen, dass man mit mobilen Agenten elektronische Einkäufer kreieren kann, die autonom z.B. besonders günstige Angebote suchen könnten. Darüber hinaus besteht die Möglichkeit, dass diese Agenten die komplette Abwicklung des Auftrags überwachen und bei Schwierigkeiten auf Anbieterseite im Sinne des Kunden eingreifen können: z.B. kann ein mobiler Agent, der ein Flugticket gebucht hat, im Agentensystem der Fluggesellschaft verbleiben und bei Verspätungen im Flugverkehr

den Kunden informieren bzw. Alternativen ermitteln [40]. Ein weiterer Vorteil in diesem Bereich ist die Robustheit von mobilen Agenten gegen Verbindungsstörungen.

1.2.4.2 Informationsbeschaffung und -management

Beschreibungen möglicher Anwendungen in diesem Bereich finden sich u.a. in [6], [5], [7], [27]. Dieses Gebiet hat einen gewissen Schwerpunktcharakter, da die Informationsflut, die das Internet bereitstellt, mit den existierenden Technologien für den "Kunden" nur noch mit enormem Aufwand an Zeit und uneffektiver Routinearbeit einigermaßen nutzbar ist. Deshalb werden neue, elegantere Ansätze auf Basis mobiler Agenten auf eine breite Akzeptanz stoßen.

1.2.4.3 Netzwerkmanagement und -analyse

Hier findet sich die größte Anzahl an Publikationen, die neben möglichen Anwendungsfällen schon prototypische Realisierungen beschreiben [1]. Der anscheinende Entwicklungsvorsprung dieses Bereichs liegt darin begründet, dass die hier besprochenen Schwierigkeiten am wenigsten ins Gewicht fallen, da die benötigte Infrastruktur (lokale Netzwerke) existiert und unter der vollständigen Kontrolle der jeweiligen Forschungsinstitution steht. In diesem Bereich sind die ersten kommerziellen Produkte zu erwarten.

1.2.4.4 Telekommunikation

Insbesondere der Bereich der drahtlosen Kommunikation hat ein Interesse an Technologien, die robust gegen Verbindungsstörungen sind und die Übertragungskapazität schonen. In [15] wird eine mögliche Anwendung, sogenannte "smart messages", die auf mobilen Agenten basiert, beschrieben. Weitergehende Informationen wurden aus Zeitmangel nicht intensiv gesucht. Hier ist eine zusätzliche Recherche im Rahmen weitergehender Arbeiten dringend erforderlich.

1.2.5 Migration

Zuerst stellt sich die Frage, was genau mit Migration gemeint ist: Heutzutage werden mehrere Plattformen benutzt, um Texte von einem System zu einem anderen System zu verschicken. Mobile Agenten wurden entwickelt, um nicht nur Texte, sondern auch autonome Programme an dem Bestimmungsort auszuführen. Das Agentensystem "verpackt" die Daten in einem gewissem Umfang (d.h. einem mobile Agent) und transportiert diesen Agent dann von einem System zum anderen System oder von Computer zu Computer. Agenten-Plattformen stellen für den Agenten die benötigte Infrastruktur bereit. Der Plattformwechsel wird dabei als Migration bezeichnet.

Der Migrationsprozess wird durch drei Aspekte definiert: "Programmer' s point of view", "Agent's point of view", "Network' s point of view". Hier wird nur auf die "Agent's point of view" eingegangen, weil eine optimierte Strategie für die Migration möglichst autonom durch Agenten mit der Unterstützung durch Intelligenz ausgewählt wird.

Eine Methode verschickt den Code von mobilen Agenten als Migration über das Netzwerk, was als Strategie der Migration bezeichnet wird. Wenn mobile Agenten migrieren, bringen sie sowohl den Code, als auch den Zustand der Ausführung und der Daten mit ein. Es gibt viele verschiedene Strategien der Migration. Die Strategien der Migration werden mit den folgenden Fragen klassifiziert: Wieviel Code wird transportiert? Wann wird die Code transportiert? Wo

wird der Code hin transportiert ?

Eine optimale Migration hat eine minimale Belastung des Netzwerks und eine minimale Zeitspanne während der Migration zur Folge - sowohl für den nächsten Standort des Reiseplans der Agenten, als auch für den gesamten Reiseplan. Weil der Reiseplan im Voraus für die Optimierung bekannt sein muss, müssen die Bedingungen von Orten im Reiseplan ebenfalls im Voraus bekannt sein. Dieses Faktum gilt als sehr schwierig zu handhaben. Deshalb ist das Hauptziel, dass ein optimiertes Lösungskonzept für "benutzer-definierte" Aufgaben gewählt wird. Dafür wird es in zwei Klassen geteilt: Das Makro-Level und das Mikro-Level.

- **Makro-Level**: Dieser baut auf dem Mikro-Level auf. Er konzentriert sich auf Fragen, die mit dem Lösungskonzept der benutzer-definierten Aufgaben verbunden werden.

- **Mikro-Level**: Er wird mit den Fragen, die eine ideale Migration definieren (besonders die der Strategien der Migration und des Transports), verbunden.

Die verschiedenen Ziele des Makro- und des Mikro-Levels ergeben sich durch die Anforderungen der verschiedenen Datenkonzepte. Darum wird die Benutzung einer Intelligenzhilfe in Betracht gezogen, um die Daten zu analysieren. Desweiteren ergibt sich folgende Frage: Wo kann diese Intelligenz innerhalb der Level benutzt werden und welche Intelligenz kann dafür benutzt werden? [11].

Das obige Lösungskonzept ist nur ein mögliches Lösungskonzept. Außerdem gibt es noch ein zentrales Lösungskonzept und ein verteiltes Lösungskonzept, dessen Informationen auf die Server, die Dienste und das Netzwerk verteilt werden."Mobile Agent Server" bieten die Informationen über die verfügbaren Dienste und Qualitäten des Netzwerk im verteilten Lösungskonzept an. Deshalb besteht es auch aus verschiedenen Strukturen, welche die benötigten Informationen bereitstellen: eine Globale-Netzwerk-Struktur, eine Nachbar-Struktur und ein Struktur des Randfeldes.

Außerdem kann die Strategie für die Migration auf einer Kalkulation der Klassifikation einer Intelligenz für die Migration basieren. Das ausführbare Modell wird in einem Baum aufgebaut, so dass die Belastung des Netzwerks und die Zeit für das Transportieren berechnet werden kann.

1.3 Rechtssicherheit

Wie in Abschnitt 1.2 beschrieben wurde, sind mobile Agenten heutzutage eine neue Technik in der Informationstechnologie, die ein großes Forschungsgebiet der Informatik darstellt. Die Migration ist als eine der wichtigsten Eingenscaften von mobilen Agenten besonders zu berücksichtigen. Bei der Migration von Agenten gilt es, Methoden für rechtliche Aspekte zu gestalten, um den Bedarf des Benutzers an Sicherheit erfüllen zu können. Diese Zielvorgabe der Rechtssicherheit wird im folgenden Abschnitt dargestellt.

1.3.1 Aufgabendelegation

Ein Fallbeispiel entsteht, wenn ein Kunde einen Gegenstand per Internet kaufen will. Um den Kauf fertigzustellen, füllt der Kunde zuerst ein Formular aus, das nach dem Namen des Kunden, dem Vornamen, der Adresse, Email etc. verlangt. Danach wird das Formular per Internet

zum Zielsystem verschickt. Ob diese Aktion erfolgreich war, bestätigt das Zielsystem oder der Verkäufer in der Regel nicht. Der Kunde weiß nicht, wie der Zustand des Formulars ist und es ist möglich, dass er sein Geld verliert und den gekauften Gegenstand gar nicht bekommt. Deswegen ist hier ein Maß von Rechtssicherheit nötig, bei dem sowohl die Spuren der mobilen Agenten überprüft, als auch Informationen von dem Zielsystem eingeholt werden. Es entstehen also rechtliche Züge in der Informationstechnologie, wenn mobile Agenten migrieren. Es ist nicht nur bequem, sondern garantiert auch Sicherheit für die Benutzer, wenn sie die neue Technik bei ihren Geschäften anstatt traditioneller Methoden verwenden.

1.3.2 Unabstreitbarkeit

Die obige Aufgabendelegation zeigt, dass das System der Technikgestaltung die Zielvorgabe der Rechtssicherheit sowohl für die Empfänger als auch für die Sender anbietet und kein Konflikt zwischen den Sendern und Empfängern entstehen kann. Die Empfänger können durch Bestätigung den Empfang nachweisen. Die Sender können ihre Aktion des Kaufens kontrollieren, sobald ihr Formular das Zielsystem erreicht hat.

1.3.3 Gestaltung

Um die Zielvorgabe der Rechtssicherheit zu realisieren, braucht es zuerst die Existenz einer funktionierenden "Infrastruktur des Vertrauens" [31]. Es kommt darauf an, dass der mobile Agent über die Rechner im Netzwerk migrieren und benötigte Informationen zu dem Rechner schicken kann, den er gerade verlassen hat.

1.4 Vergleichbare Arbeiten

Im folgenden Abschnitt werden Infrastrukturen für Basisverfahren von mobilen Agenten im Vergleich zu **SeMoA** vorgestellt.

1.4.1 Basis-Verfahrensweise der Infrastruktur für mobile Agenten

Zuerst wird die Basis-Verfahrensweise für ein Lösungskonzept von mobilen Agenten dargestellt. Diese Verfahrensweise unterstützt sowohl die dynamische Konfiguration als auch die Ausführung der mobilen Agenten mit einem flexiblem Management. Außerdem wird eine Basisverfahrensweise für Middleware beschrieben. Diese erlaubt, dass die Migration der Agenten während der Ausführung an entstehende Anforderungen der Applikation und die Bedingungen der Umgebung angepasst wird; dabei wird allerdings nicht auf die Implementierung des Codes vom Agenten eingegangen [25].

Mobile Agenten führen autonome Objekte aus, die mit einem Code-Teil und einen Status-Teil durch ein Netzwerk wandern. In der traditionellen Annäherung einer Programmierung von mobilen Agenten gibt es keine Trennung zwischen der Berechnung von Funktionen und dem mobilen Teil. Diese Eigenschaft der Struktur verursacht Schwierigkeiten bei Rekonfigurationen oder der Erstellung von neuen Migrationsmustern der Agenten. Deswegen ist die Trennung zwischen zu berechnenden Funktionen und dem mobilen Teil eine von den Basis-Verfahrensweisen, die erforscht werden, um die Komplexität zu reduzieren. Diese Basis-Verfahrensweise für mobile Agenten wird "Comprehensive Programming Model" genannt. Um diese Basis-Verfahrensweisen zu entwerfen, werden die Strategien des mobilen Teils bestimmt. Dies wird durch die "Ponder

Policy” Sprache erreicht.

Eine Basis-Verfahrensweise für die Infrastruktur ist die Verwaltung der integrierten Middleware für die Methoden des Lebenszyklus-Managements der mobilen Agenten. Sie wird in der Architekturebene mit großen Anzahlen von Dienste auf jeder verschiedenen Ebene strukturiert (siehe Abbildung 1.1). Die Infrastruktur der Services wird durch das Soma Mobile Agent System aufgebaut. In **Soma** stehen die Services Naming Agent, Communication, Security und Interoperability zur Verfügung. Nur die im Folgenden genannten Services für die Verfahrensweisen werden berücksichtigt: Policy Specification und Initialisation, Policy Enforcement, Monitoring und Event Support, Policy-controlled Mobile Agent sowie Policy Mobility Types (Taxonomy).

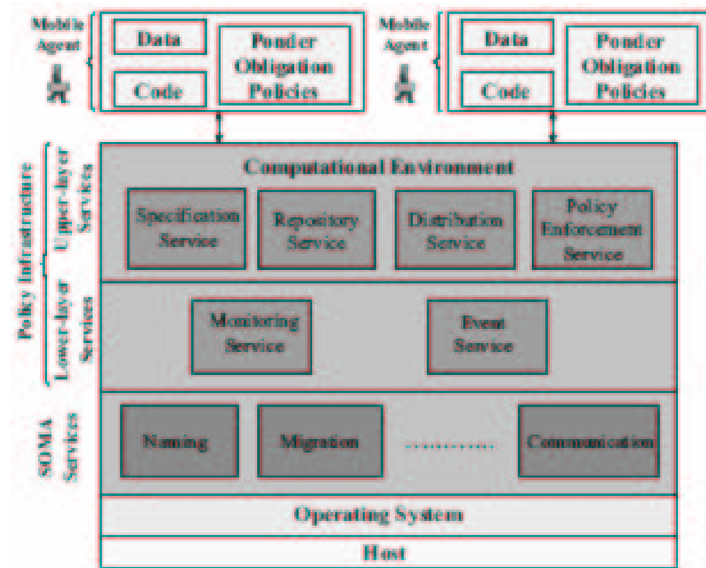

Abbildung 1.1: Policy-based Infrastruktur

Idealerweise gilt hier, dass der Administrator des Netzwerk-Systems die Trennung zwischen zu berechnenden Funktionen und dem mobilen Teil anbietet, um die “Policy” gesteuerten mobilen Agenten zwischen den Knoten zu bewegen und dabei lokales Lastenausgleichsmanagement benutzen zu können. Zum Vergleichen mit dieser Basis-Verfahrensweise ist **SeMoA** noch nicht optimal, wenn man die zu berechnenden Funktionen in dem Mobile-Teil weglässt. Wenn ein mobiler Agent in **SeMoA** migriert, kann er entweder die zu berechnenden Funktionen mitbringen oder es bleiben lassen, wenn sie im Zielsystem existieren. Eine optimalere Verfahrensweise ermöglicht den Agenten den Aufruf einer nicht migrierten Funtion durch dynamisches Nachladen der entsprechenden Implementierung.

1.4.2 Annäherung an Sicherheitsaspekte für mobile Agenten

Mobile Agenten bieten eine Technologie für elektronisches Handeln an. Mit der Erweiterung des Global Internet Service ändert elektronischer Handel die traditionellen Wege des Geschäftbetriebs. Ein bemerkbarer Vorteil vom elektronischem Handel ist die Unabhängigkeit vom geographischen Standort und die Geschwindigkeit beim elektronischen Handel ist höher als im traditionellen Transaktionsprozess. Deswegen muss die Aufgabe der Sicherheitsmaßnahmen

ausgiebig behandelt worden sein, bevor sie im elektronischem Handel eingesetzt werden.Im Folgenden wird eine Annäherung an Sicherheitsaspekte für die Kontrolle der Migration vom mobilen Agenten beschrieben, die auf einem P/V (Pass und Visum) basiert. P/V basiert auf dem **SAFER** (Secure Agent Fabrication Evolution and Roaming for e-commerce) [13] Framework, weil **SAFER** eine Menge von Sicherheitsmechanismen bereit stellt, die für "Agenten-Gesellschaften" eingesetzt werden können.

Der sogenannte MSC (Message Security Control) reguliert und überwacht als Gateway die Immigration und Emigration von Agenten in bzw, aus einer lokalen Gesellschaft.
Sobald der mobile Agent migriert, wird P/V als Authentifizierung des digitalen Berechtigungsnachweises benutzt: D.h. MSC ist ein vertrauenswürdiges Objekt. Es besteht aus zwei Teilen: Immigrationsservice und Emigrationsservice. Im Prinzip gibt MSC den Berechtigungsnachweis an und stempelt den Berechtigungsservice zur Eingangs- und Ausgangskontrolle ab:

- **Ausgangskontrolle**: Wenn der mobile Agent von der lokalen Gesellschaft emigrieren will, wird er einen gestempelten Pass vom MSC bekommen.

- **Eingangskontrolle**: Der MSC stempelt das Visum für mobile Agenten ab, wenn sie in eine ferne Gesellschaft immigrieren wollen.

Dies ist der erste Schritt für die Kontrolle von mobilen Agenten und dem Schutz von ihrem Host, der Gesellschaft. Der MSC führt schwarze Listen: Eine schwarze Liste der Gesellschaft, eine schwarze Liste der Betriebe und eine schwarze Liste der Diener oder Besitzer. Mit der schwarzen Liste kann der MSC die Kontrolle über Vergabe oder Nicht-Vergabe der Pässe oder Visa für mobile Agenten beschränken.

Als Sprache wird Java für die Implementierung von P/V und dem System der **SAFER** Migration benutzt: Das Security Package von Java und APIs für die Implementierung zum Verschlüsseln und für digitale Signaturen.

P/V ist ein praktikables Schema für die Migrationskontrolle von mobilen Agenten, denn es ermöglicht das Testen und die Bewertung der Implementierung. Das Verfahren bietet Aspekte zur Kontrolle an, wenn der mobile Agent migriert: Das ist auch die Idee von **SeMoA**. In **SeMoA** werden "Gateways" ("Ingate" und "Outgate" anstatt von P/V in diesem Verfahren) berücksichtigt, um die Agenten beim Immigrieren und Emmigrieren zu prüfen. Dies ist ein wichtiger Sicherheitsaspekt der Migration.

Mobile Agenten sind autonome und proaktive Software Einheiten, die über heterogene Netzwerke migrieren können, um miteinander zu kommunizieren und zu kooperieren. Deswegen brauchen sie einen "Reiseplan" (Itinerary), der das Besuchen der Hosts im Netzwerk für die Lösung ihrer Aufgaben ermöglicht.

Folgender Sachverhalt sei gegeben: Der Besitzer gibt eine Aufgabe für den Agenten an, aber der Agent erhält keine konkreten Informationen zur Lösung dieser Aufgabe. Deshalb soll eine Reise organisiert werden, um das Netzwerk nach nötigen Informationen für den Besitzer zu durchsuchen. Dafür braucht es eine optimale Reise, die durch den Reiseplan aufgebaut wird. Um diese zu realisieren, werden die verfügbaren Informationstrukturten der Informationen benutzt.

Das Ziel: Eine Infrastruktur anzubieten, die für autonome Agenten bereitsteht. Die Wichtigkeit ist hier nicht unbedingt, was der Agent machen muss, sondern wie er es machen muss. Ein generelles Framework und eine Infrastruktur werden entworfen, um dieses Ziel zu erreichen. Der Überblick über das allgemeine Framework besteht aus drei Teilen: Abbildung der Module, Verwaltung der Planung und Planung der Migration.

- **Abbildung der Module**: Es werden Daten gesammelt und auf deren Basis lokale Informationen erzeugt. Jede Agentur verwaltet ihre lokalen Abbilder. Wegen der Quantität sollen die lokalen Abbilder nicht alle Informationen des Netzwerks speichern. Das "Tracy Domain" [4] Konzept wird benutzt, um dem lokalen Bereich vom Netzwerk zu entsprechen. Außerdem hat ein Teil der Module die Kontrolle der Sensoren im Netz inne, welche die Daten für den Teil sammeln. Im Intervall dieser Sensoren ergibt sich das Ergebnis als ein Maß im lokalen Bereich. Dies und die Eigenschaften der Agentur erzeugen das Update oder die Komplementierung der Daten des Teils.

- **Planung des Routen-Moduls**: Dieses berechnet den kürzesten Weg der Migration im Netzwerk. Es benutzt vor allem Daten über die Topologie, die Verbindung und die Qualität. Der Planungsprozess des Routen Moduls basiert auf dem Traveling Salesman Problem, das NP-vollständig ist.

- **Planung der Migration-Module**: Es wird ein Reiseplan aufgebaut, um eine optimale Durchreise zu ermöglichen. Das Ziel ist hier, dass nur der nötigste Teil des Codes transportiert wird, der dann in der fernen Agentur verbleibt. Deswegen werden Agenten in deren Quantität reduziert, sowie über das Netzwerk geladen und dadurch sollte die Zeitspanne für das Transportieren kleiner sein. Um eine Entscheidung zu treffen, welcher Teil des Codes geliefert wird, wird eine Bewertung seiner Kosten gemacht. Dieser Vorgang wird Migrationsstrategie genannt. Die Strategie der Migration wird durch zwei gestellte Fragen bestimmt - was wird transportiert und wie wird es transportiert.

Zusammenfassung: Mit der Unterstützung der obigen Module passt sich der Agent an die wechselnden Netzwerke und Umgebungen der Services an. Die Basisinformationen werden auf die lokal verfügbare Struktur verteilt. In **SeMoA** ist der Reiseplan wie im obigen Abschnitt beschrieben, noch nicht entwickelt worden. Diese Idee bietet wichtige Aspekte für mobile Agenten und die Migration. Es kommt darauf an, dass **SeMoA** in der Zukunft dahingehend weiter entwickelt wird.

1.4.3 Für mobile Agenten Transaktionsmodell

In der heutigen Zeit bieten Hardware-Technologien drahtlose Funkverbindungen an, um ein weltweites Informationsnetzwerk für die Gesellschaft unter Berüchsichtigung mobiler Nutzer zu ermöglichen. Eine von diesen neuen Hardware-Technologien ist der mobile Computer - dessen Zugriff auf Daten in einem festen Netzwerk besteht aus "Transaktionen". In diesem Teil wird eine mobile Transaktion vorgestellt, welche den Erwartungen der Benutzer von mobilen Transaktion gerecht wird.

Die mobile Transaktion wird als die fundamentale Einheit der Berechnung in einer mobilen Umgebung definiert. Die ausführbare Hauptfunktion der mobilen Agenten ist der DAA (Data Access Agent) [9]. DAA, das Quelle System und mobile Transaktionen erzeugen das drei Schichten-Referenz-Modell für das System der Datenbank. Die gesamte Komponente wird MTM (Mobile Transaction Manager) genannt. Es gibt fünf Anforderungen an die mobile Transaktion:

- **Der Aufbau auf existierenden Systemen von multiplenDatenbanken ohne Duplizierung (systemseitig unterstützt),**
- **Das Erfassen der Bewegungen von mobilen Transaktionen sowie der Datenzugriff,**
- **Die Transaktionskontrolle,**
- **welche die Bewegung der mobilen Einheiten verfolgt,**
- **Das Anbieten der Atomarität und Dauerhaftigkeit von Transaktionen.**

In diesem Abschnitt wird ein spezielles Modell für mobile Transaktionen beschrieben: die sogenannten “Kangaroo Transaktionen”. Dieses Modell wird auf dem traditionellen Transaktionstyp aufgebaut. Das bedeutet eine Reihung von Operationen, die unter der Kontrolle eines DBMS (Database Management System) ausgeführt werden: Die globale Transaktion in einer Umgebung von multiplen Datenbanken und die verteilte Transaktion. Die Kangaroo Transaktion besteht aus sechs formalen Definitionen: eine LT (lokale Transaktion), eine GT (globale Transaktion), eine JT (Joey Transaktion), eine KT (Kangaroo Transaktion), Pouch und Äquivalenz Kangaroo Transaktionen. Die Datenstrukturen des mobilen Transaktionsmanagers werden durch Funktionen von dem MTM dargestellt. Die Kontrolle zur Ablaufsteuerung der Kangaroo Transaktion wird ebenfalls über den MTM geregelt. In der Forschung erfasst kein anderes Modell die Eigenschaften der Bewegungen von mobilen Transaktionen besser als die Kangaroo Transaktion.

Zusammenfassung: Die Implementierung der Bewegungstransaktionen muss Nebenläufigkeit, Atomarität und Wiederherstellung unterstützen. Ein neues Modell der Transaktionen bietet die Eigenschaften von korrektem Verhalten bei “Disconnection” und zur Anordnung für die verschiedenen Charakteristiken der Netzwerke an. Hier stellt sich eine optimale Form von mobilen Transaktionen dar, weil sie analog zu den Transaktionen von kommerziellen Datenbanken entwickelt wurden. Wenn ein Agent in **SeMoA** migriert, muss er normalerweise mindestens in einer ähnlichen Form die obigen Aspekte wie Nebenläufigkeit, Atomarität und Wiederherstellung berüchsichtigen - diese Aspekte weden dann bei **SeMoA** lediglich anders genannt werden. Das Modell von **SeMoA** ist allerdings anders aufgebaut und es fehlen **SeMoA** noch die Aspekte der Transaktion - ähnlich den Datenbanken im obigen Modell - die eine “bessere” Migration bei **SeMoA** gewährleisten könnten.

1.5 SeMoA

SeMoA ist die Abkürzung für “Secure Mobile Agent”[1]. Dies ist ein Projekt am Fraunhofer Institut für graphische Datenverarbeitung in Darmstadt. In dem werden erweiterbare und offene

[1] www.semog.org

Server für mobile Agenten entwickelt werden. Die Server werden vollständig in der Programmiersprache JAVA^{TM} implementiert. Entwicklungsschwerpunkt von **SeMoA** ist die Bereitstellung von Sicherheitssmechnismen, um den beiden Problemstellunggen des "Malicous Host" bzw. "Malicous Agent" zu begegnen. "Malicicous Host" bezeichnet z.B. den Fall, dass ein böswilliger Server die mobilen Agenten angreift, indem er Daten, Code und Ausführungszustände der Agenten manipuliert. Dadurch kann ein mobiler Agent Schaden auf anderen Servern anrichten z.B. in Form eines Trojanischen Pferdes. Außerdem kann ein böswilliger Server bösartige Agenten zu anderen Servern oder Agenten übertragen, d.h. Viren oder Würmer.

1.5.1 Systemarchitektur

SeMoA besteht in der Grundform aus einem stationärem Teil, der SeMoA-Plattform und einem mobilen Teil, den **SeMoA**-Agenten. Die Architektur von **SeMoA** kann grob in folgende Konzepte aufgeteilt werden [24]:

- **SeMoA**-Plattform
- **Migration**
- **Lokalisierung**
- **Kommunikation**

In dem **SeMoA**-Abschnitt dieser Arbeit werden zwei Komponenten, die **SeMoA**-Plattform und die Migration berücksichtigt. Der Kern der **SeMoA**-Plattform fasst die "Shell", das "Environment" und inhärente Sicherheitsmechanismen zusammen. Auf dieser Basis werden dann lokale Dienste und mobile Agenten ausgeführt.

Wenn die **SeMoA**-Platform startet, erfolgt die Konfiguration des Agenten-Servers über das Laden der Shellskripte, die die Laufzeitumgebung für Dienste und Agenten initalisieren. Um die verschiedenen Dienste (Services) zu benutzen, die einen besonderen Zusammenhang mit der erkennbaren modularen Bauweise der Plattform haben, muss der Benutzer sie im "Environment" registrieren. Der Administrator kann Dienste während der Laufzeit entfernen und registrieren, welche über die Shell, Agenten, sowie Dienste durch entsprechende Java-APIs zur Verfügung bestellt werden. Der Agent kann nicht direkt auf die Dienste zugreifen, welche die Plattform-**SeMoA** anbietet. Um den Zugriff auf diese Dienste zu ermöglichen, liegen diese Dienste im Environment. Es gibt nur die vier Klassen `Environment`, `Mobility`, `Variables` und `Communication`, die ein Agent direkt ansprechen kann. Für den Fall, dass der Zugriff auf diese Dienste dem Zugriffsrecht entspricht, wird für den Nutzer eine Sicht auf diese Dienste dargestellt.

Die Shell gestaltet sowohl die Registrierung der Dienste als auch die Wechselwirkung der Plattformen in einer Weise, die der gängigen Unix Shell nachempfunden wurde. Zu dem kann durch das Kommndo "java" eine beliebige Java-Klasse instanziiert und im Rahmen von **SeMoA** ausgeführt werden.

Wenn eine Schnittstelle von einfachen Agenten implementiert wird, wird die Schnittstelle `Runnable` (bzw. dessen **SeMoA**-Erweiterung `Resumable`) benutzt, zusätzlich muss ein Agent das Interface `Serializable` imlementieren, um den Vorgang der Migration zu ermöglichen.

Jeder Agent in **SeMoA** wird über das Interface `AgentContext` zusammengefasst. Die Klasse `AgentContext` besteht aus: `Lifecycle`, `ClassLoad` und eigener `ThreadGroup`, die jeder Server in einer Multi-Thread Umgebung ausführt. Der Agent wird in zwei Teile geteilt:

- **Den statischen Teil**, der unverändert bleibt: Der statische Teil enthält Klasen plus initiate Daten und Parameter. Der implizite Name eines Agenten wird von der Signatur dieser Daten abgeteitet.
- **Den dynamischen Teil**, der sich in der Lebenszeit (Lifecycle) des Agenten ändern kann.

Während der Migration läuft der Agent durch Signal-Filter, in denen der statische Teil des Agenten sowie der Agent im Ganzen gleichzeitig verifiziert wird. Um die Adresse der Agenten zu lokalisieren, bietet die **SeMoA**-Plattform das Verfahren ATLAS ("Agent Tracking and Location Service") mit dem Protokolltyp "ship" an, z.B. `ship://AgentServerX.DomainB.net:-47470.` d.h. diese Addresse ist Aufenhaltsort des Agenten sowie Identifikation des Server. Die Agentenprogrammierung betrifft Aspekte wie die Lebenszeit der Agenten, die Aufgaben welche Agenten erfüllen sollen, die Migrations- und Kommunikationsziele, welche die Parameter der Agenten der **SeMoA**-Plattform bestimmen, welche auf der anderen Seite die Laufzeitumgebung, Dienste und Ressourcen zur Verfügung stellt.

1.5.2 Migrationskomponenten

Die Migration ist der wesentliche Aspekt der mobilen Agenten. Deshalb gilt ihr die Aufmerksamkeit beim Entwickeln von **SeMoA** in besonderer Weise.

Wenn ein mobiler Agent migriert, durchläuf er eine Pipline von Sicherheitsfiltern. Der Agent, bestehend aus Programmcode, Daten, serialisiertem Zustand sowie verschiedener Meta-Daten (z.B. Signaturen), wird über ein sogenanntes Gateway als JAR (Java-Archive) verschickt. Es gibt zwei Arten von "Gateways": Ein Gateway ist für den Empfang (`InGate` mit den Filtern `VerifyFilter` und `DecryptFilter`) und eines für das Versenden von Agenten (`OutGate` mit den Filtern `EncryptFilter` und `SignFilter`) zuständig. Beim Transport der Daten werden eine Reihe von Filtern (siehe Abbildung 1.2) durchlaufen. Daneben wird die Integrität der JAR-Datei durch eine digitale Signatur nach PSKC7 [2] geschützt. **SeMoA** ermöglicht das selektive Signieren und Verschlüsseln von Inhalten in der JAR-Datei. Wie in Abschnitt 1.5.1 beschrieben, wird die JAR-Datei des Agenten in zwei Teile geteilt: Den statischen Teil und den dynamischen Teil, der auch den Teil für das Verschlüsseln und die Signatur enthält. Der Agent kann nicht nur Klassen im statischen Teil, sondern auch Dateien wie Bildformate und Textformate im dynamischen Teil in der gepackten JAR-Datei mitbringen. Normalerweise werden Klassen nicht von dem Agenten mitgebracht, wenn diese Klassen im Zielsystem schon existieren, um den Vorgang der Migration zu optimieren. Im Zielsystem werden die Signaturen des Agenten geprüft und der Agent teilweise entschlüsselt (Deserialisierung), dann wird er entpackt, bevor ihn die **SeMoA**-Plattform ausführt.

Um die Migration in **SeMoA** zu realisieren, muss der mobile Agent ein Ticket setzen. Dieses enthält die URL des Zielsystems.

1.5.3 Sicherheitsarchitektur

Der Agenten-Server stellt einen wesentlichen Teil der Infrastruktur bereit, auf deren Basis mobile Agenten migrieren und miteinander kommunizieren können. Hier bieten die Sicherheitsme-

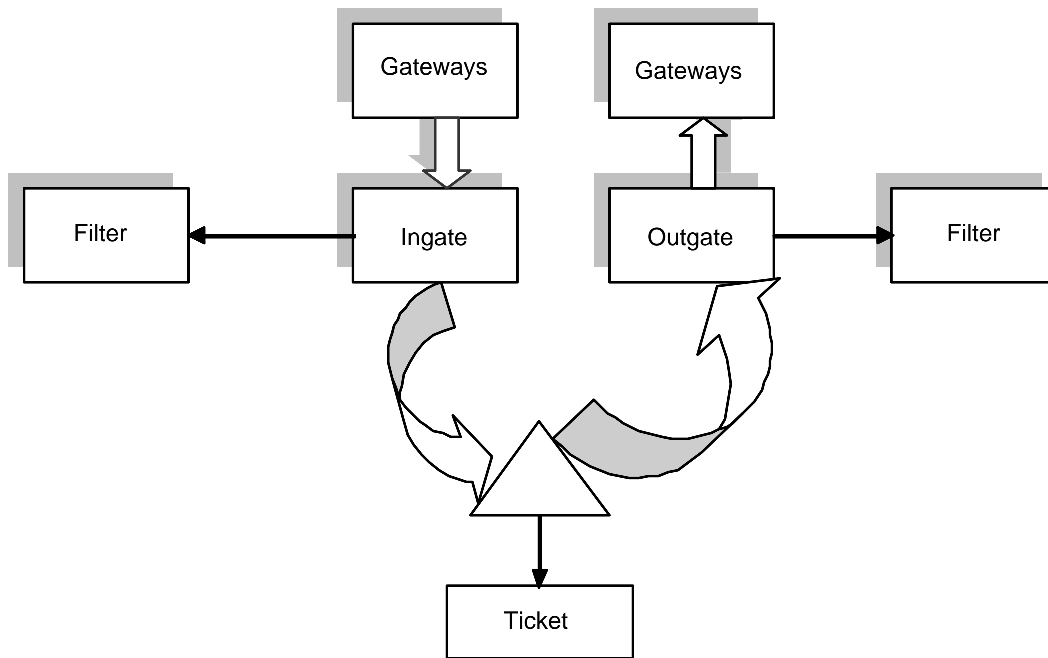

Abbildung 1.2: Transport

chanismen von **SeMoA** eine größtmögliche Sicherheit an, um Daten und Code der Agenten bei der Migration zu schützen. Die erste wichtige Aufgabe für den Agenten-Server ist die saubere Trennung von den Agenten und dem Server sicherzustellen. Um sauber zwischen den Agenten und dem Server zu trennen, finden Methoden der Code-Trennung, Thread-Trennung und Objekt-Trennung Verwendung. Außerdem darf der Besitzer eines Agenten nicht direkt auf einen andere Agenten zugreifen können. Einzige Interaktionsmöglichkeit zwischen Diensten und Agenten stellt die Indirektion über des Environment dar. Die zweite Aufgabe ist die Verhinderung von schädlichen und gefährlichen Funktionen von Agenten und die dritte ist das Schützen des Wirtssystems gegen unzulässige Zugriffe.

Die Sicherheitsarchitektur eines Servers (siehe die Abbildung 1.4 für mobile Agenten) lässt sich in vier Schichten darstellen:

- **Transport Layer Security**: Sicherheit besteht hier für die Migration der Agenten über eine Transportschicht. Diese Aufgabe übernimmt die äußerste Schicht. Das Protokoll SSL/TLS bietet gegenseitige Authentifizierung für den Agenten-Server sowie den Schutz der Integrität und Vertraulichkeit der übertragenen Daten.

- **Content Inspection**: Die untersuchten Daten werden durch zwei Pipelines (der eingehende und ausgehende Agenten) gefiltert. Jeder Filter aus diesen Pipelines realisiert eigene Filterprozesse und entscheidet, ob der Agent akzeptiert oder verworfen wird. Diese Filterprozesse wurden im Vergleich zu dem Konzept von Firewalls erstellt. Das Verfahren der Filter bietet die Sicherheit des Authentifizierens nicht nur für die Migration im Netzwerk, sondern auch für die Bewegung der Daten bei der Lokalisierung an.

- **Sandbox und Environment**: Wenn die Agenten im Zielsystem ausgeführt werden, benutzen sie außer den vier Klassen `Environment`, `Mobility`, `Variables` und `Commun-`

`ication` nur die Dienste des Environments (siehe Abschnitt 1.5.1). Um die Dienste von anderen Agenten zu nutzen, erzeugen Agenten einen “Dynamic Proxy” wie in Abbildung 1.3, in dem die Dienste für die Benutzung der Agenten im “gemeinsamen Environment” gekapselt werden. Es gibt noch eine andere Sicherheitsmethode für Agenten, “Implict Name” genannt (siehe Abschnitt 1.5.1). In der Sandbox-Schicht setzt der Agenten-Server nach dem Filtern der Daten die Sandbox auf und die Agenten werden dort gestartet. Zwei Aspekte der Sandbox sind zu betrachten: `ThreadGroup` und `Thread` eines Agenten verhindern böswillige Agenten, und Kopien (Klone) werden durch den impliziten Bezeichner erkannt: Der `ClassLoader` prüft die Integrität durch eine konfigurierbare Menge an Hashfunktionen. Deswegen werden Trojanische Pferde in dem Namensraum eines Agenten verhindert. Das Environment kontrolliert den geregelten Zugriff auf die Ressourcen, die einem Agenten zur Verfügung stehen. Das Environment hat auch wieder zwei Aspekte: Der erste Aspekt verhindert, dass ein Agent, der ein Objekt referenziert, das Trojanische Pferde in sich birgt, dieses an andere Agenten weitergibt. Der zweite Aspekt stellt sicher, dass falls ein Agent terminiert, alle Objekte von dem Agenten durch den Server gelöscht oder ungültig gemacht werden.

Die Sicherheitsarchitektur ist eine der wichtigsten Aspekte von **SeMoA**, wie man schon am Namen des Projektes erkennen kann. Wenn ein Agent migriert, könnte ein bösartiger Server die Daten des Agenten manipulieren, um durch den Agenten einem anderen Server zu schaden. Andererseits kann ein bösartiger Agent versuchen unauthorisierten Zugang auf einen Server zu bekommen. Deshalb werden in **SeMoA** Schutzmechanismen des Agentensystems für folgende Szenarien entwickelt:

- **Angriffe des Agenten auf Agentenserver**
- **Angriffe des Agentenservers auf Server**
- **Angriffe zwischen Agenten**
- **Verändern und Ausspähen von Agenten während des Transports**
- **Verändern und Ausspähen anderer Daten**

Wenn der Agent migriert, muss er alle Schichten der Sicherheitsarchitektur in **SeMoA** durchlaufen, bevor er im Laufzeitsystem zugelassen wird und die erste Klasse eines Agenten in die “Java Virtuell Maschine” des Servers geladen wird.

Ein Agent von **SeMoA** wird von anderen Agenten im Server strikt getrennt. Im Moment ist es nicht möglich, die Objektreferenzen direkt zwischen den Agenten auszutauschen. Damit ein Agent die Dienste der anderen Agenten und umgekehrt benutzen kann, werden die Dienste dieser Agenten im globalen Environment erzeugt, weil es nur das globale Environment ermöglicht, Informationen bzw. Objekte zu publizieren oder auszutauschen.

Um die Daten des Agenten in einem fremdem Wirtssystem zu schützen, realisieren die Basismechanismen in **SeMoA** digitale Unterschriften und Verschlüsselung. Für statische Daten des Agenten, die nicht verändert werden dürfen, werden ihre Authentifizierung und Integrität durch digitale Unterschriften sichergestellt, um unveränderte Daten zu gewährleisten. Es ist auch möglich, dass die Geheimhaltung der Daten der Agenten durch anwendungsabhängige Sicherheitspolitiken für jeden einzelnen Agenten realisiert wird.

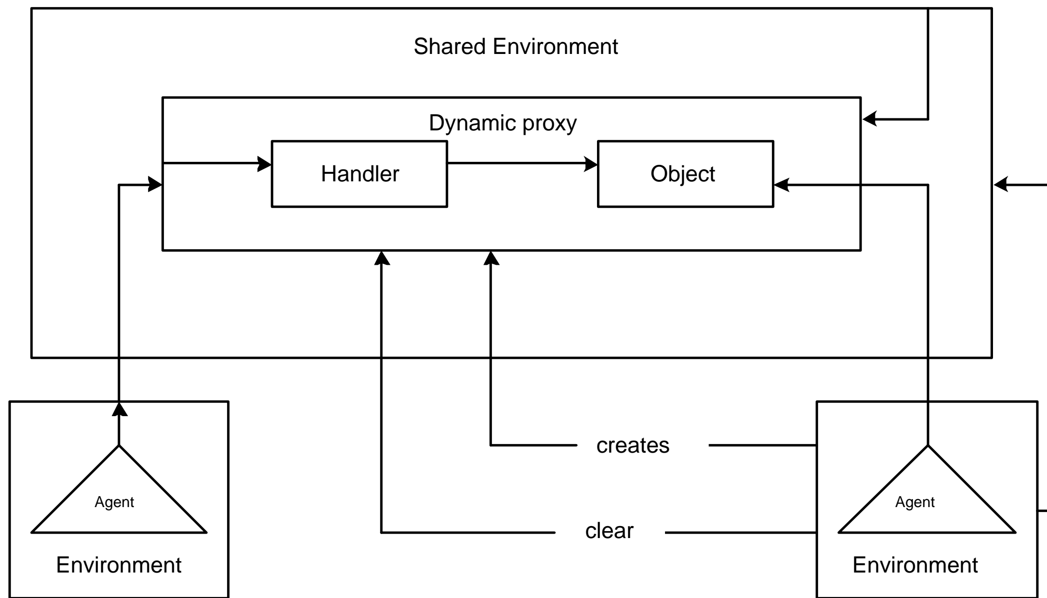

Abbildung 1.3: Dynamic Proxys

Es gibt drei verschiedene Arten von Daten in der Agentenstruktur: Statische Daten, veränderliche Daten, Meta-Daten. Um sie auf dem fremden Wirtssystem zu schützen, werden als Basismechanismen die digitale Unterschrift und Verschlüsselungen benutzt. Dadurch wird die Integrität und Vertraulichkeit der statischen Daten während der Migration geschützt. Ein Dateisystem wird als eine Abstraktion für die Struktur von Agenten ausgewählt. Dann wird eine kryptographische Behandlung zum Auslagern der Nutzdaten innerhalb des Dateisystem benutzt: Andererseits wird der Schutz der Kooperation zwischen Agenten berücksichtigt. Ein Mechanismen wird angeboten, um eine Bedrohung zwischen interagierenden Agenten zu vermeiden. Das Problem der bösartigen Wirtssysteme ist die größte Herausforderung an Sicherheitsmechanismen für mobile Agenten.

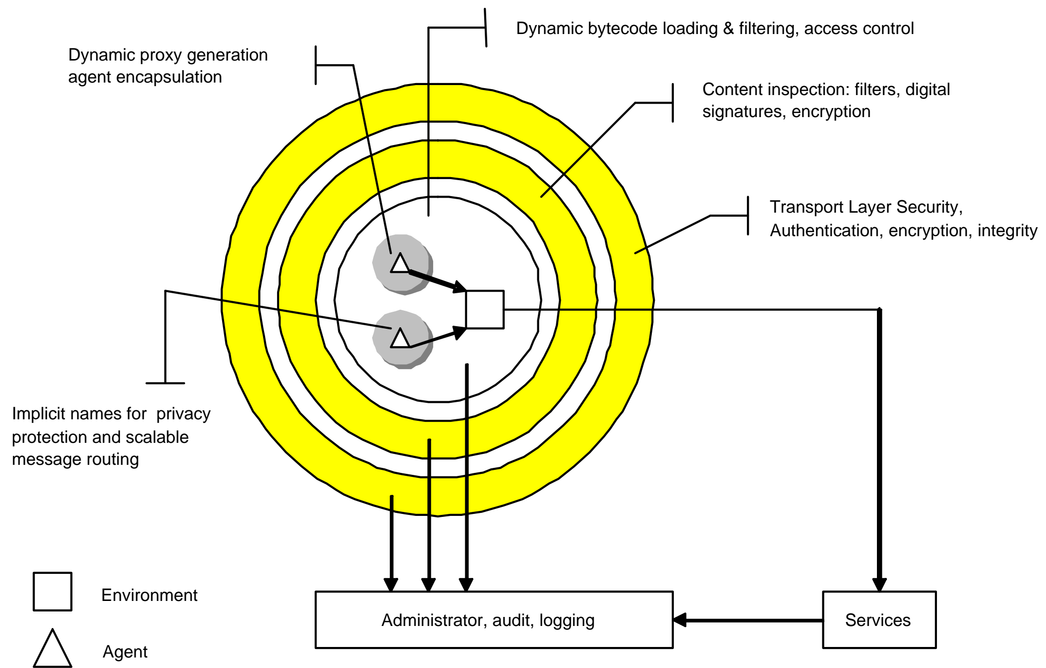

Abbildung 1.4: **SeMoA**-Security

Kapitel 2

Protokollentwicklung: Transaktionsbasierte Migration

2.1 Anforderungsanalyse

2.1.1 Sicherheitsaspekte

Die Sicherheit von Agentensystemen ist - im Hinblick auf den Schutz von Rechnerplattformen und Kommunikation - in zahlreichen Arbeiten bereits ausgiebig diskutiert worden; hier werden die wichtigsten Punkte noch einmal aufgeführt. Dabei stehen Methoden zur Ressourcenverwaltung und Nutzung sowie zur Zugangskontrolle im Vordergrund. Das Hauptaugenmerk dieses Abschnitts richtet sich jedoch auf das weitgehend ungelöste und daher besonders kritische Problem von Sicherheit und Integrität auf Seiten der mobilen Agenten selbst.

2.1.1.1 Grundsätzliche Sicherheitsbedürfnisse mobiler Agenten

Grundsätzlich sind folgende Sicherheitsdienste zum Schutz von Agenten erforderlich oder anzustreben:

- **Authentifikation:**
  - **der Benutzer des Agentensystems (Wer hat Zugriff?),**
  - **der beteiligten Rechner (Agentenplattformen) untereinander (Ist mein Gegenüber wirklich der, für den ich ihn halte?),**
  - **des Programmcodes (Vertrauenswürdige Implementierung?) und**
  - **der Agenten selbst (Kann ein verantwortlicher Besitzer zugeordnet werden?).**
- **Integritätskontrolle**: Wurde der Programmcode des Agenten oder seine Daten manipuliert?
- **Geheimhaltung**: Wie kann ich meine Daten vor dem Zugriff Dritter schützen?
- **Autorisierung**: Welche Ausführungsrechte werden dem Agenten zugebilligt?
- **Unabstreitbarkeit von Nachrichten und Aktionen**: Wer ist der Urheber?

- **Buchführung über sicherheitsrelevante Vorkommnisse zur späteren Auswertung**: Wer hat wann welche Aktion ausgeführt?
- **Anonymität und Schutz der Privatsphäre**: Können Aktivitäten einem bestimmten Teilnehmer ohne dessen Wissen oder ausdrückliche Zustimmung zugeordnet in einem Benutzerprofil erfasst werden?

2.1.1.2 Angriffe gegen mobile Agenten

Sobald mobile Agenten jedoch ihren Ursprungsrechner verlassen haben, sind sie Ziel potentieller Angriffe. Darunter fallen:

- **Ausspionieren oder Manipulation von**:
  - **Code,**
  - **Daten,**
  - **Kontrollfluss,**
  - **Kommunikation zwischen Agenten,**
  - **Interaktion mit anderen Agenten.**
- **Abfangen**: Das Abfangen von Agenten bei deren Migration zur nächsten Rechnerplattform bzw. Überwachen des Migrationsverhaltens, um Rückschlüsse auf Art und Umfang der Agentenaktivitäten und somit indirekt auf das übergeordnete Benutzerverhalten zu erhalten.
- **Maskierung**: Durch Maskierung der eigenen Identität versucht ein Rechner (oder eine andere Partei), dem Agenten gegenüber eine andere Identität vorzuspielen, um den Agenten zu bestimmten Handlungen zu verleiten oder geheime Daten zu erschleichen.
- **Dienstverweigerung**: Die Ausführung des Agenten kann gezielt verweigert oder verzögert werden. Fehlerhafte oder manipulierte Agenten können ebenfalls durch störende Interaktion andere Agenten blockieren.
- **Fehlerhafte Programmausführung**: Anstatt den Ablauf im Agentenprogramm zu ändern, wird der Kontroll?uss zum Zeitpunkt der Ausführung des Programmcodes auf dem Host direkt manipuliert, um das Programmverhalten gezielt zu beeinträchtigen, z.B. durch das Verändern von Speicher- oder Registerwerten.
- **Verfälschte Systemaufrufe**: Systemaufrufe werden gezielt maskiert oder manipuliert, um dem Agenten falsche Tatsachen vorzuspielen (z.B. falsche Rechneradresse) und damit seine korrekte Ausführung zu verhindern.

Da sich der Agent bei seiner Ausführung dem gastgebenden Rechner anvertrauen muss, liegen dort auch die meisten potentiellen Gefahrenstellen und Angriffsmöglichkeiten.

2.1.1.3 Schutzmaßnahmen

Jede Plattform stellt für sich eine Sicherheitspolitik (engl. security policy) auf, die z.B. die beteiligten Entitäten bestimmt und deren Rechte definiert. Eine Vielzahl von Sicherheitsmechanismen beugen dem Missbrauch von Ressourcen und unerlaubtem Verhalten von Seiten der Agenten vor. Eine umfangreiche Analyse von Maßnahmen zum Schutz von Agentenplattformen liefern Jansen und Karrygiannis in [16] sowie Pleisch in [29], wie beispielsweise signierter Code, beweismitführender Programmcode oder das Sandkastenmodell (sandbox model). Die Probleme des Schutzes der Migration und der Kommunikation bei mobilen Agenten sind weitestgehend gelöst. Sichere Kommunikation kann mittels bekannter Verfahren zur Authentifkation und mittels Verwendung sicherer Kanäle bewerkstelligt werden. Die sichere Übertragung von Agenten wird durch entsprechend sichere Protokolle erreicht - entweder direkt durch das zuständige Transferprotokoll, wie z.B. das Agent Transfer Protocol in [18], oder über sichere Punkt-zu-Punkt-Verbindungen mittels der Etablierung kryptographisch geschützter Kanäle.

- **Schutz der Mobilen Agenten**: Eine potentielle Gefahrenquelle für mobile Agenten stellen andere Agenten dar, die in derselben Agenten-Umgebung ausgeführt werden. Um eine sichere Ausführung von Agenten zu gewährleisten, werden diese entweder in isolierten Adressräumen ausgeführt, was auch als isolierte Ausführung bezeichnet wird. Oder die Kapselung und der Schutz privater Daten wird durch Einhaltung einer strengen Typisierung, wie beispielsweise in der Programmiersprache Java geschehen, erzwungen. Die größte zu bewältigende Gefahr für mobile Agenten bleibt das Auftreten bösartiger Rechnerplattformen (Malicious Host), von denen die Mehrzahl der Angriffe ausgeht. Das Einschleusen von fehlerhaften oder bösartigen Klassen wird durch die strikte Zuweisung von genau einem `Classloader` je Agent verhindert. Hierbei ist jedoch zu berücksichtigen, dass auf unsicheren Plattformen dieser Mechanismus selbst Gegenstand von Manipulationsversuchen sein kann, und dessen Zuverlässigkeit daher nicht immer gewährleistet ist. Manipulationen am Agenten selbst sind kaum zu verhindern. Mit Hilfe von Signaturen kann zumindest die Integrität der zum Zeitpunkt der Initialisierung festgeschriebenen unveränderlichen Komponenten eines Agenten vor dessen Ausführung überprüft werden. Zur Authentisierung werden digitale Signaturen eingesetzt, die in der Regel auf asymmetrischen Schlüsselpaaren (öffentlicher und privater Schlüssel) basieren, z.B. mittels dem El Gamal- oder dem RSA-Signaturverfahren nach Rivest, Shamir und Adleman [10] und [30]. Dies setzt jedoch auch das Vorhandensein einer entsprechenden PKI (Public-Key-Infrastruktur) voraus. Damit geheime Daten des Agenten nicht von unbefugten Dritten eingesehen werden können, finden die bekannten kryptographischen Verfahren zur Verschlüsselung Anwendung. Je nach Eignung und Bedarf werden die effizienteren symmetrischen Verschlüsselungsalgorithmen (z.B. Advanced Encryption Standard AES) oder die im Funktionsumfang mächtigeren asymmetrischen Public-Key Verfahren (z.B. RSA) verwendet. Um aber auch das unbefugte selektive Zerstören oder Ersetzen von Datenfragmenten durch Dritte zu verhindern, sind umfassendere Schutzmaßnahmen vonnöten.

- **Verschlüsselte Funktionen**: In diese Richtung zielen die noch überwiegend theoretischen Ausführungen von Sander und Tschudin [37] und [36]. Mittels verschlüsselter Funktionen (En- crypted Functions, Function Hiding) soll die korrekte Ausführung von Agenten auf unsicheren, evtl. bösartigen Plattformen ohne die Verwendung von Spezialhardware ermöglicht werden. Die entwickelten Ansätze sind jedoch zur Zeit noch auf Polynome und rationale Funktionen beschränkt und scheinen nur in sehr begrenztem Maße geeignet zu sein,die gestellten Aufgaben in punkto Sicherheit zu lösen.

- **Detection Objects**: Meadows schlägt die Verwendung von so genannten Detection Objects, Erkennungsobjekten, vor [34]; das sind anwendungsspezifische Dummy-Objekte, deren alleinige Funktion darin liegt, unerlaubte Manipulationsversuche an den Daten (Objekten) mobiler Agenten zu erkennen. Die Grundidee ist, dass sowohl der Agent als auch die ausführende Plattform nicht wissen, welche Datenobjekte die Erkennungsobjekte darstellen. Im Falle eines Angriffes werden mit sehr hoher Wahrscheinlichkeit auch die Dummy-Objekte modifiziert und diese Modifikationen später durch Objektvergleich festgestellt. Voraussetzung ist jedoch, dass die Detection Objects nicht von den eigentlichen Objekten unterschieden werden können, auch nicht bei mehreren Durchläufen, und dass sie das Verhalten und die Entscheidungsfindung von Kooperationspartnern auf fremden Plattformen nicht beeinflussen.

- **Fehlertoleranz**: Andere Überlegungen folgen dem Ansatz der Fehlertoleranz. Sie setzen dabei einerseits auf eine redundante Ausführung durch den gleichzeitigen Einsatz replizierter Agenten, die sich nach jeder Bearbeitungsstufe durch Abstimmung auf ein gemeinsames gültiges Ergebnis einigen. Die Grundprinzipien sind also Replikation von Agenten und Mehrheitsentscheid [33]. Die Nebenbedingungen dabei sind jedoch u.a. die Verfügbarkeit von Replikationen des zu besuchenden Servers; ebenso ist deren unerlaubte Kooperation nicht auszuschließen, so dass erhaltene Resultate nicht notwendigerweise verlässlich sind. Die geforderte Redundanz an (identischen) Servern ist zudem teuer und in der Praxis oft nicht gegeben. Pleisch untersucht in [29] auch die Steigerung der allgemeinen Zuverlässigkeit von Mobile-Agenten-Systemen durch deren fehlertolerante Auslegung. In [38] wird die sichere Replikation von Diensten besprochen, auch mit Bezug zur Schwellenwert-Kryptographie (engl. threshold cryptography), bei der Geheimnisse oder Funktionen von einer Gruppe von Benutzern geteilt werden [32], [8] und [35].

- **Software Blackbox**: Hohl hat ein Verfahren entwickelt, das mobile Agenten für eingeschränkte. Zeiträume zur sicheren software-basierten Blackbox werden lässt [14]. Es beruht auf Algorithmen, die durch geschickte Zerlegung und Rekomposition von Code-Fragmenten des Agenten bestimmte Vorgänge verschleiern. Dadurch wird angestrebt, dass zumindest innerhalb eines festgelegten Zeitraumes die Analyse und sinnvolle Manipulation des Agenten verhindert wird, und dass geheime Daten im zeitlichen Gewährleistungsintervall nicht unerlaubt für anderweitige Zwecke missbraucht werden können. Gegenstand solcher Verschleierungsaktionen sind Variablen, Schlüssel und der Kontroll?uss selbst. Schwachstellen sind hierbei jedoch, dass die Güte und Effektivität der verwendeten Verschleierungs-Algorithmen nur schwer ermittelt werden kann, und dass die Wahl des zeitlichen Gültigkeitsintervalls - die insbesondere im Hinblick auf die asynchrone, zeitlich entkoppelte Arbeitsweise mobiler Agenten - unter Umständen erhebliche Probleme bereiten kann.

- **Sichere Protokolle**: Ein protokollbasiertes Verfahren zum Schutz sich frei bewegender Agenten wurde von [41] vorgestellt und durch Karjoth et al. verbessert [17]. Eine Ausprägung des Algorithmus basiert auf dem Vorhandensein einer PKI mit eindeutig Zuweisbahren digitalen Unterschriften für jeden beteiligten Server. Eine Alternative dazu ist ein Mechanismus, der ohne eine solche PKI auskommt und stattdessen die zu schützenden Daten auf Seiten des Agenten mittels Hash-Ketten sichert. Das Protokoll erlaubt, die Integrität des Agenten auch unterwegs auf unsicheren Rechnern zu überprüfen, seine Daten gegen Manipulation zu schützen und sie geheim zu halten. Mit Hilfe einer Public-Key- Infrastruktur wird außerdem die Nichtabstreitbarkeit der Herkunft von gesammelten Daten

realisiert. Damit lässt sich der Zustand eines Agenten samt Datenbestand bis zum Besuch des ersten bösartigen Rechners schützen. Anschließend sind jedoch z.B. Replay-Angriffe möglich, in deren Verlauf der Agent durch eine ältere, zuvor gespeicherte Kopie ersetzt wird, so dass die dazwischen gesammelten Daten unbemerkt entfernt werden können.

- **Vertrauenswürdige, sichere Hardware**: Der zurzeit einzige Weg, die Ausführung mobiler Agenten auf unsicheren, möglicherweise bösartigen Rechnerplattformen gegen Manipulation und Ausspionieren zu schützen, ist die Verwendung von sicheren, vertrauenswürdigen Geräten. Diese Geräte erscheinen bezüglich Ihrer Funktionsweise nach außen hin als ein schwarzer Kasten. Kritische Operationen des Agenten können nun so ausgelegt werden, dass sie nur innerhalb der speziellen sicheren Geräte ausführbar sind und sich somit dem Einflussbereich des zugehörigen Rechners entziehen. Die Bandbreite derartig vertrauenswürdiger, sicherer Geräte reicht von nur kreditkartengroßen SmartCards über komplette geschützte Rechner bis hin zur teuren Spezialhardware.

Die in Abschnitt 1.5.3 dargestellte **SeMoA**-Sicherheitsarchitektur vermindert viele Risiken im System, aber diese Sicherheitsaspekte sind noch nicht ausreichend, wenn der Agent in **SeMoA** migriert. Wenn der Migrationprozess des Agenten vollständig erfolgreich war, d.h. zum Rechner, von dem der Agent gerade geschickt wurde, kommt keine fehlerhafte Nachricht zurück und nachdem der Agent zum Zielort weggeschickt wurde, wird die Verbindung zwischen dem Sender und Empfänger verworfen. Deswegen bekommen Sender keine Information über ihren Agenten, der gerade weggeschickt wurde, von dem Empfänger zurück. Das ist ein großer Nachteil für die Benutzer, weil es möglich ist, dass es negative Zustände gibt, z.B. der Zielserver stürzt ab, der Zielserver lehnt die Agenten ab oder der Empfänger bekommt den Agenten des Senders gar nicht. Dies bedeutet aber auch weiterhin, dass diese Agenten dazu noch vollständig bei allen Servern gelöscht werden: Das ist ein bislang nichtakzeptabler Umstand. Um das Problem zu lösen, wird ein neuer Sicherheitsaspekt für ein Feedback zwischen Rechnern in **SeMoA** entwickelt. Diese Sicherheitstransaktionen in **SeMoA** mit Feedback werden detailliert im nächsten Kapitel beschrieben.

2.1.2 Rechtliche Aspekte

Bezüglich einer Simulationsstudie [23] wird die Rechtsverträglichkeit von **SeMoA**-Agenten in diesem Abschnitt kurz beschrieben. Darin werden auch technische Gestaltungsvorschläge untersucht, die das **SeMoA**-System verbessern können.

Die folgenden Annahmen muss ein Agentensystem erfüllen, um Delegation in der beschriebenen Form durch Agenten zu nutzen:

- **Jeder Agent hat eine eindeutige Identität, die nicht manipuliert werden kann und während des Lebenszyklus eines Agenten unverändert bleibt.**
- **Agenten werden auf sichere Weise durch das Agentensystem authentifiziert.**
- **Die Identität eines Agenten ist für andere Agenten und Dienste, die sich auf der gleichen Plattform befinden, verfügbar.**
- **Es ist eine Public-Key-Infrastruktur vorhanden, bei der private Schlüssel auf sichere Weise gespeichert werden und Zertifikate öffentlich verfügbar sind.**

Wenn ein Kunde zum Beispiel durch Agenten auf ein Bild eines Bildanbieters zugreifen möchte, muss der Agent zuerst authentifiziert werden. Um die Authentifizierbarkeit der Agenten zu realisieren, brauchen die Agenten die Hilfe von Tickets. Tickets setzen sowohl die Adresse des Zielortes, zu dem die Agenten migrieren, als auch vergeben sie feinkörnige und aufgabenspezifische Rechte der Agenten auf dem Zielsystem für eine begrenzte Zeit. Jedem Agent wird direkt ein Ticket zugeordnet, das die anderen Agenten nicht verwenden können. Tickets werden durch eine vertrauenswürdige und unabhängige Instanz ausgestellt, dessen Informationen für Rechte der Agenten über eine Policy zur Verfügung steht. Außerdem werden Tickets durch kryptographische Verfahren geschützt.

Bezüglich der Handhabung der Tickets differenziert die Simulationsstudie sich im Prinzip in vier Varianten zum Kauf von Bildern:

1. **Ein Kunde kauft ein Bild direkt bei einem bekannten Bildanbieter.**
2. **Ein Kunde kauft ein Bild über einen Broker, der als Vermittler zum Bildanbieter agiert.**
3. **Wie Variante 1 mit einem Abonnementen-Ticket.**
4. **Wie Variante 2 mit einem Abonnementen-Ticket.**

Im Rahmen der Simulationsstudie werden 4 Angriffstypen identifiziert und bewertet:

- **Angriffe aller Teilnehmer**
- **Angriffe der Kunden**
- **Angriffe der Broker**
- **Angriffe der Bildanbieter**

Außerdem nutzt ein Kunde eine Signaturkarte, um seine Daten durch kryptographische Operationen zu verschlüsseln und zu authentifizieren. Damit werden Mechanismen für eine Bildsuchmaschine möglich. Der Zugriff auf die Bilder des Bildanbieters wird in der Zeit beschränkt und ist kostenpflichtig. Technische Gestaltungsvorschläge werden auf der Basis von Angriffen aller Teilnehmer in diesem Abschnitt vorgestellt.

Die gewonnene Erfahrung aus dem obigen Kapitel dient als Ziel der technischen Gestaltungsvorschläge für einen Nachweis von Vorgängen auf den fremden Rechnern. Normalweise werden die Daten nicht vollständig auf einem lokalen Rechner konzentriert, sondern sie werden auf mehreren Rechnern und Serversystemen verteilt, damit die bösartigen Agenten oder Agentenserver die Daten nicht vollständig modifizieren können. Aber es gibt dabei auch Nachteile, da z.B. ein Sender, damit er wissen kann, dass sein Agent die Zielorte passieren, Nachweise von allen Rechnern braucht, in denen ihre Daten gespeichert werden: Dies kostet viel Zeit. Deswegen ist die beste Lösung in diesem Fall, dass jedes Mal, wenn ein Agent zu einem Zielort weggeschickt werden, der Sender sofort die Empfangsbestätigungen von dem Empfänger zurückbekommt - d.h es braucht keine Nachweise von anderen Orten zu geben, in der die Daten der Agenten auch existieren. Die Protokolle für die Agenten-Migration sind nicht sicher genug, weil die Zustände in **SeMoA** noch manipuliert werden können. Deshalb braucht es dazu Protokolle, in denen die Empfänger eine Signatur zurück zum Sender schickt. Hier ist ein großer

Vorteil für die technische Gestaltung zu erkennen: Wenn der Sender keine Empfangsbestätigung vom Empfänger bekommt, weiß der Benutzer zumindest sofort, dass irgendein Problem beim Migrationsprozess des Agenten aufgetreten ist. Aus dem Abschnitt 2.1.1, welche die technische Gestaltung [31] ausführen, ergibt die Simulationsstudie folgende Sicherheitsaspekte:

- **Empfangsbestätigungen über den Zugang von Agenten**: Benutzer bekommen die signierten Empfangsbestätigungen von dem Empfänger über den Zugang eines Agent zurück. Wenn der Benutzer keinen Nachweis von dem Zielort hat, kann man erkennen, dass es ein Problem bei der Migration gab.
- **Unabhängiges Logging der Agenten**: Vor Gericht gelten selbst signierte Dateien oder selbsterstellte Loggings nicht als Beweismittel. Damit es offizielle Beweismittel gibt, braucht es ein Dritten, nicht aber den eigenen Rechner. Hierzu wird ein Logserver empfohlen auf dem die Agenten migrieren
- **Automatisierte Aushandlung von Datenschutz, - und Sicherheitspolicies**: Dieses Konzept wird ähnlich der P3P-Methode (Plattform for Privacy Preferences) übernommen.

2.2 Protokollspezifikation

2.2.1 Aushandlung von Migrationsparametern

Wie im obigen Abschnitt beschrieben, ist das Ziel dieser Diplomarbeit die Entwicklung eines Protokolls, um die Transaktionssicherheit der mobilen Agenten zu erweitern. Bislang wurde das Programm für **SeMoA** nur bis zu dem Übertragungserfolg eines Agenten des Senders entwickelt und nach diesem Übertragungserfolg wird dieser Agent im Rechner der Sender gelöscht. Im Rechner der Empfänger durchläuft dieser Agent Sicherheitsfilter aus Pipelines, bevor die Klassen dieses Agenten in der Laufzeitumgebung ausgeführt werden. Im Fall, dass Fehler entsteht, bevor die Agentenklassen ins Laufzeitsystem geladen werden, wird dieser Agent im Rechner des Empfängers abgelehnt. Deswegen existiert dieser Agent nicht mehr im Server und der Sender hat keine Information darüber, was passiert ist. Das ist ein ungünstiger Umstand für die Transaktionssicherheit. Im Rahmen dieser Arbeit wird ein Protokoll implementiert, das diese Probleme entsprechend berüchsichtigen wird.

Bezüglich obiger Problem wird ein Feedback für Transaktionssicherheit in **SeMoA** entwickelt, d.h. Nachweise werden zwischen Agentenplattformen ausgetauscht. Nach der erfolreichen Übertragung des Agenten vom Rechner des Senders wird der Agent dort noch nicht gelöscht. Die Verbindung zwischen zwei Rechnern wird gehalten, bis der Sender eine digital signierte Empfangsbestätigung von dem Empfänger bekommt, und erst danach wird dieser Agent im Sender gelöscht.

Während der Migration eines Agenten kann die Kommunikation zwischen Rechnern jederzeit abbrechen, z.B. wegen dem Absturz der Server. Deshalb werden Netzwerkfehler im Migrationsprozess besonders berücksichtigt.

Die Aushandlung von Migrationsparametern zwischen sendender und empfangender Agentenplattform kann unnötige Migrationen eines Agenten in **SeMoA** vermeiden. Diese Parameter bestehen aus der Größe des Agenten, der Identität (des Besitzers) und den verlangten bzw. zu erwartenden Rechten auf der Zielplattform. Im Bezug auf diese Parameter trifft der Rechner des

Empfängers die Entscheidung, ob der Agent abgelehnt bzw. angenommen werden. Die Parameter sollen im Hinblick auf die Sicherheitspolitiken der Plattformen verhandelt werden können.

Außer der obigen Aushandlung von Migrationsparametern werden auch Mechanismen für "Service Discovery" in verteilten System benutzt, um unnötige Migration des Agenten zu vermeiden. Die Mechanismen umfassen eine dynamische Auswahl der Zielplattform. Bevor die Agenten migrieren, fragen sie erst angebotene Dienstumgebung im Zielort an.

2.2.2 Migration als Transaktion

Wenn ein Agent migriert, durchläuft er zuerst die Sicherheitsfiltern des `OutGate` (Gateway) im Sender, bevor er weggeschickt wird. Dieses Gateway beinhaltet momentan die drei folgenden Sicherheitsfiltern:

1. **Guard**
2. **Encrypt**
3. **Sign**

Anschließend wird der Agent weiter zum Zielort weggeschickt. Bevor er dort ausgeführt wird, durchläuft er wiederum die sechs folgenden Sicherheitsfiltern im Empfänger durchgelaufen:

1. **Verify**
2. **Decrypt**
3. **Policy**
4. **Atlas**
5. **Guard**
6. **Arrival**

Die Migration basiert im Prinzip auf dem von der Informatik her bekannten Transaktionsbegriff. Bisher wird nach erfolgreicher Übertragung des Agenten (bestehend aus dem Programmcode, dem serialisierten Zustand, assoziierter Daten und Metadaten) die Migration von Seiten des Senders als erfolgreich wahrgenommen. Aus diesem Grund werden die Zustände des Migrationprozesses in diesem Teil der Arbeit weiter entwickelt und verbessert. Wenn ein Agent den Sender verlässt, wird die Verbindung zwischen zwei Agentenplattformen noch gehalten und ein Agent wird auch im Rechner des Senders noch nicht gelöscht. Es gibt die folgenden zwei Möglichkeiten, die auftreten kann, nachdem ein Agent weggeschickt wird:

1. **Es verlief alles fehlerfrei**, d.h. ein Agent kommt erfolgreich zum Rechner des Empfängers. Im Rechner des Empfängers durchläuft ein Agent Sicherheitsfilter aus Pipelines. Dieser Agent wird sequentiell durch sechs Filter im Empfänger überprüft, und dann sowohl alles hat funktioniert, als auch die Agentenklassen werden ins Laufzeitsystem geladen, schickt der Empfänger eine digital signierte Empfangsbestätigung zum Sender zurück. Danach löscht der Sender die Agenten im Rechner und schließt die Verbindung zwischen den zwei Plattformen.

2. **Die Fehlern passieren in der Migration beim Empfang**: Jedes Mal läuft der Agent durch alle Filter. Wenn ein Fehler auftritt oder ein Filter den Agent ablehnt, schickt der Empfänger sofort einen Nachweis zum Sender zurück, um den Sender zu informieren, dass es Fehler gab und der Agent vom Empfänger abgelehnt wird. Der Sender kann dann versuchen, den Migrationsprozess zu wiederholen.

Neben obigem Fall kann auch ein anderer Fall auftreten, dass der Empfänger keine Information zum Sender zurückschickt. Es gibt dann die zwei folgenden Möglichkeiten:

1. **Obwohl die Übertragung der Agenten komplett erfolgt ist (d.h. die Klassen des Agenten werden nach der erfolgreichen Überprüfung durch die Sicherheitsfilter im Laufzeitsystem ausgeführt), bekommt der Sender keine digital signierte Empfangsbestätigung vom Empfänger in einer bestimmten Zeitspanne. Deshalb weiß der Sender nicht, was mit seinem Agent im Rechner des Empfängers passiert ist. In diesem Fall wird der Sender ATLAS (Agent Tracking and Location System) abfragen, um zu erfahren, in welchem Rechner dies Agent existiert. ATLAS gibt eine Antwort zurück und wenn der Agent in irgend einem anderen Rechner existieren, wird der Agent im Rechner des Senders gelöscht und die Verbindung zwischen den zwei Plattformen wird geschlossen.**

2. **Die Agenten werden sequentiell über sechs Filter im Empfänger überprüft. Ein Agent wird durch diese Filter abgelehnt, und der Sender bekommt keinen Nachweis dafür von dem Empfänger. Wenn der Sender nach einer bestimmten Zeit keine Nachricht von dem Empfänger bekommt, fragt der Sender bei ATLAS an, ob dies Agent auf irgend einem anderen Rechner existiert. Wenn ATLAS ein "Nein" zurückschickt, versucht der Sender noch einmal den Migrationsprozess durchzuführen. Aber auch hier kann das Problem auftreten, dass ATLAS ein "Ja" zurückschickt, aber der Agent wird von dem Empfänger eigentlich abgelehnt, weil ATLAS der vierte Filter des Empfängers ist, über den der Agent geprüft wird. Wenn ATLAS den Agent akzeptiert, kann der Agent durch den fünften oder sechsten Filter auch abgelehnt werden. Im diesem Fall wird der Agent nur für kurze Zeit im Rechner des Empfängers existieren. In dieser kurzen Zeit fängt aber der Sender an, Anfragen zu stellen, ob der Agent auf anderen Rechnern existiert. Um das Problem zu lösen, ist die Zeit, die der Sender auf den Nachweis von dem Empfänger wartet, größer als die Zeit, die der Empfänger für den kompletten Migrationsprozess des Agenten auf der Seite des Empfänger braucht.**

Auf der Grundlage der obigen Beschreibung entstanden folgende Diagramme (siehe Abbildungen 2.1 und 2.2) für die Migration:

2.2.3 Protokollentwurf

Auf Grund der obigen Abschnitte wird das Protokoll für die Migration von Agenten in diesem Abschnitt erweitert. Um die Agenten zum Zielsystem wegschicken zu können, benutzt man die drei folgenden Messages:

- **Messages-ID**: 0 (Es bedeutet die Agent-Migration)
- **URL**: Absende Adresse des sendenden Hosts

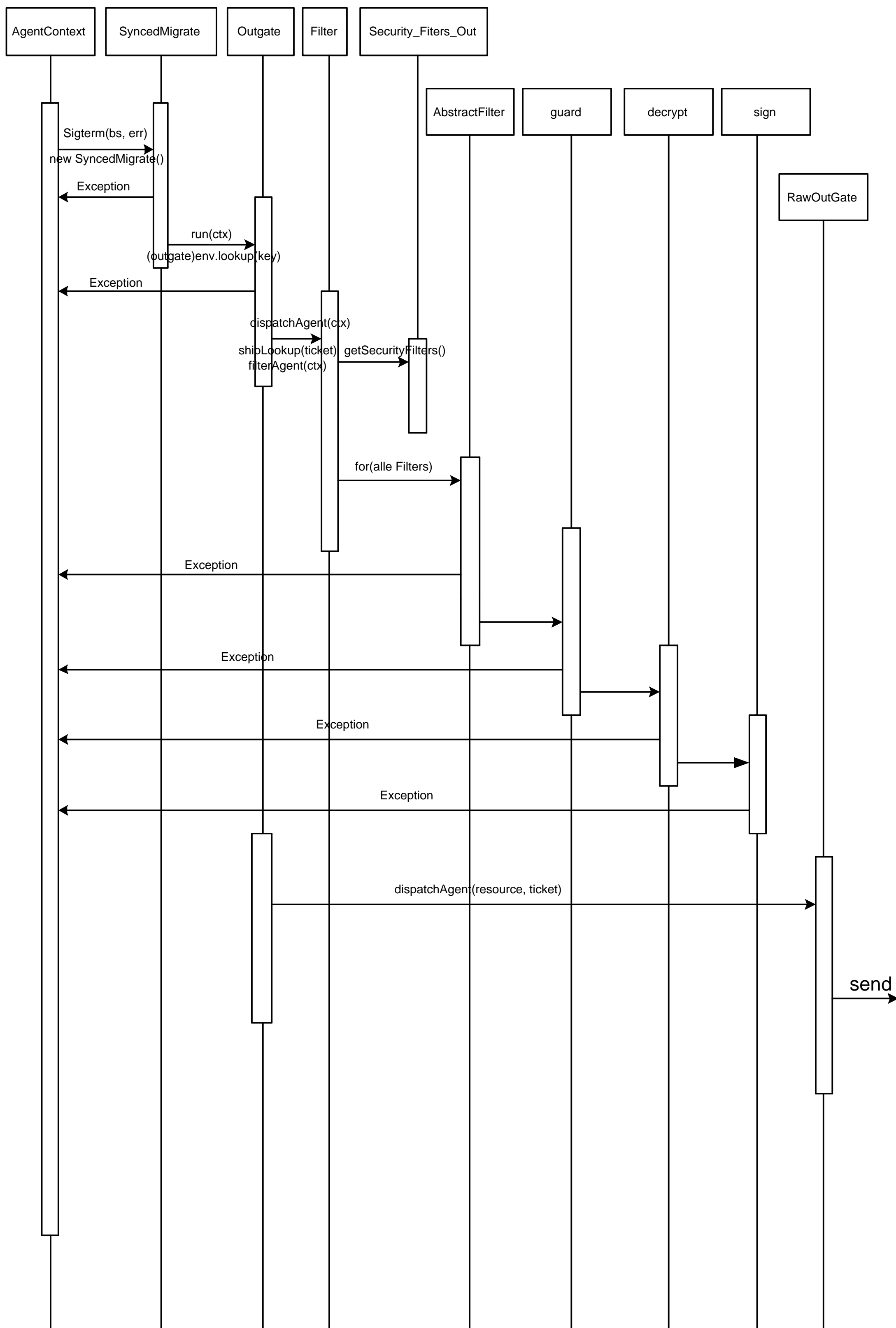

Abbildung 2.1: Diagramm Senden

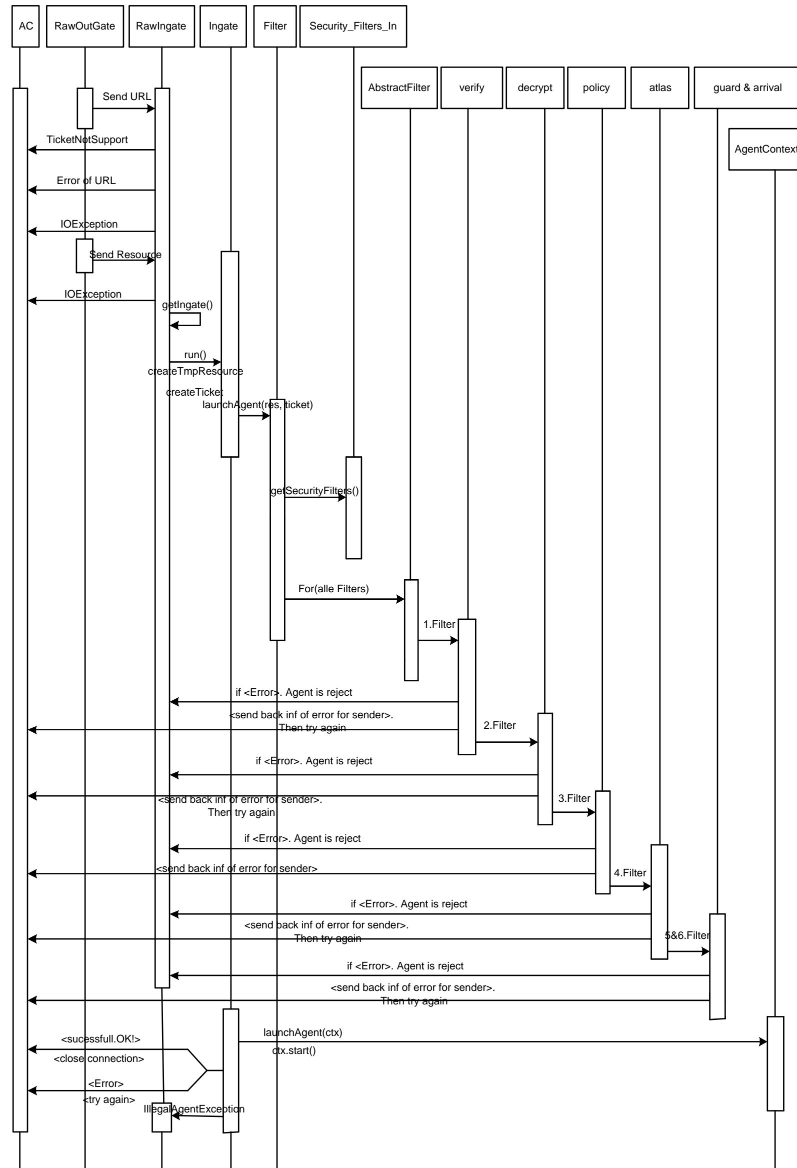

Abbildung 2.2: Diagramm Empfangen

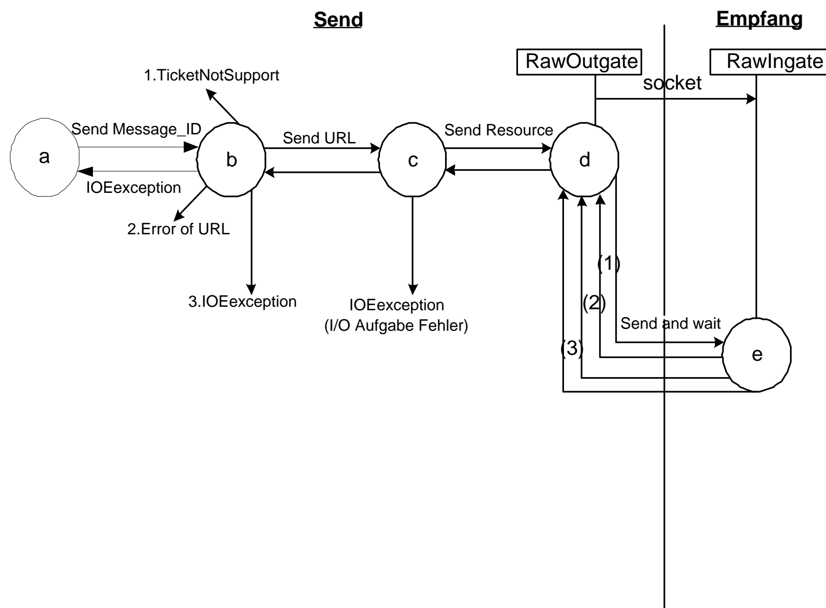

Abbildung 2.3: Diagramm Protokoll

- **Resource**: Der Agent als JAR-Datei

Die obigen drei Messages sind auf eine bestimmte Größe im System beschränkt. Damit ergibt sich das Diagramm für den Zustand des Agenten, der sich für nicht nur beim Hinsenden, sondern auch beim Zurücksenden (d.h. Die Zustände des Zurücksenden liegen im Zustand e des Diagramm 2.3) in der Migration befinden, wie folgt:

- **Messages-ID**: 1 (Es bedeutet die Status-Nachricht)
- **Implicit-Name**: Agentenbezeichner
- **Status**: Acknowledge/no Acknowledge

Nach obigem Zustand des Diagramms (siehe die Abbildung 2.3) gibt es drei Fälle:

(1): Alles lief korrekt ab: Der Sender hat die erfolgreich digital signierte Empfangsbestätigung von dem Empfänger bekommen. Danach wird der Agent im Sender gelöscht und die Verbindung zwischen Sender und Empfänger wird geschlossen.

(2): Der Empfänger lehnt den Agent ab und der Empfänger meldet diesen Fehler dem Sender (IOEexception). Daraus folgt, dass der Sender noch einmal den Migrationsprozess versucht, wie er anfangs vorgenommen wurde.

(3): Der Sender bekommt keinen Nachweis (Erfolgsnachricht oder Misserfolgsnachricht) von dem Empfänger. Nach einer bestimmten Zeit schickt der Sender eine Nachricht zu ATLAS, um anzufragen, ob dies geschickten Agent noch auf anderen Rechnern existiert. Es gibt zwei der folgenden Fälle für die Beantwortung durch ATLAS:

- **Nein**: d.h. der Agent existiert nicht auf anderen Rechnern. Daraus folgt, dass der Sender noch einmal versucht, diesen Agent zum Zielsystem zu schicken.
- **Ja**: Im diesem Fall entstehen zwei Möglichkeiten:
    1. **Der Agent wurde erfolgreich zum Zielsystem übertragen (d.h. die Agentenklassen werden im Empfänger ausgeführt), aber der Sender bekommt nicht die digital signierte Empfangsbestätigung. Damit löscht der Sender diesen Agent in seinem Rechner und schließt die Verbindung zwischen den zwei Rechnern.**
    2. **Obwohl ATLAS antwortet, dass der Agent auf anderen Rechner existiert, wird der Agent eigentlich vom Empfänger abgelehnt, weil ATLAS nicht der lezte Filter der Sicherheitsfilterpipeline des Empfängers ist. Deshalb wird der Agent von ATLAS angenommen, aber vom fünften oder sechstem Filter der Sicherheitsfilterpipeline des Empfängers abgelehnt. Um dieses Problem lösen zu können, ist die bestimmte Wartezeit des Senders größer als die Zeitspanne des Migrationsprozesses der Agenten im Empfänger.**

Das Konzept ist bereits ziemlich ausgereift. Trotzdem gibt es noch ein paar Aspekte, die bei der Migration der Agenten noch verbessert werden können (siehe Kapitel 3).

2.3 Implementierung

2.3.1 Klassen

In diesem Abschnitt werden die Klassendiagramme vorgestellt, welche die bisherige Implementierung gewinnbringend erweitern. Bezüglich des Konzepts vom Abschnitt 2.2.3 werden die Klassen im Hinblick auf die Transaktionssicherheit von mobilen Agenten in SeMoA ausgebaut.

Wie bereits erläutet wurde, muss - damit der Agent zum Bestimmungsort im Netzwerk migrieren kann - zuerst die Adresse vom Zielort ermittelt werden. Die Methode `setTicket(URL url)` der Klasse `Mobility` bestimmt die URL (die Adresse vom Zielsystem) für den Agent. Danach ruft die Funktion `sigterm(Bootstrapper bs, ErrorCode err)` der Klasse `AgentContext` die Methode `run(AgentContext ctx)` auf, um mit der Vorbereitung für die Migration zu beginnen. Von der Methode `run(AgentContext ctx)` wiederum wird die Methode `dispatchAgent(AgentContext ctx)` der Klasse `OutGate` aufgerufen. Diese bewirkt, dass die Agenten die `Filter`, die in dem `Environment` vom `OutGate` existieren, durchlaufen werden.

Wenn alles ordnungsgemäß ablief, wird weiterhin die Methode `dispatchAgent(AgentContext ctx, Resource struct, Ticket ticket)` der Klasse `RawOutGate` von der Methode `dispatchAge- nt(AgentContext ctx)` der Klasse `OutGate` aufgerufen, um sich mit dem Zielrechner über eine `Socket` zu verbinden. Anschließend werden die Agenten zum Bestimmungsort verschickt. Wenn die Agenten verschickt werden, werden nicht nur ihre URL und Ressourcen verschickt, sondern gleichzeitig auch ihre Message-ID. Es werden auch auftretende Fehler in den Methoden abgefangen. Wenn Fehler auftreten, kann der Sender sie erkennen und der Prozess der Migrationen kann durch eine Wiederholung erneut versucht werden.

Nachdem die Agenten durch die Methode `dispatchAgent(AgentContext ctx, Resource struct, Ticket ticket)` verschickt wurden, wird die Verbindung zwischen zwei Systemen noch gehalten, damit der Sender auf die Bestätigung vom Empfänger durch die Methode `waitforSignal(String str)` der Klasse `Signals` warten kann. Nach dem Senden kommen die Agenten zuerst am `RawInGate` (Gateway) des Empfängers an.

In der Klasse `RawInGate` gibt es eine Funktion `creatJob(Socket connection)`, die implementiert wurde, um sich mit dem `RawOutGate` des Senders durch eine `Socket` zu verbinden. Diese Funktion ruft eine Klasse `Job` mit der `run()`-Methode auf, um die Message-ID, die URL und den Agenten, die vom `RawOutGate` verschickt wurden, auch vollständig zu erhalten. Die `run()`-Methode ruft die Methode `launchAgent(Resource res, Ticket ticket)` der Klasse `InGate`, um die `Resource` des Agenten über die `Filter`, die in dem `Environment` von `InGate` existieren, zu prüfen. Die Klasse `Ticket` mit der `URL` als Parameter gibt die Adresse des Senders zurück. Bezüglich dieses `Tickets` werden die Informationen vom Empfänger zum Sender mit dieser Methode zurückgeschickt. Deswegen wird die `Ticket` von `ship` nach `raw` (oder `raws`, die an das Gateway zwischen `RawOutGate` und `RawInGate` passen) über die neue Klasse `ChangeTicket` (siehe die Abbildung 2.4) umgewandelt. Beispiel: von `ship://198.168.2.1` nach `raw://198.168.2.1`. (siehe Code der Klasse `ChangeTicket` in Anhang A.1)

Wenn alles nach Plan ablief, d.h. nach dem `ctx.start()` in der Methode `launchAgent(AgentContext ctx)` der Klasse `InGate` aufgerufen wurde, schickt der Empfänger eine Empfangsbestätigung zum Sender über das instanziierte `Objekt (new SendBack (Ticket ticket, String inf, AgentContext ctx)` der Klasse `SendBack` (siehe die Abbildung 2.5) zurück.

`SendBack` ist eine neue Klasse, die implementiert wurde, um Informationen (vom Typ `String`) zum `RawInGate` des Senders über das `Ticket`, das schon gewechselt wurde, zurückzuschicken. (siehe Code der Klasse `SendBack` in Anhang A.2)

Diese Informationen bestehen nicht nur aus der Empfangsbestätigung, nachdem die Agenten erfolgreich im System des Empfängers gestartet wurde, sondern auch aus Informationen über Fehler, wegen denen Agenten im System des Empfängers abgelehnt wurden. Nachdem das `RawInGate` des Senders diese Informationen vom Empfänger bekommt, soll es ein Signal zum `RawOutGate` des Senders über die Methode `sendSignal(String name, int i)` der Klasse `Signals` weiter schicken, um zu bestätigen, dass die Migration der Agenten erfolgte oder fehlgeschlagen ist. Wenn die Migration erfolgte, wird die Verbindung zwischen den zwei Systemen geschlossen und der Agent im Sender wird gelöscht. Aus obigem Konzept ergeben sich zwei Möglichkeiten: Informationen über den Erfolg oder das Fehlschlagen der Migration, die zurück zum Sender geschickt werden - was noch nicht ausreicht. Deshalb gibt es noch dritte Möglichkeit: Eine `TimeoutMeldung` des Programms. Deswegen wurde die Klasse `Timer` (siehe Abbildung 2.6) neu implementiert, um das Konzept zu vervollständigen.

Der Sender kann auf die Antwort vom Empfänger nur eine bestimmte Zeitspanne (z.B: 10 Sekunden, weil die Zeitspanne für eine erfolgreiche oder erfolglose Migration des Hin-, und Zurücksenden der Informationen kleiner als 10 Sekunden ist.) warten. Wenn der Sender nach dieser Zeitspanne keine Nachricht vom Empfänger bekommt, fragt der Sender bei ATLAS nach dem Zustand dieser Agenten im Netzwerk. Danach meldet ATLAS dem Sender die folgenden zwei

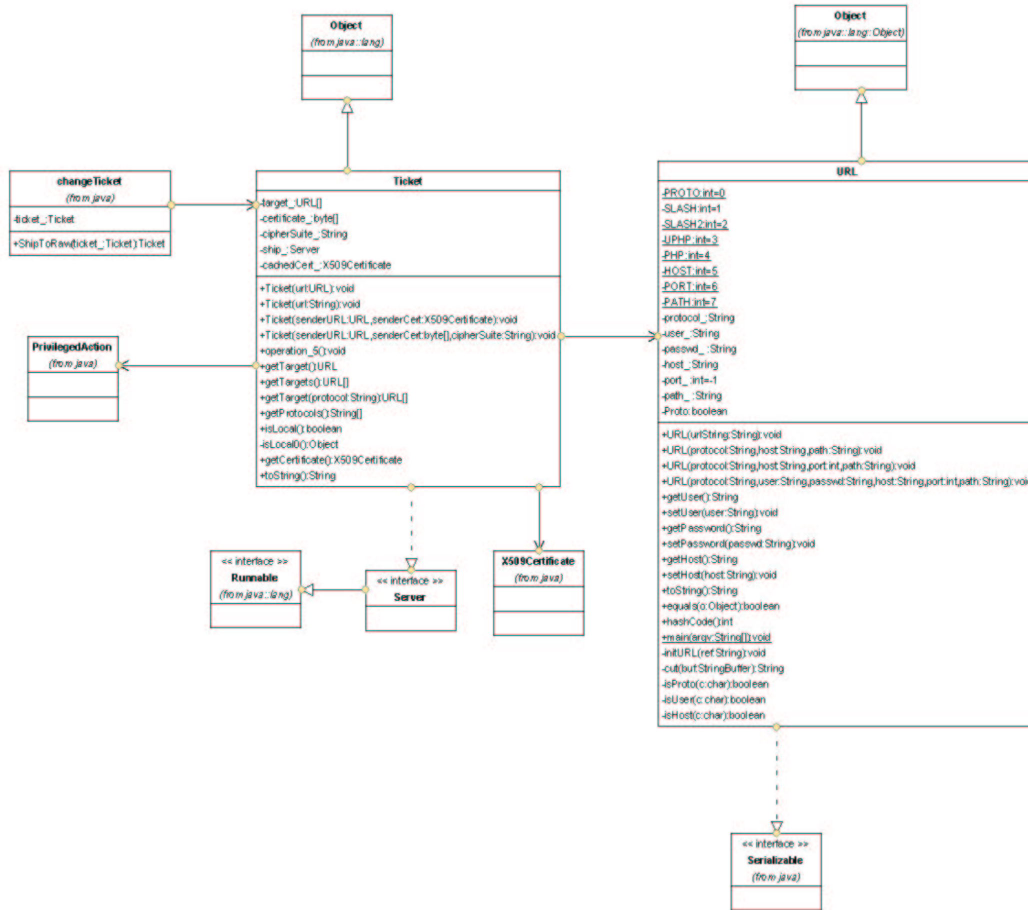

Abbildung 2.4: changeTicket

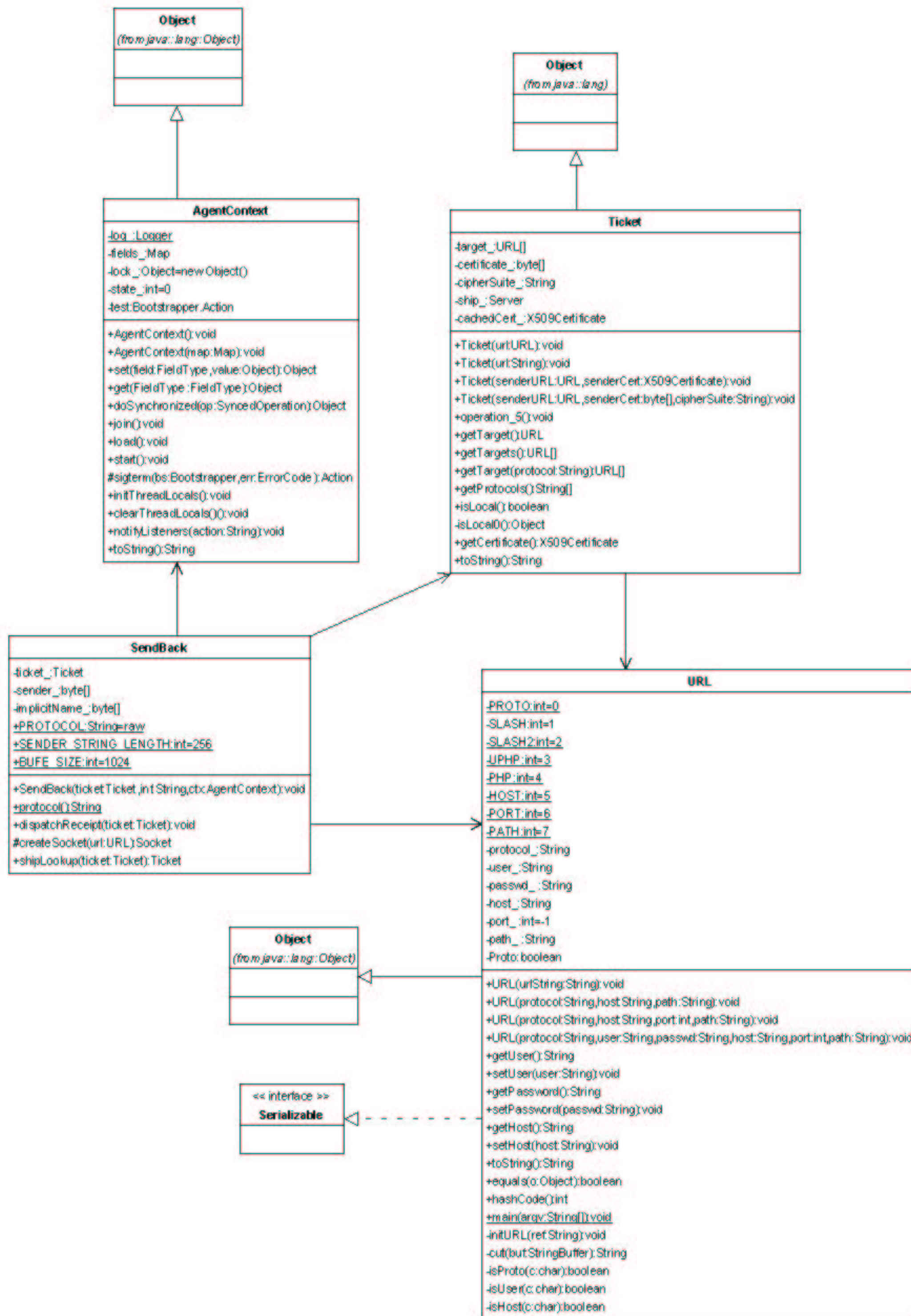

Abbildung 2.5: SendBack

Möglichkeiten zurück:

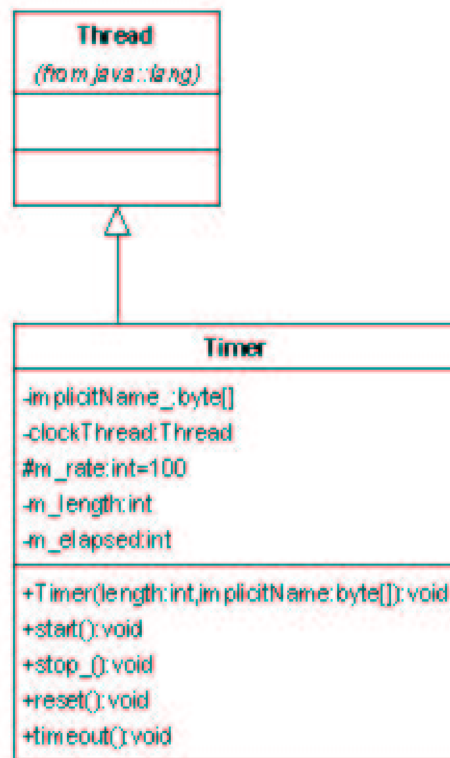

Abbildung 2.6: timeout

1. **Die Migration ist erfolgreich: d.h. der verschickten Agent wird im Zielort ausgeführt. Wenn dieser Fall auftritt, schließt das Programm die Verbindung zwischen den zwei Rechnern und dies Agent wird auch im Sender gelöscht.**

2. **Die Migration ist erfolglos: d.h. der Agent existiert nicht auf anderen Rechnern im Netzwerk außer auf dem Rechner des Senders. Wenn dieser Fall auftritt, kann der Sender den Prozess der Migration mit dem existierenden Agent, der nicht erfolgreich verschickt wurde, noch einmal versuchen.**

Diese neue Klasse wird in der Klasse `RawOutGate` aufgerufen, um diesen Zweck zu erfüllen. (siehe die Code der Klasse `Timer` in Anhang A.3)

2.3.2 Integration in SeMoA

Im letzten Abschnitt über die Klassenstruktur wurde die Integration der neuen Klassen in SeMoA schon teilweise dargestellt. Wie man gesehen hat, wurde die Transaktionssicherheit für die Migration von mobilen Agenten innerhalb der sieben Klassen `AgentGateway, LSClient, OutGate, RawOutGate, RawInGate, Ingate und Signals` erweitert. Außerdem wurden die drei neuen Klassen `SendBack, ChangeTicket und Timer` implementiert.

Die Agenten werden durch die Methode `dispatchAgent(AgentContext ctx, Resource struct, Ticket ticket)` aufgerufen (siehe Code der Klasse `RawOutGate` in Anhang A.4). In dieser Methode wird nicht nur die URL des Zielsystems und die Ressourcen des Agenten, sondern auch die Message-ID verschickt. Die Message-ID wird verschickt, um das Hinschicken oder Zurückschicken ins `RawInGate` zu vermerken. Für jeden Agenten wird eine ID gesetzt und verschickt, weil der Sender auf ein Signal mit der Agent-ID warten muss, um den Prozess jedes Agenten beenden zu können. Nachdem die Agenten verschickt wurden, wird die Methode `waitforSignal(String str)` der Klasse `Signals` aufgerufen, um auf die Empfangsbestätigung von dem Empfänger zu warten.

Die Agenten werden zum Empfänger durch die Methode `run()` der Klasse `Job` in der Klasse `RawInGate` geschickt (siehe Code der Klasse `RawInGate` in Anhang A.5). In dieser Methode werden zwei Fälle unterschieden:

- **Wenn die Message-ID = 0 ist, kommen die Agenten beim Empfänger an**

- **Wenn die Message-ID = 1 ist, kommt die Information vom Empfänger zum Sender zurück**

Weiterhin wird in der `run()`-Methode der Klasse `RawInGate` die Methode `launchAgent-(Ressource res, Ticket ticket)` der Klasse `OutGate` aufgerufen. Zuerst laufen die Agenten durch aufeinanderfolgende `Filter`, die im `Environment` vom `InGate` existieren. Wenn die Agenten durch irgendeinen `Filter` im `Environment` abgelehnt werden, wird diese Informationen zum Sender über das instanziierte Objekt der Klasse `SendBack` zurückgeschickt und die Migration kann durch einen Neustart noch einmal versucht werden. Wenn alles korrekt ablief, d.h. `ctx.start()` wird in der Methode `launchAgent(AgentContext ctx)` der Klasse `InGate` aufgerufen (siehe Code der Klasse `InGate` in Anhang A.8), wird eine Empfangsbestätigung vom Empfänger zum Sender zurückgeschickt, um den Erfolg für die Migration durch die Methode `sendSignal(String str, int i)` der Klasse `Signals`

zu bestätigen (siehe Code der Klasse `Signals` in Anhang A.6). Danach wird die `Socket`-Verbindung zwischen den zwei Systemen geschlossen und die Agenten werden im Sender gelöscht.

Außerdem kann noch ein anderer negativer Fall auftreten. Der Empfänger meldet sich nicht beim Sender bzw. das Zielsystem stürzt ab. Dadurch bekommt der Sender keine Information vom Empfänger mehr. In diesem Fall setzt der Sender eine bestimmte Zeit (z.B. 10 Sekunden), um auf die Information vom Empfänger erneut zu warten. Falls es nach dieser Zeitspanne keine Information vom Empfänger gibt, hat das Programm folgende zwei Möglichkeiten durch die Informationen des von ATLAS: Die Agenten wurden erfolgreich im Empfänger ausgeführt: die Agenten werden im Sender gelöscht und die Verbindung geschlossen; oder die Agenten existieren noch im Sender und der Sender kann den Prozess der Migration erneut versuchen. Die Klasse `Timer` wird in der Methode `dispatchAgent(Resource res, Ticket ticket, AgentContext ctx)` in der Klasse `RawOutGate` in diesem Fall erneut aufgerufen (siehe Code der Klasse `RawOutGate` in Anhang A.4)

Um obigen Prozess zu realisieren, wird ein Pfad `(RAWOUTGATE=/transport/outgate-/raw)` für das `RawOutGate` im `Environment` definiert, um die Klasse `RawOutGate` sowie Dienste für die verschiedenen Agenten benutzen zu können. Dieses Verzeichnis wird in der `whatis.conf`-Datei von **SeMoA** konfiguriert.

2.4 Evaluation

2.4.1 Testumgebung / Testszenario

Nachdem die Erweiterung fertiggestellt wurde, wurde eine Testumgebung für die neue Migration verwendet, um sie mit der alten Migration zu vergleichen. In diesem Abschnitt wird die daraus gewonnene Zeitmessungsstatistik im Detail vorgestellt werden.

Es wurde schon an anderer Stelle erwähnt, dass, nachdem die Adresse des Zielsystems (URL) und die Agenten in der "alten" erfolreich Migration verschickt wurden, die Verbindung zwischen den zwei Systemen geschlossen und die Agenten im Sender gelöscht werden. Deshalb wird die Zeit für diese Migration nur von dem Sender aus gesehen gemessen. Zuerst wird die Zeit für das Starten der Migration in der Klasse `AgentContext` (T1) gemessen. Danach ruft die Klasse `AgentContext` die Klasse `SyncedMigrate` auf, um alles für die Migration vorzubereiten. Durch die Klasse `SyncedMigrate` durchlaufen die Agenten die `Filter` in der Klasse `OutGate`. Die Zeit wird für jeden Filter, den die Agenten durchlaufen, gemessen. Im nächsten Schritt wird die Klasse `OutGate` von der Klasse `RawOutGate` aufgerufen, um die fertigen Agenten zu verschicken. Die Zeit (T5) wird auch für das Verschicken gemessen. Anschließend wird die Zeit (T6) für den Erfolg des Verschickens vom Sender, d.h. nach dem die Verbindung zwischen zwei Rechnern geschlossen wurde und die Agenten im Sender gelöscht wurden, gemessen (siehe Tabelle 2.1).

In der neuen Implementierung wird die Zeit für die erweiterte Migration in folgenden drei Fällen gemessen:

- **1. Fall**: Die Migration ist erfolgreich, d.h. der Sender bekommt eine Empfangsbestätigung vom Empfänger. Danach wird die Verbindung zwischen den zwei Systemen geschlossen

Zeitspunkt	Phase	Zeitspunkt[ms]
T1	**Startet Migration**	0 ms
T2	**Durch 1.Filter (in OutGate)**	104 ms
T3	**Durch 2.Filter (in OutGate)**	105 ms
T4	**Durch 3.Filter (in OutGate)**	105 ms
T5	**Zum Verschicken der Agenten und URL**	190 ms
T6	**Erfolg Verschicken vom Sender aus**	350 ms

Tabelle 2.1: Test für die alte Migration

Zeitspunkt	Phase	Zeitspunkt[ms]
T1	**Startet Migration**	0 ms
T2	**Durch 1.Filter (in OutGate)**	100 ms
T3	**Durch 2.Filter (in OutGate)**	110 ms
T4	**Durch 3.Filter (in OutGate)**	110 ms
T5	**Zum Verschicken der Agenten und URL**	190 ms
T6	**Erfolg Verschicken vom Sender aus**	2494 ms

Tabelle 2.2: Test für die neue erfolgreiche Migration

und die Agenten werden gelöscht.

- **2. Fall**: Die Agenten werden vom Empfänger abgelehnt. Danach bekommt der Sender eine Meldung vom Empfänger und der Sender kann die Migration noch einmal versuchen. Dieser Fall ist also durch anfänglichen Misserfolg gekennzeichnet.

- **3. Fall**: Der Sender bekommt keine Informationen vom Empfänger. Dieser Fall ist durch ein "Timeout" gekennzeichnet.

Bis zu T5 weichen die Ergebnisse in keiner Weise von der ursprünglichen Implementierung ab.

In dem 1. Fall wird die Zeit (T6) für die erfolgreiche Migration gemessen (siehe Tabelle 2.2).

In dem 2. Fall wird die Zeit (T6) für die erfolglose Migration gemessen (siehe Tabelle 2.3).

Zeitspunkt	Phase	Zeitspunkt[ms]
T1	**Startet Migration**	0 ms
T2	**Durch 1.Filter (in OutGate)**	100 ms
T3	**Durch 2.Filter (in OutGate)**	110 ms
T4	**Durch 3.Filter (in OutGate)**	110 ms
T5	**Zum Verschicken der Agenten und URL**	190 ms
T6	**Erfolg Verschicken vom Sender aus**	2393 ms

Tabelle 2.3: Test für die neue erfolglose Migration

Zeitspunkt	Phase	Zeitspunkt[ms]
T1	**Startet Migration**	0 ms
T2	**Durch 1.Filter (in OutGate)**	120 ms
T3	**Durch 2.Filter (in OutGate)**	130 ms
T4	**Durch 3.Filter (in OutGate)**	130 ms
T5	**Zum Verschicken der Agenten und URL**	190 ms
T6	**Erfolg Verschicken vom Sender aus**	10455 ms

Tabelle 2.4: Test für die neue Migration bei gleichzeitigem Erfolg und Timeout

Zeitspunkt	Phase	Zeitspunkt[ms]
T1	**Startet Migration**	0 ms
T2	**Durch 1.Filter (in OutGate)**	120 ms
T3	**Durch 2.Filter (in OutGate)**	130 ms
T4	**Durch 3.Filter (in OutGate)**	130 ms
T5	**Zum Verschicken der Agenten und URL**	190 ms
T6	**Erfolg Verschicken vom Sender aus**	10365 ms

Tabelle 2.5: Test für die neue Migration bei Erfolglosigkeit und Timeout

In dem 3. Fall wird die Zeit (T6) für ein “Timeout” der Migration gemessen (siehe Tabelle 2.4 und siehe Tabelle 2.5).

2.4.2 Ergebnisse und Bewertung

Mit diesem obigen Abschnitt (Testumgebung) wird die Zeitmessung für beide, alte und neue Migration, vorgestellt. Es kommt darauf an, dass die Vorteile und Nachteile der beiden Migrationen in diesem Abschnitt dargestellt werden.

Beim Testen beträgt die Zeitspanne vom Starten der alten Migration zum Abschluss der alten Migration ungefähr 350 ms. Die neue Migration wird fünfmal mit Erfolg, Erfolglosigkeit und Timeout getestet. Die Zeitspanne ist am kleinsten mit 786 ms für die erfolgreiche Migration nach fünf Tests und am größten mit 2494 ms. Die Zeitspanne ist am kleinsten mit 892 ms für die erfolglose Migration nach fünf Tests und am größten mit 2393 ms. Die Zeitspanne ist am kleinsten mit 10184 ms für ein “Timeout” und erfolgreicher Migration nach fünf Tests und am größten mit 10455 ms. Die Zeitspanne ist am kleinsten mit 10184 ms für ein “Timeout” und erfolglose Migration nach fünf Tests und am größten mit 10365 ms. Deswegen ist der Zeitfaktor bei der alten Migration zum Vergleichen mit der neuen ein Vorteil. Aber die Zeit dieser Migration wird nur vom Sender aus gesehen gemessen: D.h. wenn die Agenten nicht im Zielsystem vorhanden sind, aber der Sender meldet, dass die Migration erfolgreich war und diese Agenten danach nicht mehr im Netz des Systems existieren. Dies ist ein Nachteil der alten Migration: Diese Migration hat weniger Sicherheit für den Benutzer. In der neuen Migration ist die Zeitspanne größer als die Zeitspanne der alten Migration, weil die Zeitspanne für das Verschicken und Zurückschicken berechnet wird - aber der Zeitverlust ist mehr als akzeptabel. Die Sicherheit der Migration für die Benutzer wird erhöht: hier liegt der große Vorteil der neuen

Migration.

Kapitel 3

Resümee

3.1 Zusammenfassung und Ausblick

3.1.1 Zusammenfassung

Diese Arbeit wurde im Bereich Agententechnologie entwickelt, vobei mobile Agenten besonders von Interesse waren. Was die mobilen Agenten betrifft, ist der Anteil der Migration eine wichtige Entwicklung der Informatik. Migration ist der Transport der Agenten von einem Rechner zum anderen im Netzwerk, um Informationen schnell und bequem untereinander austauschen zu können. Hier setzt **SeMoA** an, ein Programm vom Fraunhofer Institut, das für diese Migration der mobilen Agenten erforscht und entwickelt wird. In dem ursprünglichen Programm werden die mobilen Agenten schon relativ zufriedenstellend migriert, d.h. nachdem die mobilen Agenten über die Filter im Sender geprüft und akzeptiert worden sind, werden sie zum Bestimmungsort verschickt und danach wird die Verbindung zwischen den zwei kommunizierenden Rechnern geschlossen und die Agenten werden im Sender gelöscht. Im Zielort werden diese Agenten dann über die Filter im Empfänger überprüft. Wenn alles validiert ist, werden die Agenten im Zielort ausgeführt. Deswegen sind die Sicherheitvorkehrungen der Migration in diesem Fall nicht ausreichend für die Benutzer. Es ist eine neue Idee nötig, um die Migration des Programms für die Benutzer zu verbessern. Diese Idee betrifft die Transaktionssicherheit für die Migration von mobilen Agenten: Wenn die Agenten am Zielort angekommen sind, soll der Rechner des Empfängers eine Empfangsbestätigung zum Sender zurückschicken, damit der Sender weiß, ob die Agenten erfolgreich oder erfolglos im Zielort migrieren. Oder, falls der Rechner des Empfängers abstürtzt und die Agenten ihn nicht erreichen, der Sender eine Information darüber erhält.

Diese Idee wurde in dieser Arbeit verwirklicht. Um sie zu ermöglichen, wurden zuerst die Anforderungen der Rechtsicherheit der Migration analysiert. Danach wurden die Protokolle für die Migration basierend auf datenbanktypischen Transaktionen entworfen. Im nächsten Schritt wurde die Implementierung für **SeMoA** realisiert. Abschließend folgte ein Test und ein Benchmark für die neue Implementierung, um das Ergebnis der neuen Migration zu bewerten.

3.1.2 Evaluation

Diese Arbeit betrifft ein relativ neues Gebiet in der Agententechnologie, welches im Moment in der Informatik erforscht wird. Sie ist eine wichtige Entwicklung, nicht nur für die Informations-

technologie sondern auch für das alltägliche Leben. Bezüglich dieser Entwicklung verwenden die Benutzer die Computer bequem und mit der Gewißheit der Sicherheit beim Transport der Daten im Netzwerk. Um diese Sicherheit gewährleisten zu können, müssen die Benutzer mehr Zeit für das Warten der Antwort des Rechners oder auf das Programm im Empfänger in Kauf nehmen. Aber dieser Nachteil ist akzeptabel, weil die Zeitintervalle des Wartens durch den Benutzer geregelt werden können. Meiner Meinung nach ist diese Entwicklung wirklich sehr interessant - nicht nur für meine Kenntnisse, sondern auch für die Verwendung dieser Technik für das Leben der Menschen, das dadurch nach und nach besser, bequemer und sicherer wird.

Die Entwicklung des Programms für die Migration in **SeMoA** ist ausreichend für die Benutzer, aber noch nicht wirklich optimal für die Migration. Wenn die Benutzer auf **SeMoA** zugreifen wollen, müssen sie sich zuerst registrieren, um Zugriffrechte auf die Dienste im Environment zu haben. Es kommt darauf an, dass das Programm für die Migration verbessert wird. Wenn **SeMoA** im Rechner gestartet wird, besitzt es die Identifikation jedes Benutzers. Bezüglich diesem Punkt wird die Adresse (URL) jedes Rechners mit der Identität des Benutzer identifiziert, d.h. wenn die URL des Rechners im Sender zum Zielort verschickt wird, kennt der Empfänger den Sender der Agenten über die URL des Senders. Die Idee ist hier, dass die URL des Sender zuerst zum einem bestimmten Ort verschickt wird und man dann abfragt, ob ein Zugriffrecht besteht. Wenn der Empfänger dies bejaht, verschickt der Sender die Agenten hierhin, wenn der Empfänger verneint, braucht der Sender nicht mehr die Agenten zu verschicken, sondern versucht sie zu anderen Rechnern zu schicken. Mit dieser Idee wird die Sicherheit erhöht, weil die Agenten nicht im ganzen System verschwinden. Auf der anderen Seite denke ich, dass die Wartezeit auch damit kürzerer für die Benutzer ist.

3.1.3 Ausblick

Natürlich sollte dieses Programm noch verbessert und erweitert werden, um ein optimale Benutzbarkeit für das alltägliche Leben zu erreichen. Auf Basis dieser Arbeit können weitere Sicherheitsmechanimen für die Migration von miblen Agenten entwickelt werden:

- **Signierte Empfangsbestätigung**
- **Eindeutige Identifikation des Zielserver**
- **Migration über Proxy-Server (als Truted Third Party)**

Anhang A

Programmcode

A.1 Programmcode von `ChangeTicket`

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package examples;

/**
 * @author tthi
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
import DE.FhG.IGD.semoa.server.Ticket;
import DE.FhG.IGD.util.URL;

public class ChangeTicket
{
    private Ticket ticket_;

    public ChangeTicket()
    {

    }

    public Ticket ShipToRaw(Ticket ticket)
    {
```

```
        URL url;

        if(ticket != null)
        {
            try
            {
                url = ticket.getTarget();

                url = new URL(
                        "raw",
                        url.getUser(),
                        url.getPassword(),
                        url.getHost(),
                        url.getPort(),
                        url.getPath());

                ticket_ = new Ticket(url);


                return ticket_;
            }
            catch(Exception e)
            {
                System.err.println("Error " + e);
            }

        }
        return null;
    }
}
```

A.2 Programmcode von `SendBack`

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */

package DE.FhG.IGD.semoa.net;

import java.io.*;
```

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.Socket;
import java.util.Arrays;

import DE.FhG.IGD.semoa.net.ship.ShipException;
import DE.FhG.IGD.semoa.net.ship.ShipService;
import DE.FhG.IGD.semoa.server.AgentCard;
import DE.FhG.IGD.semoa.server.Environment;
import DE.FhG.IGD.semoa.server.AgentContext;
import DE.FhG.IGD.semoa.server.FieldType;
import DE.FhG.IGD.semoa.server.Ticket;
import DE.FhG.IGD.semoa.server.TicketNotSupportedException;
import DE.FhG.IGD.util.URL;
import DE.FhG.IGD.util.WhatIs;

/**
 * Send back receipt via simple TCP/IP socket connections.
 * @author tthi
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class SendBack
{
    /**
     * The private reference to the ticket.
     */
    private Ticket ticket_;

    /**
     * The buffer that contains a String representation of the URL of
     * this server’s management service. This is sent during a migration.
     */
    private byte[] sender_;

    private byte[] implicitName_;

    /**
     * The protocol name.
     */
    public static final String PROTOCOL = "raw";

    /**
     * The size of the buffer that is used for writing the
     * agents to the sockets.
     */
    public static final int BUF_SIZE = 1024;
```

```
/**
* The second <i>n</i> bytes of the data stream are expected to
* describe the senders management STRING. This constant defines
* the maximum length of STRING.
*/
protected static int SENDER_STRING_LENGTH = 256;

/**
 * Creates an instance.
 */
public SendBack(Ticket ticket, String inf, AgentContext ctx)
{
    byte[] implicitName;
    byte blank;
    byte [] b;
    int n;
    ticket_ = shipLookup(ticket);
    implicitName = ((AgentCard)ctx.get(FieldType.CARD)).getName();
    implicitName_ = implicitName;
    b = inf.getBytes();
    sender_ = new byte[SENDER_STRING_LENGTH];

    /* I don't know really what to do if the host URL is too
     * long. Maybe it's the best to leave the array to be sent
     * empty.
     */
    if(b.length > SENDER_STRING_LENGTH)
    {
        return;
    }

    /* We copy the url to the specified array and fill the
     * remaining space with blanks. They are not allowed in URLs
     * and easy to remove.
     */
    for(n=0; n<b.length; n++)
    {
        sender_[n] = b[n];
    }

    blank = (byte)' ';
    Arrays.fill(sender_, n, SENDER_STRING_LENGTH-1, blank);

    try
    {
        dispatchReceipt(ticket_);
    }
    catch(TicketNotSupportedException e)
```

```
        {
            /* Ignore */
            System.err.println(
                "[SendBack] Transport error: \""
                +e.getMessage()
                +"\"");
        }
        catch(IOException e)
        {
            /* Ignore for now, pass agent to the
             * queue handler later.
             */
            System.err.println(
                "[SendBack] I/O error: \""
                +e.getMessage()
                +"\"");
        }
    }

    /* ----- Methods from AgentGateway.Out ----- */
    public String protocol()
    {
        return PROTOCOL;
    }

    public void dispatchReceipt(Ticket ticket)
        throws TicketNotSupportedException, IOException
    {

        BufferedOutputStream out;
        Socket socket;
        URL[] targets;
        URL url;

        if (ticket == null )
        {
            throw new NullPointerException("Ticket or Resource!");
        }
        targets = ticket.getTarget(protocol());

        if (targets.length == 0)
        {
            throw new TicketNotSupportedException(
                "Bad ticket, no \""+protocol()+"\" target specified!");
        }

        socket   = null;
        out      = null;
```

```
        try
        {
            socket = createSocket(targets[0]);
            out    = new BufferedOutputStream(
                        socket.getOutputStream(), BUF_SIZE);
            out.write(1);
            out.write(implicitName_,0,implicitName_.length);
            out.write(sender_,0,SENDER_STRING_LENGTH);

            /* It is necessary to check, if the socket has already
             * been closed by the recipient server's ingate.
             */
            if (!socket.isClosed())
            {
                out.flush();
            }
        }
        catch(Exception e)
        {
            System.err.println("[SendBack] An exception occured: "+e);
        }
        finally
        {
            try{
                if (socket!=null)
                {
                    socket.close();
                }
            }
            catch(Exception e)
            {
                System.err.println("[SendBack] An exception occured: "+e);
            }
        }
    }

    protected Socket createSocket(URL url)throws IOException
    {
        return new Socket(url.getHost(), url.getPort());
    }

    /**
     * Returns the correct transport URL after a lookup at the remote
     * SHIP service. We interpret the fully qualified URLs enclosed in
     * the ticket as URL to a SHIP port. This is because actually
     * vicinity post its port within the LAN. Hence, we have to do a
     * SHIP lookup in order to get the URL of the desired transport
```

```
     * protocol.<p>
     *
     * EXAMPLE: Given that we have a ticket containing the URL
     * <tt>raw://somehost:47470</tt> and the port of RawInGate at
     * 'somehost' is 50000. Now, the port 47470 is understand to be
     * the port of the remote SHIP service. This method invokes a SHIP
     * request and - hopefully - transforms the ticket URL into
     * <tt>raw://somehost:50000</tt>.<p>
     *
     * This method does this resolving for all URLs contained in the
     * given ticket. If any error occurs during the ship request(s)
     * the URL will remain unchanged. This is the same if the local
     * ship service is not running. So occasionally the returned
     * ticket may be the given <code>ticket</code>.<p>
     *
     * Note: this URL replacement is a fine mechanism to do load
     * balancing on a cluster of agent servers. Simply configure the
     * cluster ingate server in a way that it tells different
     * transport URLs on every request.
     *
     * @param ticket The agent ticket to be transformed.
     * @return A ticket with the transformed URLs.
     * @exception NullPointerException iff <code>ticket</code> is
     *     <code>null</code>.
     */
    protected Ticket shipLookup(Ticket ticket)
    {
        ShipService ship;
        Environment env;
        String proto;
        String path;
        String key;
        String val;
        URL[] urls;
        URL url;

        if(ticket == null)
        {
            throw new NullPointerException("Parameter ticket may not
                                                       be null");
        }

        env  = Environment.getEnvironment();
        path = WhatIs.stringValue("SHIP");
        try{
            ship = (ShipService)env.lookup(path);

            if(ship == null)
```

```
        {
            System.err.println("[OutGate] No ship service found at '"
                                   + path + "'.");
            return ticket;
        }
    }
    catch(ClassCastException ex)
    {
        System.err.println("[OutGate] Service at '"
                              + path
                              +"' is not a ship service.");
        return ticket;
    }

    urls = ticket.getTargets();
    for(int i=0;i<urls.length; i++)
    {
        /* We use a well known key here which is build as:
         * transport.in.<protocol>.url
         */
        try{
            proto = urls[i].getProtocol();
            key = "transport.in." + proto + ".url";
            val = ship.getValue(key,urls[i]);

            /* If we didn't get a proper answer, we assume the url
             * to be already the correct protocol url.
             */
            if(val == null || val.equals(""))
            {
                continue;
            }

            url = new URL(val);

            /* Make sure to stick with the protocol. Of course
             * this check should also be performed with the host
             * name. But since we cannot be sure that an IP
             * address equals a fully qualified host name and vice
             * versa we skip the latter test.
             */
            if(! proto.equals(url.getProtocol()))
            {
                System.err.println("[OutGate] Warning: SHIP
                      retrieved " +"an URL with a different protocol
                      than " +"the URL in the original ticket "
                             +"specifies. Conversion form  "
                             + url + " to " + urls[i] + " denied.");
```

```
                    continue;
                }

                /* The common way is to replace the current URL with
                 * the received value.
                 */
                urls[i] = url;
            }
            catch(ShipException ex)
            {
                continue;
            }
            catch(MalformedURLException ex)
            {
                continue;
            }
        }

        return new Ticket(urls);
    }

}
```

A.3 Programmcode von `Timer`

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package examples;

/**
 * @author tthi
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

import DE.FhG.IGD.semoa.server.Environment;

import DE.FhG.IGD.semoa.net.*;
```

```
import DE.FhG.IGD.util.WhatIs;
import DE.FhG.IGD.util.URL;

import DE.FhG.IGD.atlas.core.*;
import DE.FhG.IGD.atlas.lsp.*;

import java.lang.Thread;


public class Timer extends Thread
{
    /**
     * The buffer that contains a String representation of the name
     * of agent of this server's management service. This is sent
     *during a migration.
     */
    private byte[] implicitName_;

    /**
     * Creates a thread, in its start method
     */
    private Thread clockThread;

    /**
     *  Rate at which timer is checked
     */
    protected int m_rate = 100;

    /**
     * Length of timeout
     */
    private int m_length;

    /**
     * Time elapsed
     */
    private int m_elapsed;

    /**
     * Creates a timer of a specified length
     * @param length Length of time before timeout occurs
     */
    public Timer ( int length, byte[] implicitName )
    {

       implicitName_ = implicitName;
  m_length = length;
```

```
        m_elapsed = 0;

     }

     /**
      * Starting a Thread
      */
     public void start()
     {
        if (clockThread == null)
      {
   clockThread = new Thread(this, "Clock");
         clockThread.start();
  }
     }

     /**
      *Stopping a Thread
      */
     public void stop_()
     {
clockThread = null;
     }

     /**
      * Resets the timer back to zero
      */
     public synchronized void reset()
     {
m_elapsed = 0;
     }

     /**
      * Performs timer specific code
      */
     public void run()
     {
Thread thisThread;

thisThread= Thread.currentThread();

/**
   * Keep looping
         */
while (clockThread == thisThread)
{
/**
 * Put the timer to sleep
```

```
   */
try
{
Thread.sleep(m_rate);
  }
catch (InterruptedException ioe)
{
        continue;
}

/**
 * Use 'synchronized' to prevent conflicts
                 */
synchronized ( this )
{

  /**
   * Increment time remaining
                   */
m_elapsed += m_rate;

/**
   * Check to see if the time has been exceeded
                   */
if (m_elapsed > m_length)
{

/*
   * Trigger a timeout
   */
        timeout();
}

}

}
    }

    /**
     * Override this to provide custom functionality
     */
    public void timeout()
    {
LSClientService clientService;
RawOutGate signal;
LSPReply reply;
Server ship;
URL url;
```

```
System.err.println ("Network timeout occurred....continue");

ship = (Server)Environment.getEnvironment().lookup(
            WhatIs.stringValue("SHIP"));
clientService = (LSClientService)Environment.getEnvironment().lookup(
        WhatIs.stringValue("ATLAS")+"/client/service");

   signal = (RawOutGate)Environment.getEnvironment().lookup(
                WhatIs.stringValue("RAWOUTGATE"));

/**
         * We don’t use clientService.lookup() here, because this
 * implies a localLookup() call, which returns the local
 * Ship-URL in every case, since the agent is still
 * registered in the local Environment.
 * Rather we make use of the methods proxyLookup() and
 * globalLookup of LSClientServiceImpl.
 */
reply = ((LSClientServiceImpl)clientService).proxyLookup(
implicitName_);

if(reply.getState() != LSPReply.ACKNOWLEDGE)
        {
   reply = ((LSClientServiceImpl)clientService).globalLookup(
            implicitName_);
        }

if(reply.getState() == LSPReply.ACKNOWLEDGE)
        {
           url = ((LSPLookupResult)reply.getBody()).getContactAddress();
           System.out.println("URL: "+url);
           if(!url.equals(ship.localURL()))
           {
               System.out.println("Agent migrates successful!");
               signal.notifyAgent(new String(implicitName_),0);
               stop_();
           }
           else
           {
               System.err.println("Agent don’t migrate yet.");
               signal.notifyAgent(new String(implicitName_),2);
               stop_();
           }
        }
        else
        {
            System.err.println("Agent don’t exist on Server.");
```

```
            signal.notifyAgent(new String(implicitName_),2);
            stop_();

        }

    }
}
```

A.4 eingefügte Programmcode von `RawOutGate`

```
public void dispatchAgent(AgentContext ctx, Resource struct, Ticket ticket)
        throws TicketNotSupportedException, IOException, IllegalAgentException
    {
        BufferedOutputStream out;
        long currentTime;
        ErrorCode err;
        Socket socket;
        String agentName;
        Timer timer;
        URL[] targets;
        boolean b;
        byte[] implicitName;


        if (ticket == null || struct == null)
        {
            throw new NullPointerException("Ticket or Resource!");
        }
        targets = ticket.getTarget(protocol());

        if (targets.length == 0)
        {
            throw new TicketNotSupportedException(
                "Bad ticket, no \""+protocol()+"\" target specified!");
        }

        implicitName = ((AgentCard)ctx.get(FieldType.CARD)).getName();
        agentName = new String(implicitName);
        log_.info("ImplicitName of Agent: "+agentName);
        System.out.println("ImplicitName_Send: "+agentName);
        socket   = null;
        timer    = null;
        out      = null;

        try{
```

```
            System.out.println ("Starting timer.");

    timer = new Timer(10000, implicitName);
    timer.start();

    currentTime = System.currentTimeMillis();
    log_.info("Starting time for send(T4): "
            +currentTime);
    System.out.println ("Starting time for send(T4): "
            +currentTime);
            socket = createSocket(targets[0]);

           /*
            *Reset timer - timeout can occur on connect
            */


           out    = new BufferedOutputStream(
               socket.getOutputStream(), BUF_SIZE);


           out.write(0);

           out.write(sender_,0,SENDER_URL_LENGTH);
           Resources.zip(struct, out);


           /*
            * Reset timer - timeout is likely to occur during the
            * wait of signal
            */
    timer.reset();
    signal.waitForSignal(agentName);

        /**
            * Every time signal receives a confirmation from the send,
            * there are 2nd cases to happen by message from sender:
            * Case 1: Everything is successful
            * Case 2: there is any error
            */
           if(signal.getMessage()==0)
           {
               currentTime = System.currentTimeMillis();
               System.out.println("Signal arrived successful" +
                " to RawOutGate(T8): " +currentTime);
           }
           else
           {
```

```
                if(signal.getMessage()==1){
                    currentTime = System.currentTimeMillis();
                    System.err.println(
                            "Agent is reject by the destination. "
                            +(int)currentTime);
                    timer.stop_();

                    throw new IllegalAgentException("Agent is reject");
                }
                else{
                    currentTime = System.currentTimeMillis();
                    System.err.println(
                            "Destinations don't answer back"
                            +(int)currentTime);
                    timer.stop_();

                    throw new IllegalAgentException(
                            "No Answer back for sender ");
                }

             }

            /* It is necessary to check, if the socket has already
             * been closed by the recipient server's ingate.
             */
            if (!socket.isClosed())
            {
                out.flush();
            }

            /*
      * Shutdown timer
      */
     timer.stop_();

        }
        catch(IllegalAgentException e)
        {
            log_.debug("[Destination]"
                    + "An migration occured: "+e);
            System.err.println("[Destination]"
                    + "An migration occured: "+e);
            throw e;
        }
        catch(Exception e)
        {
            System.err.println("[RawOutGate]"
                    + "An exception occured: "+e);
```

```
        }
        finally
        {
            try{
                if (socket!=null)
                {
                    socket.close();
                }
            }
            catch(Exception e)
            {
                System.err.println("[RawOutGate]"
                        + "An exception occured: "+e);
            }
        }
    }


    /**
     * This method assumes the function sendSignal of class signals
     * to the class RawOutGate can receives a confirmation from RawInGate
     * of sender
     *
     * @param name The imlicitName of every agent
     * @param i The message from function sendSignal
     */
    public void notifyAgent(String name, int i)
    {
       signal.sendSignal(name, i);
    }
```

A.5 eingefügte Programmcode von `RawInGate`

```
/**
 * Runs the <code>job</code>
 */

public void run()
{
            changeTicket chTicket;
            RawOutGate signal;
            InputStream in;
            SendBack send;
            Resource res;
            String inf;
```

```
String agentName;
Server ship;
Ticket ticket;
URL url;
byte[] implicitName;
int msg_id;
int inf_id;
byte[] buf;

try
{

    in = con_.getInputStream();
    msg_id = in.read();
    System.out.println("Receive msg_id: "+msg_id);

    if(msg_id == 0)
    {
        log_.info("Receive msg_id from sender: "+msg_id);
        res = ingate_.createTmpResource();
        buf = new byte[RawOutGate.SENDER_URL_LENGTH];
        try
        {
            in.read(buf, 0, RawOutGate.SENDER_URL_LENGTH);
            ticket = createTicket(buf);
            Resources.unzip(in, res, getMaximumSize());
            ingate_.launchAgent(res, ticket);
         }
         catch(Exception e)
         {
            cleanup(res, e);
         }
         finally
         {
             try{
                 con_.close();
             }
             catch(Exception ee)
             {
                System.err.println("Error "
                                   + ee.getMessage());
             }
         }
    }else{
        log_.info("Receive msg_id from receiver: "+msg_id);
        implicitName = new byte [20];
        buf = new byte[SendBack.SENDER_STRING_LENGTH];
```

```
                    signal = (RawOutGate)Environment.getEnvironment().
                          lookup(WhatIs.stringValue("RAWOUTGATE"));
                    try{
                        in.read(implicitName,0,implicitName.length);
                        in.read(buf,0,SendBack.SENDER_STRING_LENGTH);
                        agentName = new String(implicitName);
                        log_.info("implicitName_Receiver: "+agentName);
                        inf = new String(buf);
                        inf_id = inf.compareTo(
                                "[Information from Receiver]: "+
                                "Agent arrived. Things work out!");
                        if(inf_id == 196)
                        {
                            System.out.println(inf);
                            signal.notifyAgent(agentName,0);
                        }
                        else{
                            log_.debug("Agent is rejected from
                                    the acceptor. "+ inf);
                            System.err.println(inf);
                            signal.notifyAgent(agentName,1);
                        }
                    }catch(Exception e){
                        System.err.println("[SendBack] An exception
                                     occured: "+e);
                    }
                }

            }catch(Exception e)
            {
                System.err.println("[Error] An exception occured: "+e);
            }
}
```

A.6 eingefügte Programmcode von `Signals`

```
 /**
     * This method signals all threads waiting on the
     * signal with the given name. If a signal with
     * that name does not exists then nothing happens.
     *
     * @param name The name of the signal.
     */
    public void sendSignal(String name, int i)
```

```
    {
        Object signal;
        setMessage(i);

        synchronized(signals_)
        {
            signal = signals_.get(name);
        }
        if (signal != null)
        {
            synchronized(signal)
            {
                signal.notifyAll();
            }
        }
    }

/**
     * This method set message_id for every agent that
     * is sent back to sender know
     * what the agents happen in the acceptors
     *
     * @param i The id of message
     */
    public void setMessage(int i)
    {
        msg = i ;
    }

    /**
     * This method holds back id of message
     *
     * @return The message_id of agent.
     */
    public int getMessage()
    {
        return msg;
    }
```

A.7 eingefügte Programmcode von `OutGate`

```
for (j=0; j<protos.length; j++)
        {
            gate = (AgentGateway.Out)getEnvironment().lookup(
                key+"/"+protos[j]);
```

```
            if (gate != null)
            {
                try{
                    currentTime = System.currentTimeMillis();
                    System.out.println("at gateway for out(T3): "+
                            (int)currentTime);
                    gate.dispatchAgent(ctx, resource, ticket);

                    /* Below, we invoke a callback that can be used
                     * to do silly stuff to soothe the desire of
                     * sales droids for effects, e.g. play an audio
                     * file that signals transport of the agent.
                     */
                    onTransport(ctx);

                    return;
                }
                catch(TicketNotSupportedException e)
                {
                    /* Ignore */
                    System.err.println(
                        "[OutGate] Transport error: \""
                        +e.getMessage()
                        +"\"");
                }
                catch(IOException e)
                {
                    /* Ignore for now, pass agent to the
                     * queue handler later.
                     */
                    System.err.println(
                        "[OutGate] I/O error: \""
                        +e.getMessage()
                        +"\"");
                }

            }
        }
```

A.8 eingefügte Programmcode von `InGate`

```
 /**
     * Launches the agent with the given <code>Resource</code> and
     * origin <code>Ticket</code>.<p>
```

```
 *
 * The agent’s <code>Resource</code> is piped through the
 * security filters registered under the path defined by the
 * {@link WhatIs WHatIs} key &quot;SECURITY_FILTERS_IN&quot;.
 * All filters have to accept the agent for it to be run. If
 * so, the agent is started by calling
 * <code>launchAgent(AgentContext ctx</code> where <code>ctx
 * </code> is created by this method.<p>
 *
 * One of the filters has to generate a unique <code>AgentCard
 * </code> for the agent, or the agent will be rejected/dropped
 * after having passed the filter pipeline.
 *
 * @param struct The <code>Resource</code> that contains
 *   the agent’s structure and data.
 * @param ticket The <code>Ticket</code> that identifies
 *   the sender of the agent, or <code>null</code> if the
 *   agent was launched locally.
 * @exception IllegalAgentException if the agent is rejected
 *   by one of the security filters, or an agent with the
 *   same card is already known.
 */
public void launchAgent(Resource struct, Ticket ticket)
    throws IllegalAgentException
{
    AgentFilter.In filter;
    ChangeTicket chTicket;
    AgentContext ctx;
    SortedMap filters;
    ErrorCode error;
    SendBack send;
    Map.Entry entry;
    AgentCard card;
    Iterator i;
    //Ticket tickLocal;
    String name;
    long time;
    int n;
    int j;


    if (struct == null)
    {
        ctx = new AgentContext();
        ctx.set(FieldType.REVERSE_TICKET, ticket);
        chTicket = new ChangeTicket();
        ticket_ = chTicket.ShipToRaw(ticket);
        send = new SendBack(ticket_, "[NullPointerException]"
```

```
+ "from Receiver: "
                +"Need a resource!", ctx);
        throw new NullPointerException("Need a resource!");


    }
    ctx = new AgentContext();
    ctx.set(FieldType.RESOURCE, struct);
    ctx.set(FieldType.REVERSE_TICKET, ticket);
    ticketLocal = (Ticket)ctx.get(FieldType.REVERSE_TICKET);

    try{
        if(ticketLocal != null){
           chTicket = new ChangeTicket();
           ticket_ = chTicket.ShipToRaw(ticket);

        }
        ctx.load();
    }
    catch(IOException e)
    {
        if(ticketLocal != null){
            send = new SendBack(ticket_,"[IllegalAgentException]" +
             "from Receiver: "
                   +"Error loading agent: "
                   +e.getClass().getName()
                   +"(\""
                   +e.getMessage()
                   +"\")", ctx);
        }
        throw new IllegalAgentException(
            "Error loading agent: "
            +e.getClass().getName()
            +"(\""
            +e.getMessage()
            +"\")"
        );
    }
    /* We now check if the agent type is in principle
     * supported. At least, judging from the properties
     * of the agent structure.
     */
    try{
        if (!Lifecycles.isSupported(ctx))
        {
            if(ticketLocal!=null)
            {
                send = new SendBack(ticket_,"[IllegalAgentException]" +
```

```
                    "from Receiver: "
                           +"agent type not supported", ctx);
                }
                throw new IllegalAgentException("agent type not supported");
            }
        }
        catch(NullPointerException e)
        {
            if(ticketLocal!=null)
            {
                send = new SendBack(ticket_,"[IllegalAgentException]" +
                 "from Receiver: "
                       +"agent type not defined", ctx);
            }
            throw new IllegalAgentException("agent type not defined");
        }
        filters = getSecurityFilters();
        time    = System.currentTimeMillis();
        j=0;

        for (i=filters.entrySet().iterator(); i.hasNext();)
        {
            entry  = (Map.Entry)i.next();
            name   = entry.getKey().toString();
            filter = (AgentFilter.In)entry.getValue();


            if (debug_)
            {
                if(ticketLocal!=null)
                {
                    send = new SendBack(ticket_,"[Debug_]from Receiver"
                           +"[InGate] Filtering through: "+name, ctx);
                }
                System.out.println(
                    "[InGate] Filtering through: "+name);
            }
            try{
                error = filter.filter(ctx);

                if (error != ErrorCode.OK)
                {
                    if(ticketLocal!=null)
                    {
                        send = new SendBack(ticket_,"[IllegalAgentException]"
+"from Receiver: "
                               +"Agent rejected by filter \""
                               +name
```

```
                            +"\", return code ("
                            +error
                            +")", ctx);
                    }
                    throw new IllegalAgentException(
                        "Agent rejected by filter \""
                        +name
                        +"\", return code ("
                        +error
                        +")"
                    );
                }
                j++;
            }
            catch(IllegalStateException e)
            {
                /* How unfortunate. Some filter was retracted
                 * right as we want to use it. In principle,
                 * this cannot happen because security filters
                 * should not be published using a proxy but
                 * hey -- you never know what happens!
                 */
                if(ticketLocal!=null)
                {
                    send = new SendBack(ticket_,"[Error an InGate]" +
                     "from Receiver: "
                            +"I just lost security filter \""+name+"\"",
                            ctx);
                }
                System.err.println(
                    "[InGate] I just lost security filter \""+name+"\"");
            }
            catch(Exception e)
            {
                if(ticketLocal!=null)
                {
                    send = new SendBack(ticket_,"[IllegalAgentException]"
                     +"from Receiver: "
                            +"Filter \""+name+"\": "+e.getMessage(), ctx);
                }
                throw new IllegalAgentException(
                    "Filter \""+name+"\": "+e.getMessage());
            }
        }
        /* Let's record some statistics. How much time do
         * we spend in filter pipelines? We only count
         * successful passes.
         */
```

```
        updateStatistics(System.currentTimeMillis() - time);

        /* Software is only as good as its demo, right? Right?
         * NO!!!
         * But the sales droids will never understand that *sigh*.
         * So we invoke a callback to a method that can be used to
         * do some multimedia crap.
         */
        onAccept(ctx);

        /* Publish the agent's context such that other
         * components can find it. This is important.
         */
        launchAgent(ctx);
    }


    /**
     * Launches the agent given by <code>ctx</code>. The agent
     * is <b>not</b> piped through the security filters by this
     * method. Nor is the given <code>AgentContext</code>
     * initialized any further. This method is more or less for
     * internal use only. Regular gateways should not call this
     * method. It is basically used by start-up initialization
     * tools to launch built-in agents and agents that perform
     * basic services and maintainance tasks. The given agent
     * is published in the global <code>Environment</code> using
     * its card as the name and the path for {@link WhatIs
     * WhatIs} key &quot;AGENTS&quot; as the parent path.<p>
     *
     * The <code>AgentCard</code> must have been set up already
     * by the caller.<p>
     *
     * @param ctx The <code>AgentContext</code> of the agent
     *   that shall be launched. The context must be readily
     *   initialized.
     * @exception IllegalAgentException if the given <code>
     *   AgentContext</code> does not have a valid <code>
     *   AgentCard</code>, or another agent with the same
     *   <code>AgentCard</code> is already published.
     * @exception IllegalStateException if <code>ctx</code>
     *   is not in the <code>LOADED</code> state.
     */
    protected void launchAgent(AgentContext ctx)
        throws IllegalAgentException
    {
        long currentTime;
        AgentCard card;
```

```
SendBack send;

try{
    card = (AgentCard)ctx.get(FieldType.CARD);
}
catch(Exception e)
{
    card = null;
}
if (card == null || AgentCard.ROOT.equals(card))
{
    if(ticketLocal!=null)
    {
        send = new SendBack(ticket_,"[IllegalAgentException]" +
         "from Receiver: "
                +"Agent has no valid card!", ctx);
    }
    throw new IllegalAgentException("Agent has no valid card!");
}
/* Publish the context. The context is published
 * without being wrapped in a proxy and it is not
 * removed automatically when the environment of
 * the ingate goes away.
 */
try{
    getEnvironment().publish(
        WhatIs.stringValue("AGENTS")+"/"+card,
        ctx,
        Environment.NO_PROXY | Environment.DETACH
    );
}
catch(ObjectExistsException e)
{
    if(ticketLocal!=null)
    {
        send = new SendBack(ticket_,"[IllegalAgentException]" +
                "from Receiver: "
                +"Agent \""+card+"\" already exists!", ctx);
    }
    throw new IllegalAgentException(
        "Agent \""+card+"\" already exists!");
}

if (debug_)
{
    if(ticketLocal!=null)
    {
        send = new SendBack(ticket_,"[Debug_]" +
```

```
                    "from Receiver: "
                    +"[InGate] Launching agent.", ctx);
            }
            System.err.println("[InGate] Launching agent.");
        }
        /* No we run the agent.
         */
        try{
            ctx.start();
            if(ticketLocal != null){
                send = new SendBack(ticket_,"[Information from Receiver]: " +
                 "Agent arrived. Things work out!", ctx);


            }

        }
        catch(LifecycleException e)
        {
            if(ticketLocal!=null)
            {
                send = new SendBack(ticket_,"[IllegalAgentException]" +
                        "from Receiver: "
                        +e.getMessage(), ctx);
            }
            throw new IllegalAgentException(e.getMessage());
        }
    }
```

A.9 Programmcode von `LSClient`

```
/**
     * Initializes the <code>LSClient</code> from command line.
     *
     * @param argv The command line parameters.
     */
    public static void main(String[] argv)
        throws ArgsParserException, ObjectExistsException
    {
        EventReflector er;
        Environment env;
        ArgsParser p;
        SortedMap map;
        LSClient lsClient;
        String configFile;
```

```
String key;
char[] ch;
URL configUrl;


p = new ArgsParser(OPT_DESCR);
p.parse(argv);

if (p.isDefined("help"))
{
    p.help(System.out);
    return;
}

/* Initialize LSClient:
 * If the option "configurl" is given and the
 * correspondig LDAP server can be requested successfully,
 * a given "configfile" will be ignored.
 */
lsClient = new LSClient();

configUrl  = null;
configFile = null;

if (p.isDefined("configurl"))
{
    try{
        configUrl = new URL(p.stringValue("configurl"));
    }
    catch(Exception e)
    {
        log_.caught( e );
    }
}

if (!lsClient.initLSInfrastructure(configUrl))
{
    if (p.isDefined("configfile"))
    {
        configFile = p.stringValue("configfile");
    }

    lsClient.initLSInfrastructure(configFile);
}


/* Add our event listener to the existing
 * agent event reflector
```

```
         */
        env = Environment.getEnvironment();

        er = (EventReflector) env.lookup(
            WhatIs.stringValue("AGENT_EVENTS"));
        er.addListener(lsClient);


        /* Publish only the LSClient and LSClientService interfaces
         * (by using the mode PLAIN_PROXY)
         */
        env.publish(
            WhatIs.stringValue("ATLAS")+"/client",
            lsClient,
            Environment.PLAIN_PROXY | Environment.DETACH
        );

        env.publish(
            WhatIs.stringValue("ATLAS")+"/client/service",
            lsClient.clientService_,
            Environment.DETACH
        );
    }
```

A.10 Programmcode von `AgentGateway`

```
public interface Out extends AgentGateway
{

    /**
     * Used to query the gateway whether the ticket is acceptable.
     * The gateway checks whether at least on of the URLs in the
     * ticket is supported.
     *
     * @param ticket The ticket to a destination.
     * @return <code>true</code> iff an agent can be sent to at
     *   least on URL being specified by the ticket.
     */
    public boolean acceptsTicket(Ticket ticket);


    /**
     * Transports the given agent to the destination determined
     * from the given ticket.
     *
```

```
         * @param struct The agent’s data as a <code>Resource</code>.
         * @param ticket The agent’s destination.
         * @exception TicketNotSupportedException if this gateway
         *   does not support any of the protocols in <code>ticket
         *   </code>.
         * @exception IOException if an error occurs during transport.
         */
        public void dispatchAgent(AgentContext ctx, Resource struct,
                                            Ticket ticket)
            throws TicketNotSupportedException, IOException,
IllegalAgentException;
     }
```

Anhang B

Akronyme

ACL	Agent Communication Language
ATLAS	Agent Tracking and Location System
DAA	Data Access Agent
DBMS	Database Management System
EPS	Encapsulated PostScript
GT	Global Transaktion
JAR	Java Archive
JPG	siehe JPEG
JT	Joey Transaktion
JVM	Java Virtual Maschine
KT	Kangaroo Transaktion
LAN	Local Area Network
LT	lokale Transaktion
MSC	Message Security Control
MTM	Mobile Transaction Manager
PKCS	Public-Key Cryptography
P3P	Platform for Privacy Preferences
SeMoA	Secure Mobile Agents
P/W	Pass and Visa
SAFER	Secure Agent Fabrication Evolution and Roaming for e-commerce

SSL	Secure Socket Layer
TLS	Transport Layer Security
URL	Uniform Resource Locator
WWW	World Wide Web

Literaturverzeichnis

[1] A. Acharya, M. Ranganathan, and J. Saltz. Dynamic linking for mobile programs. In *In:Proceedings of Mobile Object System: Towards the Programmable Internet*, LNCS, pages 245–262. Springer-Verlag, 1997.

[2] Axel Baldauf. Entwurf auf einer XML basierenden Beschreibungsprache für Benutzerschnittstellen im Kontext von Mobile-Agenten-System. Master's thesis, Fachbereich Informatik, Fachhochschule Konstanz und Fraunhofer Institut für Graphische Datenverarbeitung, 28. April 2003.

[3] A. Biezszad, B. Pagurek, and T. White. Mobile Agents for Network Management. In *In: Proceedings of IEEE Communication Surveys 1*, pages 2–9, 1998.

[4] Peter Braun. *Über die Migration bei mobilen Agenten*. Lehrstuhl für Softwaretechni, Jena, Germany.

[5] D. Chess, B. Grosof, C. Harrison, and A. Kershenbaum. Are you good ideal? In *In: Proceedings of Mobile agents*, number 19887 in RC. IBM Research Report, IBM, 1994, Springer-Verlag, 1997.

[6] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Paris, and Tsudik. Itinerant agents for mobile computing. Technical report, IBM Research Report RC 20010, IBM, 1995.

[7] J. Dale and D. C. Deroure. Towards a framework for developing mobile agents for managing distributed information resources. In *In: Proceedings of Dep. of Electronics and Computer Science*. University of Southamton, 1997.

[8] Yvo Desmedt. Threshold Cryptography. In *In: Proceedings of European Transactions on Telecommunications 5(4)*, pages 449–457, 1994.

[9] Margaret H. Dunham, Abdelsalam Helal, and Santosh Balakrishnan. A mobile transaction model that captures both the data and movement behavior. In *In: Proceedings of Mobile Networks and Applications 2*, pages 149–162, 1997.

[10] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *In: Proceedings of IEEE Transactions on Information Theory 31/4*, 1985.

[11] Christian Erfurth, Peter Braun, and Wilhelm Rossak. Some Thoughts on Migration Intelligence for Mobile Agents. Technical report, Computer Science Department, Friedrich Schiller University Jana, 07740 Jena, Germany, September 2001.

[12] D. Gibert, M. Aparicio, B. Atkinson, S. Brady, Ciccarino, B. Grosof, P. O'Connor, D. Osisek, S. Pritko, R. Spagna, and L. Wilson. IBM intelligent agent strategy. Technical report, UNICOM Seminar on Agent Software, 1995.

[13] Sheng Uei Guan, Tianhan Wang, and Sim Heng Ong. A Secure Approach for Mobile Agent Migration Control. In *In: Proceedings of Seventh Symposium on Computers and Comunications (ISCC'02)*, 2002.

[14] Fritz Hohl. Time Limited Blackbox Security: Protecting Mobile Agents from Mali- cious Hosts. In *In: Proceedinhs of Lecture Notes in Computer Science 1419*, page 92 ff, 1998.

[15] L. Hurst, P. Cunningham, and F. Sommers. Mobile agents-smart messages. In *In: Proceedings of the 1st International Workshop on Mobile Agents*, 1997.

[16] Wayner Jansen and Tom Karygiannis. Mobile Agents Security. In *In: Prceedings of Technical Report*. Unpublisher Draft, 1999.

[17] G. Karjoth and C. Gulcu. Protecting the Computation Results of Free Roaming Agents. In *In: Proceedings of In Lecture Notes in Computer Science*, pages 195–207. Spinger-Verlag, 1998.

[18] Neeran M. Karnik. Security in Mobile Agents System. In *In: Proceedings of Dissertation*. University of Minnesota, Oktober 1998.

[19] D. Keith, K. D. Kotay, and D. Kotz. Transportable agents. In *In: Proceeding of CIKM workshop on Intelligent Information Agents at 3rd International Conference on Information and Knowledge Management*, 1994.

[20] D. Kotz and R. S. Gray. Mobile Agents and the Future of the Internet. Technical report, ACM Operating System Review, 1999.

[21] D. Kotz, R. S. Gray, and D. Rus. Transportable Agents Support Worldwide Applications. In *In: Proceedings of the 7th SIGOPS European Workshop*, 1996.

[22] E. Kovacs, K. Röhrle, and M. Reich. Mobile Agents OnTheMove - Integrating an Agent System into the Mobile Middleware. In *In: Proceedings of the ACTS Mobile Summit*, 1998.

[23] Linda Mathew and Daniela Hermann. Leifaden Simulationsstudie. In *In: Proceedings of Leitfaden zur Bedienung der Anwendung, die im Rahmen der Simulationsstudie in Darmstadt zum Einsatz kommen*, 29./30.April 2002.

[24] Stivens Milic. Sichere Kommunikation zwischen Mobilen Agenten. Master's thesis, Johann Wolfgang Goethe-Universität Frankfurt und Fraunhofer Institut für Graphische Datenverarbeitung, 27. Dezember 2002.

[25] R. Montanary, C. Stefanelli, and G. Tonti. A Policy-based Mobile Agent Infrastructur . In *In: Proceedings of International Symposium on Applications and the Internet (SAINT'03)*, Orlando, Gennaio, 2003. IEEE Computer Society.

[26] Hyacrinth S. Nwana. Software agents: An overview. Technical report, Knowledge Engineering Review, 11(3):205-204, 1996.

[27] H. Peine. An introduction to mobile agents programming and Ara System. In *In: Proceedings of Deparment of Computer Science*. University of Keiserslautern,Germany, 1997.

[28] Ulrich Pinsdorf and Christoph Bush. *Neu Potenziale für mobile Diensleistung*. Multimedia-Arbeitsplatz der Zukunft.

[29] Stefan Pleisch. State of the Art of Mobile Agent Computing - Security, Fault Tolerance, and Transaction Support. In *In: Prceedings of Research Report*, number 3152 in RZ. IBM Zurich Research Laboratory, Juni 1999.

[30] Adi Shamir und Leonard Adleman Ronald L. Rivest. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In *In: Proceedings of Communications of the ACM 21(2)*, pages 120–126, Februar 1978.

[31] Alexander Rossnagel, Rotraud Gitter, and Roland Steidle. Technische Gestaltungsvorschläge. Technical report, provet Universität Kassel, January 2003.

[32] Alfredo De Santis and Yvo Desmedt nad Yair Frankel und Moti Yung. How to Share a Functi- on Securely. In *In: Proceedings of 26th Annual ACM Symposium on Theory of Computing*, STOC, pages 522–533, 1994.

[33] F.B. Schneider. Towards Mobile Cryptography. In *In: Proceedings of IEEE Proceedings of Security und Privacy*, Mai 1998.

[34] Fred B. Schneider. Towards Fault-tolerant and Secure Agentry. In *In: Prceedings of In 11th Int. Workshop on Distributed Algorithms*, Saarbrücken, Germany, September 1997.

[35] Adi Shamir. How to Share a Secret. In *In: Proceedings of the ACM 22(11)*, pages 612–613, November 1979.

[36] Tomas Sander und Christian F. Tschudin. Detecting Attacks on Mobile Agents. In *In: Proceedings of In Workshop on Foundations for Secure Mobile Code*, number 20375 in DC, Washington, 1997.

[37] Tomas Sander und Christian F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In *In: Proceedings of In Mobile Agents and Security*, pages 44–60. Springer-Verlag, 1998.

[38] Michael K. Reiter und Kenneth P. Birman. How to Securely Replicate Services. In *In: Proceedings of ACM Transactions on Programming Languages and Systems 16(3)*, pages 986–1009, Mai 1994.

[39] J. Vitek, M. Serrano, and D. Thanos. Security and Communication in Mobile Object Systems. In J.Vitek and C.Tschudin, editors, *In: Proceedings of Mobile Object Systems: Towards the Programmable Internet*, number 1222 in LNCS. Springer-Verlag, 1997.

[40] J. E. White. Mobile Agents White Paper. Technical report, General Magic, Inc, 1996.

[41] B.S. Yee. A Sanctuary For Mobile Agents. In *In: Prceedings of In Third Workshop on Mobile Object Systems, Jgg. Technical Report*, number 537 in CS97, UC San Diego, April 1997.