

Prüfer: Prof. Dr. Kurt Rothermel

Betreuer: Fritz Hohl

Begonnen am: 15.12.98

Beendet am: 14.06.99

CR-Nummer: C.2.4, D.4.8, J.1.3, C.4

Diplomarbeit Nr. 1750

**Abrechnung erbrachter
Dienstleistungen in
Mobile-Agenten-Systeme**

Sven Tränkle

Kurzfassung

Unter Mobilien Agenten versteht man Programmkonstrukte, die die Fähigkeit haben, autonom zu handeln, zu kommunizieren und von Rechner zu Rechner zu migrieren, falls auf beiden Seiten ein Agentensystem installiert ist. Das Modell der Mobilien Agenten ist ein neuer, vielversprechender Ansatz im Bereich der Verteilten Systeme. Die Abteilung Verteilte Systeme forscht auf diesem Gebiet und erstellt im Rahmen des Projektes Mole ein System, mit dem Mobile Agenten eingesetzt werden können.

Ein kommerzieller Einsatz solch eines Systems ist nur denkbar, wenn es eine Möglichkeit gibt, erbrachte Dienstleistungen abzurechnen. Dabei handelt es sich zum einen um Dienstleistungen, die das System zur Verfügung stellt (z.B. Bereitstellung von Systemressourcen), zum anderen um Dienstleistungen, die von den Agenten selber angeboten werden.

Hierzu sind Mechanismen notwendig, die alle erbrachten und benutzten Dienste mitprotokollieren. Aufbauend auf diesen Daten können die Dienste dann mit den Dienstbenutzern abgerechnet werden.

Im Idealfall findet die Bezahlung der Dienstleistungen ebenfalls auf dem elektronischen Weg statt, z.B. unter Zuhilfenahme eines elektronischen Zahlungssystems.

Im Rahmen dieser Diplomarbeit wurde eine Abrechnungskomponente für Mobile-Agenten-Systeme im allgemeinen und für das Mole-Agentensystem im speziellen entwickelt. Die entwickelte Abrechnungskomponente wurde dann prototypisch in Mole implementiert.

Inhaltsverzeichnis

1	Einleitung	1
2	Einführung.....	3
2.1	Agententechnologie	3
2.1.1	Agenten	3
2.1.2	Mobile Agenten.....	3
2.1.3	Agentensysteme	5
2.1.4	Das Agentensystem Mole	5
2.1.5	Andere Agentensysteme.....	6
2.2	Accounting	6
2.3	Zahlungssysteme.....	12
2.3.1	Einführung.....	12
2.3.2	Kategorisierung der vorhandenen Zahlungssysteme.....	12
2.3.3	Bewertung der Kategorien von Zahlungssystemen.....	14
2.3.4	Rechtliche Aspekte.....	15
2.3.5	Existierende Zahlungssysteme in Deutschland.....	15
3	Entwurf.....	16
3.1	Komponente zur Erfassung der Dienstleistungen (Accounting).....	17
3.1.1	Überwachte Ressourcen	17
3.1.2	Überwachte Agenten	20
3.1.3	Datenhaltung der Accounting-Daten.....	21
3.2	Komponente zur Berechnung der Gebühren.....	22
3.2.1	Gebührenberechnung	22
3.2.2	Abrechnungszeitpunkt.....	24
3.2.3	Zahlungsunfähigkeit.....	24
3.3	Zahlungskomponente	25
3.3.1	Transaktionsnummer	28
4	Spezifikation des Prototypen	29
4.1	Spezifikation der Accountingkomponente	29
4.1.1	Ressourcen / Dienste	29
4.1.2	Konfiguration	30
4.1.3	Accounting-Datenhaltung	31
4.2	Spezifikation der Gebührenberechnungskomponente.....	31
4.2.1	Berechnung der Gebühren über eine dafür bereitgestellte Klasse	32
4.2.2	Konfiguration und Berechnung der Gebühren mit systemeigenen Methoden	33
4.2.3	Datenhaltung	35
4.2.4	Abrechnung	35
4.2.5	Währungseinheit der Geldbeträge.....	36
4.2.6	Abfrage der Gebühren durch Agenten	36
4.3	Spezifikation der Zahlungskomponente.....	37
4.3.1	Definition der Schnittstellen: Mole <-> Zahlungssystem	37
4.3.2	Prototypische Zahlungskomponente	39
4.4	Interaktion der einzelnen Komponenten in einem Beispiel	40
5	Implementierung	42
5.1	Implementierung der Accountingkomponente	42
5.1.1	Accounting der einzelnen Ressourcen	42
5.1.2	Accounting von Diensten	46
5.1.3	Implementierung der Accounting-Datenhaltung.....	46
5.2	Implementierung der Gebührenberechnungskomponente	50
5.2.1	Gebührenberechnung	50
5.2.2	Gebühren-Datenhaltung	50

5.2.3	Gebührenabrechnung	51
5.2.4	Abrechnungszeitpunkt.....	52
5.2.5	Abfrage der Gebührenstruktur	53
5.2.6	Transaktionsidentifikation.....	54
5.3	Implementierung der Zahlungskomponente	54
6	Schnittstellenbeschreibung	57
6.1	Schnittstellenbeschreibung für Zahlungssysteme	57
6.2	Schnittstellenbeschreibung für eigene Gebührenberechnung	58
6.3	Schnittstellenbeschreibung für Agentenprogrammierer	58
7	Messungen.....	63
8	Zusammenfassung und Ausblick.....	65
Anhang	66
A	Quellcode des für die Messung verwendeten Testagenten	66
B	Quellcode der Klasse Accounting	68
Literaturverzeichnis	70

Abbildungsverzeichnis

ABBILDUNG 1:	Speicherbelegung eines Prozesses während seiner Laufzeit	8
ABBILDUNG 2:	Speicherbelegung eines Prozesses während seiner Laufzeit und periodische Abtastung durch den Speicher-Accounting-Prozeß	9
ABBILDUNG 3:	Kategorisierung elektronischer Zahlungssysteme im Internet.....	14
ABBILDUNG 4:	Komponenten eines Abrechnungssystems für Mobile-Agenten-Systeme	16
ABBILDUNG 5:	Näherungen des Speicherverbrauchs	19
ABBILDUNG 6:	Entwurf der Accounting-Datenstruktur	22
ABBILDUNG 7:	Integration des Zahlungssystems	27
ABBILDUNG 8:	Integration mehrerer Zahlungssysteme.....	28
ABBILDUNG 9:	Schnittstellen zwischen Agent (auch Location) und dem Schnittstellenobjekt PaymentObject	38
ABBILDUNG 10:	Direkte Kommunikation der PaymentObjects zu Test- und Demonstrationszwecken	40
ABBILDUNG 11:	Ablauf einer Dienstleistung mit Bezahlung.....	41
ABBILDUNG 12:	Zustellung einer Message in Mole.....	44
ABBILDUNG 13:	Ablauf eines RPCs	45
ABBILDUNG 14:	Meldung einer Dienstleistung an die Accounting-Komponente...	46
ABBILDUNG 15:	Die Accounting-Datenstruktur.....	49
ABBILDUNG 16:	Weiterleitung der Nachrichten	55

1 Einleitung

Im Bereich der Verteilten System existiert seit einiger Zeit das Forschungsgebiet der Mobilten Agenten. Dieses Gebiet beschäftigt sich mit autonom arbeitenden Softwareprogrammen, die zu mehreren parallel ein Problem lösen können und in der Lage sind, ihren Aufenthaltsort selbstständig zu ändern: den Mobilten Agenten. Die Voraussetzung hierfür ist, daß an allen potentiellen Aufenthaltsorten ein sogenanntes Agentensystem installiert ist. Ein Agentensystem ist eine Art Laufzeitumgebung für Mobile Agenten, das den Agenten unter anderem Mechanismen zur Kommunikation und zur Migration bereitstellt.

Die Haupteinsatzmöglichkeiten von Mobilten Agenten werden derzeit im Bereich des elektronischen Handels (Electronic Commerce), des elektronischen Diensteverkehrs, als Grundlage für die Anbindung mobiler Endgeräte sowie in anderen Gebieten der Verteilten Systeme gesehen.

Agentensysteme werden zur Zeit an mehreren Universitäten weltweit sowie bei einigen Firmen, hauptsächlich zu Forschungszwecken, entwickelt. Es existieren auch wenige kommerzielle Agentensysteme auf dem Markt.

Auch an der Universität Stuttgart wird in der Abteilung Verteilte Systeme des Instituts für Parallele und Verteilte Höchstleistungsrechner im Bereich der Mobilten Agenten geforscht. In der Projektgruppe „Mole“ wurde dabei auch ein Agentensystem gleichen Namens entwickelt und implementiert. Das Agentensystem Mole ist in der Programmiersprache Java implementiert. Dieses ermöglicht den Einsatz des Systems auf vielen verschiedenen Hardware-Architekturen und Betriebssystemen.

Dem Mole-System fehlt zum jetzigen Zeitpunkt jedoch noch eine Komponente, die für den kommerziellen Einsatz besonders im Bereich des elektronischen Handels und der elektronischen Dienste zwingend notwendig ist: die Abrechnung von erbrachten Dienstleistungen. Dabei handelt es sich sowohl um Dienstleistungen, die vom System zur Verfügung gestellt werden, als auch um Dienste, die von Agenten selbst angeboten werden. Dienste, die das System zur Verfügung stellt, sind hauptsächlich die Benutzung von Systemressourcen wie CPU-Zeit, Speicher oder Netzwerk. Es sind aber auch andere Dienste denkbar. Die von Agenten zur Verfügung gestellten Dienste sind vielfältig. Denkbar sind zum Beispiel Datenbankabfragen, Berechnungen, aber auch der Verkauf von Waren, die nicht zwangsläufig auf dem elektronischen Weg zustellbar sein müssen.

Im kommerziellen Einsatz müssen die Dienste, die von Agenten geleistet und in Anspruch genommen werden, registriert und zu einem späteren Zeitpunkt abgerechnet werden. Dabei bietet sich an, die Abrechnung ebenfalls auf dem elektronischen Weg durchzuführen, zum Beispiel unter Zuhilfenahme von elektronischen Zahlungssystemen.

Im Rahmen dieser Diplomarbeit soll ein solches Abrechnungssystem entwickelt und implementiert werden.

Die Ausarbeitung gliedert sich in sieben Kapitel. Nach der allgemeinen Einführung in die Agententechnologie, in das Accounting von Diensten und in elektronische Zahlungssysteme, schildert Kapitel drei den Entwurf eines System zur Abrechnung erbrachter Dienstleistungen in Mobile-Agenten-Systemen. Das vierte Kapitel gibt die Spezifikation eines Abrechnungssystem für das Agentensystem Mole wieder, dessen Implementierung im fünften Kapitel beschrieben wird. Das sechste Kapitel beschreibt

die neu implementierten Schnittstellen des Agentensystems Mole. In Kapitel sieben werden Messungen am neuen und am alten Mole-System präsentiert. Kapitel acht faßt die Ergebnisse zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

2 Einführung

Das nächste Kapitel führt in die Thematik von Agentensystemen, des Accountings von Ressourcen und Dienstleistungen sowie des elektronischen Bezahlen in Computersystemen ein.

2.1 Agententechnologie

Im Folgenden wird der Begriff des Agenten, Mobile Agenten, einige Agentensysteme und im speziellen das Mole Agentensystem vorgestellt.

2.1.1 Agenten

Im Bereich der Informatik existieren viele verschiedene Definitionen von **Agenten**, die nach Sichtweise und Einsatzgebiet sehr differieren¹. Exemplarisch seien hier zwei davon herausgegriffen.

Laut der Definition des Duden² handelt es sich bei einem Agenten um „*jemanden [...], der im Auftrag für einen anderen eine Aufgabe erledigt*“.

Die zweite Agentendefinition, die hier vorgestellt wird, ist die von [WJ95]. Für die Autoren ist ein Agent ein unabhängiges Programm, das in der Lage ist, seine Entscheidungen und sein Handeln, basierend auf der Wahrnehmung seiner Umwelt, bei der Verfolgung eines oder mehrerer Ziele selbständig zu kontrollieren.

In [WJ95] werden folgende Eigenschaften beschrieben, die ein Programm erfüllen muß, damit es als Agent angesehen werden kann:

- **Autonomie:** Die Agenten lösen eine ihnen gestellte Aufgabe selbständig, d.h. ohne weitere Benutzereingriffe und mit eigenständiger Kontrolle über ihr Handeln und ihren internen Status.
- **Kooperationsfähigkeit:** Agenten sind in der Lage, mit anderen Agenten und Anwendern zusammenzuarbeiten³.
- **Reaktionsfähigkeit:** Agenten beobachten ihre Umgebung und reagieren auf auftretende Veränderungen.
- **Unternehmungsgest:** Agenten reagieren nicht nur auf Veränderungen, sie ergreifen sogar die Initiative bei der Verfolgung ihrer Ziele.

2.1.2 Mobile Agenten

Mobile Agenten sind mit den Agenten, wie sie im letzten Kapitel beschrieben wurden, verwandt, d.h. sie erfüllen zum Teil die Forderungen und Eigenschaften, die für Agenten gelten, wie z.B. Autonomie und Kooperationsfähigkeit. Zusätzlich dazu besitzen Mobile Agenten noch eine weitere Eigenschaft. Sie sind, wie der Name schon sagt, *mobil*, d.h. sie können ihren Aufenthaltsort ändern; man nennt dies migrieren. Ein

1. vgl. [Se94], [Ou95], [GK94], [Du89], [WJ95] und [Fo93]

2. siehe [Du89]

3. siehe [GK94]

Agent beendet seinen Programmablauf und setzt ihn an einem anderen Ort wieder fort. Dabei handelt es sich für den Agenten um einen transparenten Mechanismus. Er signalisiert dem System, wohin er migrieren will. Daraufhin wird er eingefroren. Wenn er wieder aufwacht, befindet er sich an dem spezifizierten Ort. Die Migration von Agenten stellt die Grundlage für ein neues Programmierparadigma dar, das für den Einsatz im Bereich der Verteilten Systeme gut geeignet zu sein scheint.

Während eine normale Prozeßmigration auf einer mehr oder weniger eng gekoppelten Mehrprozessormaschine oder innerhalb eines Workstationclusters durchgeführt wird, können sich mobile Agenten frei auf beliebigen vernetzten heterogenen Rechnern bewegen; im Fall des Internets also weltweit. Die einzige Voraussetzung hierfür ist, daß auf allen beteiligten Rechnern ein kompatibles Agentensystem läuft (siehe Kapitel 2.1.3). Wie oben schon erwähnt, müssen diese Rechner natürlich vernetzt sein und über ein beliebiges Netzwerkprotokoll kommunizieren können, z.B. TCP/IP im Internet.

Die bisher in weit verteilten Systemen verwendeten „Remote Procedure Calls“ (RPC) folgen dem Client/Server-Prinzip. Nach Aufrufen eines RPCs sendet der Client die Daten bzw. die Anfragen direkt über das Netzwerk zum Server. Auf diesem wird die Prozedur ausgeführt. Die Ergebnisdaten werden über das Netzwerk zum Client zurückgesendet. Dieser Vorgang geschieht transparent für den Aufrufer. Das RPC-System simuliert ihm weitestgehend einen normalen Prozeduraufruf.

Bei Agentensystemen mit mobilen Agenten gibt es mehrere Möglichkeiten. Der mobile Agent kann RPCs an anderen Orten, an denen sich ein Agentensystem befindet, aufrufen oder Nachrichten mit Agenten, die sich an anderen Orten befinden, austauschen. Er kann aber auch, wenn er es für sinnvoll bzw. effizient hält, zu dem Ort migrieren, an dem sich die gewünschten Informationen befinden. Er kann dann seine Aufgabe vor Ort erfüllen, d.h., daß der Programmcode zur Berechnung der Ergebnisse, der sich im mobilen Agenten befindet über das Netzwerk gesendet und lokal auf dem Zielrechner ausgeführt wird. Dies kann vorteilhaft sein, wenn große Mengen an Daten verarbeitet oder durchsucht werden müssen. Die Daten müssen nicht komplett über das Netzwerk transportiert werden, sondern lediglich der Agent, der den Programmcode zur Bearbeitung der Daten beinhaltet. Der Agent bearbeitet die Daten dann vor Ort und sendet die Ergebnisse zurück bzw. migriert zurück und nimmt die Ergebnisse mit.

Mit diesem Ansatz läßt sich keine vollkommen neue Funktionalität modellieren. Durch die lokale Ausführung des Programmcodes auf dem Server und der damit möglicherweise verbundenen Einsparung von Datentransporten über das Netzwerk vereinfacht sich jedoch die Programmierung vieler Aufgaben erheblich, da z.B. auf weniger Fehlerbedingungen Rücksicht genommen werden muß¹.

Mobile Agenten bieten weiterhin die Möglichkeit, Probleme massiv parallel zu lösen, indem ein Problem von mehreren Agenten parallel verarbeitet wird. Zur Synchronisation und Absprache der mobilen Agenten untereinander existieren diverse Kommunikationsmethoden, die das Agentensystem zur Verfügung stellt.

1. vgl. [Ho95], [Ho96]

Somit bieten mobile Agenten ein neues Programmierparadigma, mit dem bestimmte Probleme in Verteilten Systemen einfacher und effizienter als mit anderen Programmiermodellen zu lösen sind.

Die Einsatzgebiete dieser neuen Technologie werden im Bereich des elektronischen Diensteverkehrs gesehen, im elektronischen Handel (Electronic Commerce), als Grundlage für die Anbindung mobiler Endgeräte sowie in anderen Gebieten der Verteilten Systeme.

2.1.3 Agentensysteme

Alle Agenten können nur innerhalb eines **Agentensystems** existieren. Das Agentensystem ermöglicht überhaupt erst die Ausführung der Agenten, d.h das Agentensystem bildet den Rahmen, in dem sich Agenten bewegen können. Es stellt die Orte zur Verfügung, an denen sich die Agenten aufhalten können. Desweiteren ermöglicht es den Nachrichtenaustausch zwischen den Agenten, stellt den Mechanismus zur Migration zur Verfügung sowie diverse andere, teilweise individuelle Dienste, die von den Agenten genutzt werden können. Es regelt und ermöglicht außerdem den Zugriff auf Systemressourcen.

2.1.4 Das Agentensystem Mole

Mole ist ein Agentensystem, das in der Abteilung Verteilte Systeme am Institut für Parallele und Verteilte Höchstleistungsrechner der Universität Stuttgart entwickelt wurde. Es entstand im März 1995 im Rahmen einer Diplomarbeit¹. Die aktuelle Version ist Mole 3.0.

Das Agentensystem und die Agenten sind in der Programmiersprache Java entwickelt (Version 3.0 basiert auf Java 1.1). Java stellt einige der benötigten Sicherheitsmechanismen sowie die Möglichkeit einer plattformunabhängigen Implementierung zur Verfügung. Diese ermöglicht die Migration von Agenten auf beliebige Rechnerarchitekturen, für die eine Java-Laufzeitumgebung vorhanden ist.

Das Mole-Agentensystem besteht im wesentlichen aus zwei Teilen: den Agenten und den Orten (Locations), auf denen die Agenten ausgeführt werden, wobei Mole in der Lage ist, mehrere Locations innerhalb eines Verwaltungsprozesses (Engine) auszuführen, d.h. es muß nicht für jede Location ein eigener Java-Interpreter gestartet werden. Dies kann Rechenzeit und andere Systemressourcen sparen.

Mole unterscheidet zwei Klassen von Agenten:

1. Ortsfeste Systemagenten: Diese Agenten können auf beliebigen Locations gestartet werden, aber nicht migrieren. Sie können auf lokale Systemressourcen zugreifen.
2. Benutzeragenten: Diese Agenten können nicht auf lokale Systemressourcen zugreifen. Sie sind in der Lage, ihren Aufenthaltsort zu wechseln, d.h. sie können migrieren, wenn sie als mobil spezifiziert werden.

1. siehe [Ho95]

Die Kommunikationsmechanismen, die von Mole bereitgestellt werden sind Nachrichten und RPCs.

Nachrichten können von Agenten an beliebige andere Agenten oder Locations auch über Location-Grenzen hinaus verschickt werden. Über Nachrichten können beliebige serialisierbare Java-Objekte verschickt werden. Die Kommunikation erfolgt asynchron, d.h. der Agent wird nicht bis zur Zustellung der Nachricht blockiert, sondern läuft direkt nach dem Versenden der Nachricht weiter.

RPCs (Remote Procedure Calls) sind synchrone Kommunikationsmechanismen. Agenten können „öffentliche“ Methoden beliebiger anderer Agenten und Locations auch über Location-Grenzen hinaus aufrufen. Der Agent wird blockiert und wartet, bis die Gegenstelle die aufgerufene Methode abgearbeitet und das Ergebnis zurückgeliefert hat.

2.1.5 Andere Agentensysteme

Neben Mole gibt es noch diverse andere Agentensysteme, von denen hier nur zwei kurz genannt werden sollen.

- **D'Agents**¹: Ein Agentensystem, das am Center for Mobile Computing des Dartmouth College, Hanover, New Hampshire entwickelt wird. Agenten können in verschiedenen Sprachen, u.a. Java und TCL, geschrieben werden.

Besonders interessant ist das geplante Konzept zur Ressourcenkontrolle. Es sollen marktbasierende Prinzipien zur Ressourcenkontrolle eingeführt werden, d.h., daß Anbieter von Ressourcen diese an die anderen Agenten im System verkaufen. Jede Ressourcenbenutzung im System muß bezahlt werden. Dies verhindert, daß Agenten durch exzessive Ressourcenverwendung das System lahmlegen. Jeder Agent bekommt die gleiche Menge Geld und kann somit die gleiche Menge Ressourcen verbrauchen. Dies garantiert auch die Fairneß bei der Ressourcenverteilung.

- **Telescript**²: Ein von der Firma General Magic entwickeltes kommerzielles Agentensystem basierend auf C/C++. Die Agenten werden in einer eigenen objektorientierten Sprache, der „Telescript Language“, programmiert.

In Telescript gab es bereits Accounting-Mechanismen und ein rudimentäres Abrechnungssystem. Die Dienste wurden in Teleclicks abgerechnet. Genaue Informationen über die Accounting- und Abrechnungskomponente sind aber schwer zu finden.

Telescript hat wesentlich zum aktuellen Interesse an mobilen Agenten beigetragen, seine Entwicklung wurde jedoch eingestellt.

2.2 Accounting

Die Grundlage für das Abrechnen von Dienstleistungen auch in Agentensystemen ist das sogenannte „Accounting“ (engl. Accounting: Buchhaltung, Berechnung). Unter „Accounting“ versteht man ganz allgemein das Zählen der Inanspruchnahmen beliebiger Dienste. Im EDV-Bereich versteht man darunter entsprechend das Mitprotokollieren.

1. siehe [BKR98], [GKC98]

2. siehe [GM94]

ren von Dienstbenutzungen. Hierunter fallen die sogenannten Betriebsmittel eines Rechners (Prozessor, Speicher, Festplatte, etc.), seine Infrastruktur (Netzwerk, Modem, etc.) sowie die von Computerprogrammen angebotene Dienste (typischerweise Anfragen an Datenbanken, Verzeichnisdienste, u.ä.). Die Inanspruchnahmen dieser Dienste werden beim Accounting mitprotokolliert und gespeichert.

Das Accounting dient mehreren Zwecken. Eines der häufigsten Einsatzgebiete ist die Erhebung von Daten zur späteren Abrechnung der geleisteten Dienste und zur Verfügung gestellten Systemressourcen. Den Vorgang der Abrechnung dieser Dienste nennt man auch „Billing“. Typische Szenarien sind in diesem Fall u.a. Network Service Provider, hierbei beschränkt sich das Accounting dann in der Regel auf die Netzwerkbenutzung. Der Network Service Provider fordert später, bei volumen- oder zeitabhängiger Abrechnung, die Zahlung der mittels des Accounting berechneten Gebühren von seinen Kunden. Weitere typische Szenarien sind Rechenzentren, die ihre Rechner externen Benutzern kostenpflichtig zur Verfügung stellen. Das Accounting ermittelt hierbei pro Benutzer die verbrauchten Ressourcen (hauptsächlich CPU-Zeit, Speicherbenutzung und Festplattenbenutzung), die später in Rechnung gestellt werden.

Ein zweites Einsatzgebiet des Accounting ist das Erstellen von Statistiken zum Aufspüren von Systemengpässen. Durch Analyse der durch das Accounting gewonnenen Daten können stark nachgefragte Ressourcen des Systems erkannt und daraufhin erweitert werden.

Im Folgenden werden die Ressourcen und Dienste, die beim Accounting überwacht werden, vorgestellt. Dabei wird auch kurz auf die Möglichkeiten der Überwachung der einzelnen Dienste sowie die Einheiten, in der man sie messen kann, eingegangen.

Systemressourcen:

- Prozessor- (CPU-) Laufzeit:

Die Prozessoren eines Systems führen, solange sie nicht inaktiv sind, Befehle aus. Diese Befehle gehören zu Prozessen, die auf dem System laufen. In heutigen Systemen (Multi-Tasking / Multi-Processing) belegt ein Prozeß keinen Prozessor alleine für sich, sondern die einzelnen Prozesse, die im System laufen, teilen sich die vorhandenen Prozessoren, indem immer ein Prozeß einen Prozessor für eine bestimmte Zeitdauer (Zeitscheibe) zugesprochen bekommt. Ist die Zeitscheibe abgelaufen, bekommt der nächste Prozeß die Prozessorzeit für eine Zeitscheibe u.s.w. Die Länge einer Zeitscheibe liegt im Millisekundenbereich. Wenn man nun alle Zeitscheiben eines Prozesses addiert, erhält man die Prozessorlaufzeit eines Prozesses. Addiert man die Zeitscheiben aller Prozesse eines bestimmten Benutzers, so bekommt man die von diesem Benutzer verbrauchte Prozessorlaufzeit (CPU-Accounting auf Benutzerebene).

Die Verteilung der Prozessorzeit auf die einzelnen Prozesse geschieht für die Prozesse transparent, d.h. die einzelnen Prozesse sind an dieser Verteilung nicht beteiligt, sie haben quasi alle einen eigenen virtuellen Prozessor. Aus diesem Grund ist es auf Prozeßebene nicht möglich, die verbrauchte Prozessorlaufzeit zu messen. Die Accountingmechanismen, die hierfür verantwortlich sind, müssen also tiefer auf Betriebssystemebene sitzen. Die Betriebssystemkomponente, die für die Verteilung der Prozessorlaufzeit auf die einzelnen Prozesse zuständig ist, ist der sogenannte „Scheduler“ (am. engl. to schedule: festlegen, planen). Innerhalb dieses Schedulers

ist es am einfachsten, die Prozessorlaufzeit auf Prozeß- oder Benutzerebene zu accounten.

- Hauptspeicherverbrauch:

Prozesse belegen in der Regel Speicherplatz im Hauptspeicher des Systems. Die Speicherbelegung ist über die Lebenszeit eines Prozesses aber nicht zwangsläufig konstant. Prozesse können während ihrer Laufzeit neuen Hauptspeicher anfordern und belegten Speicher wieder freigeben. Um also ein Maß für die Speicherbelegung zu finden, muß man auch die Zeitdauer der Belegung berücksichtigen. Die folgende Abbildung zeigt, wie die Speicherbelegung eines Prozesses über seine Laufzeit gesehen aussehen könnte:

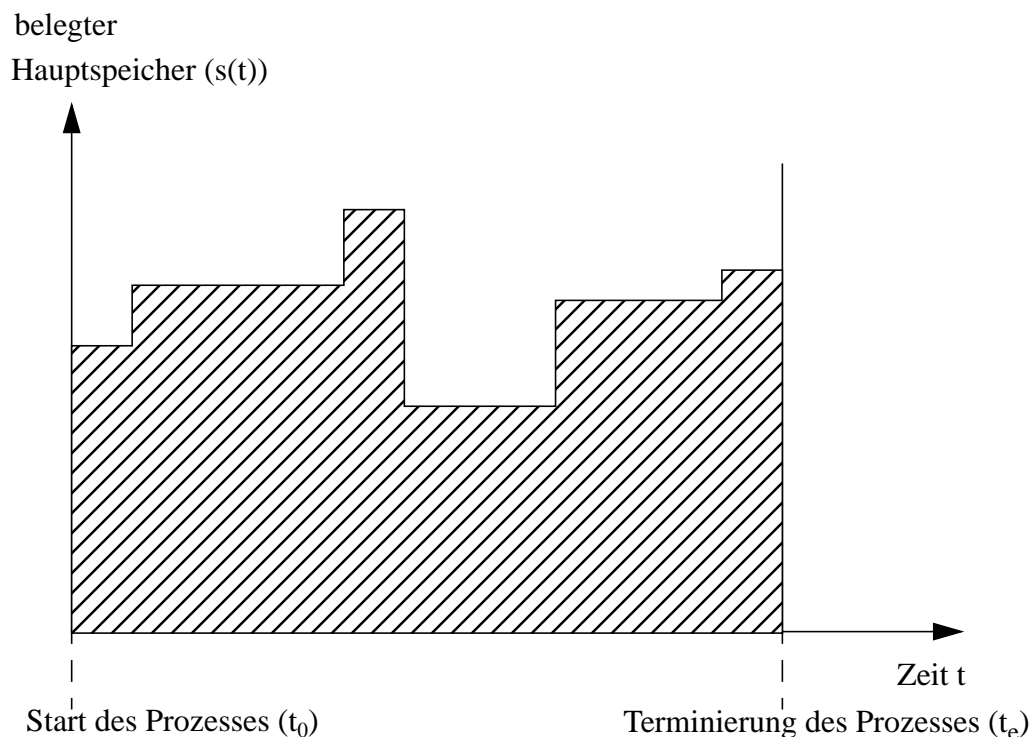


ABBILDUNG 1. Speicherbelegung eines Prozesses während seiner Laufzeit

Ein repräsentatives Maß für den Hauptspeicherverbrauch wäre die schraffierte Fläche unter der Funktion $s(t)$. Sie berechnet sich folgendermaßen:

$$\text{Speicherverbrauch} = \int_{t_0}^{t_e} s(t) dt$$

Diese exakte Berechnung des Speicherverbrauchs ist aber nur möglich, wenn man zu jeder Zeit die aktuelle Speicherbelegung des Prozesses kennt. Dies könnte dadurch realisiert werden, daß man die Accountingmechanismen zur Feststellung des Speicherverbrauchs in die Betriebssystemroutinen zur Hauptspeicherbelegung und Hauptspeicherfreigabe integriert. Auf diese Weise ist sichergestellt, daß jede Veränderung des belegten Speichers eines Prozesses bemerkt wird.

Wenn dies aus bestimmten Gründen nicht möglich ist, es aber eine Möglichkeit gibt, die Größe des belegten Speichers eines Prozesses zu jedem beliebigen Zeitpunkt festzustellen, so kann man sich mit einer Näherung helfen. Dazu überprüft ein eigener Speicher-Accounting-Prozeß periodisch die Speicherbelegung aller Prozesse. Die folgende Abbildung zeigt die Speicherbelegung eines Prozesses über seine Lebensdauer hinweg sowie die periodischen Speicherbelegungstests des Speicher-Accounting-Prozesses:

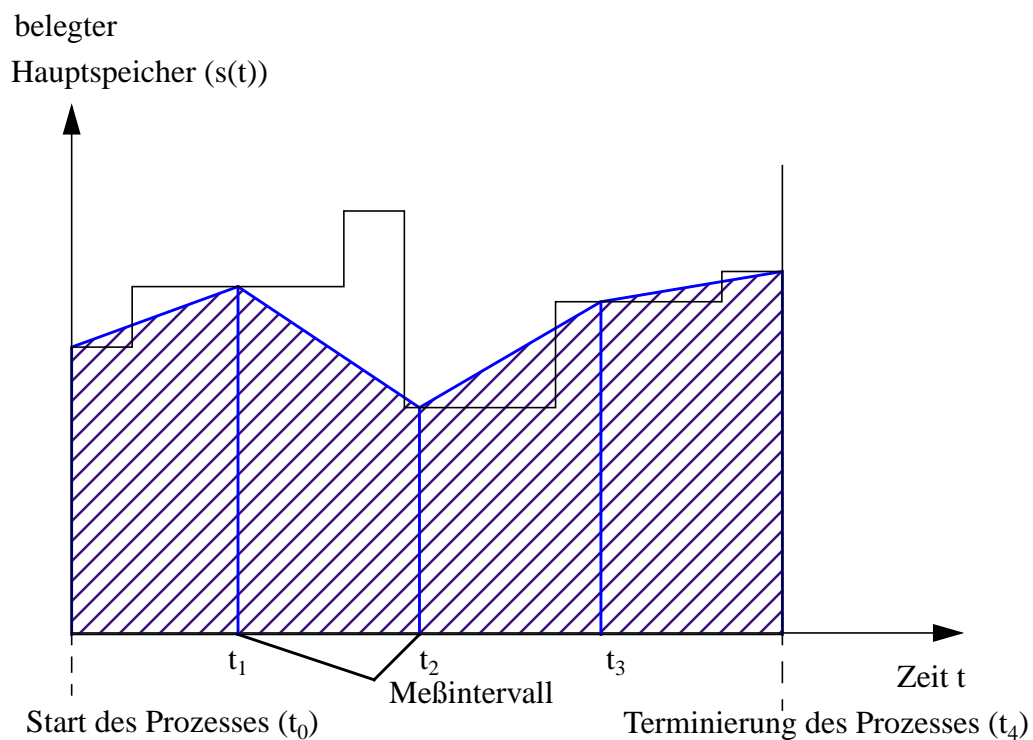


ABBILDUNG 2. Speicherbelegung eines Prozesses während seiner Laufzeit und periodische Abtastung durch den Speicher-Accounting-Prozeß

Aus den periodischen Abtastungen der Speicherbelegung des Prozesses läßt sich dann folgendermaßen eine Näherung des Speicherverbrauchs errechnen:

$$\text{Speicherverbrauch} \approx \sum_{k=1}^n (t_{n+1} - t_n) \left(\frac{|s(t_{n+1}) + s(t_n)|}{2} \right)$$

Dies entspricht in der obigen Abbildung der blau schraffierten Fläche. Zur Erinnerung: der exakte Wert des Speicherverbrauchs ist die Fläche unter der Funktion. Je kleiner das Meßintervall gewählt wird, desto genauer ist diese Näherung. Das Problem hierbei ist jedoch die Systembelastung durch den Speicher-Accounting-Prozeß. Besonders wenn die Funktion zur Messung des belegten Speichers zu einem bestimmten Zeitpunkt aufwendig ist, tritt bei klein gewähltem Meßintervall eine in keinem Maß zum Nutzen stehende Systembelastung auf. Die Größe des Meßintervalls ist also im Einzelfall genau zu überlegen.

- Festplattenspeicher:

Festplattenspeicherverbrauch ist prinzipiell genauso zu berechnen, wie der Hauptspeicherverbrauch, allerdings auf Benutzerebene und nicht auf Prozeßebene, da Festplattenspeicher nicht an einen Prozeß gebunden ist und auch über die Ausführung eines Prozesses hinaus belegt werden kann. Die Festplattenspeicherbelegung kann ebenfalls über die Zeit variieren und könnte dann ebenso als Fläche unter der Funktion der Festplattenspeicherbelegung bezüglich der Zeit berechnet werden bzw. über die Näherung mittels periodischer Abfrage. Da aber Festplattenspeicher heutzutage extrem billig und in den meisten Systemen ausreichend vorhanden ist, wird in der Regel, wenn überhaupt eine Überwachung des Festplattenspeichers vorgenommen wird, ein fester Teil des Festplattenspeichers jedem Benutzer zugewiesen. Betriebssystemroutinen verhindern dann, daß der Benutzer mehr als den ihm zugeordneten Festplattenspeicher belegt (Quota). Dies funktioniert natürlich nur in Systemen mit festen, bekannten Benutzern. In Systemen mit unbekanntem Benutzern müssen dann doch, falls Accounting des Festplattenspeichers erwünscht ist, die Systemroutinen zur Festplattenbenutzung um Accountingmechanismen erweitert werden.

- Interne Kommunikationsressourcen:

Hierbei handelt es sich um Kommunikationsmittel innerhalb eines Computersystems, über die interne Komponenten kommunizieren, wie z.B. das Bussystem. Sie sind in der Regel ausreichend dimensioniert. Auf ein Accounting dieser Komponenten wird in den meisten Fällen verzichtet. Falls dies jedoch zwingend notwendig ist, müßten Routinen zur Überwachung dieser Ressourcen im Betriebssystem bzw. den Treibern der Komponenten integriert sein.

- Externe Kommunikationsressourcen:

Die wichtigste externe Kommunikationsressource ist das Netzwerk. Netzwerkverbindungen über feste Standorte hinaus werden, wie schon erwähnt, zumeist von Netzwerk-Diensteanbietern (Network Service Provider) realisiert. Da diese Verbindungen dann kostenpflichtig sind, ist hier ein Accounting zwingend notwendig. Zum einen auf Seite des Providers zur Gebührenberechnung und für statistische Zwecke (Auslastung), zum anderen auf Seiten des Netzwerkbenutzers zur Kostenkontrolle und ebenfalls für statistische Zwecke.

Das Accounting der Netzwerkzugriffe findet für einen einzigen Rechner auf demselben statt, für das Netzwerkaccounting von ganzen Subnetzen findet es in dem

Router statt, der die Verbindung zum externen Netzwerk realisiert. Dabei werden im Protokollstack die Netzwerkpakete mit Größe sowie Start- und Zieladresse mitprotokolliert. Ist eine Erweiterung des Protokollstacks nicht möglich, so muß untersucht werden, ob die Routinen, mit denen ein Netzwerkzugriff durchgeführt wird, so erweiterbar sind, daß ein Accounting der Netzwerkzugriffe realisiert werden kann.

Dienste:

Es gibt eine Vielzahl von Diensten, die in Computersystemen existieren. Zum Großteil werden sie von permanent laufenden Prozessen angeboten. Dabei handelt es sich unter anderem um Verzeichnisdienste und Datenbankzugriffsdienste. Das Accounting dieser Dienste sollte sinnvollerweise von den dienst anbietenden Prozessen selbst durchgeführt werden.

Beim Accounting ist generell zu beachten, daß der daraus gezogene Nutzen in einem vertretbaren Verhältnis zur Systembelastung durch das Accounting steht. Accounting macht keinen Sinn, wenn dadurch eine drastische Mehrbelastung der Systemperformance entsteht.

Billing:

Unter „Billing“ (engl. für Berechnen) versteht man im EDV-Bereich die Abrechnung erbrachter Dienstleistungen mit dem Dienstbenutzer. Der Besitzer der Ressourcen stellt dabei dem Benutzer die Leistungen in Rechnung. Die Daten über Häufigkeit der Benutzung von Ressourcen oder Diensten kommen dabei vom Accounting. Aus den Accountingdaten muß hierzu ein Preis für die erbrachten Dienste berechnet werden. Dieser Preis wird dem Benutzer dann, in welcher Form auch immer, in Rechnung gestellt.

Für die Durchführung des Billings existieren verschiedene Modelle, die sich unter anderem in den folgenden Aspekten unterscheiden:

- Abrechnungszeitpunkt: Wann erfolgt die Abrechnung der geleisteten Gebühren?
Gebühren können entweder am Ende der Inanspruchnahme der Dienste, periodisch in bestimmten Intervallen oder nach Ansammlung eines gewissen Betrags abgerechnet werden.
- Gebührenberechnung: Wie werden die Gebühren berechnet?
Es gibt viele Möglichkeiten, die Gebühren für geleistete Dienste zu berechnen. Man kann z.B. jede Inanspruchnahme eines Dienstes mit einer festen Gebühr abrechnen, man kann aber auch gewisse Pauschalen verwenden.

2.3 Zahlungssysteme

2.3.1 Einführung

Eng verbunden mit dem Boom des Internets in den letzten Jahren steigt auch die Anzahl der Anbieter im elektronischen Markt. Viele Experten sehen im Electronic Commerce die Einkaufs- und Vertriebsart der Zukunft. Der Haupt Gesichtspunkt, von dem die Akzeptanz der Benutzer abhängt, ist die Bequemlichkeit und der Komfort beim elektronischen Einkauf. Im Zuge der Steigerung dieser Aspekte kommt der Bedarf an einem elektronischen Zahlungssystem auf, mit dem im Internet direkt und bequem bezahlt werden kann. Angetrieben durch die Aussicht, in einigen Jahren Millionen von potentiellen Kunden bedienen zu können, entwickeln viele Firmen Zahlungssysteme, die ein bequemes und sicheres Bezahlen im Internet ermöglichen sollen. Wie so oft bei neuen Entwicklungen gibt es diverse Firmenallianzen und Einzelfirmen, die ihre eigenen Standards definieren.

Ein gutes Zahlungssystem muß unterschiedliche Ansprüche erfüllen. Es muß auf jeden Fall resistent gegenüber Manipulation sein. Desweiteren sollten die Kosten für eine Transaktion möglichst gering sein, ohne dabei aber wieder den Sicherheitsaspekt aus den Augen zu lassen. Oftmals ist es erwünscht, eine gewisse Anonymität zu wahren, ähnlich wie beim Bezahlen mit Bargeld. Dies läßt sich aber wiederum nur schwer mit dem Sicherheitsaspekt vereinbaren. Um alle Anwendungsfelder abzudecken, müssen die Systeme auch Transaktionen in beliebiger Höhe, also vom Zehntel eines Pfennigs bis zu tausenden von Mark, zulassen. Zu guter Letzt muß für die schon erwähnte Akzeptanz der Anwender die Einfachheit der Bedienung gewährleistet sein.

Oftmals kann ein Zahlungssystem nicht alle diese Bedingungen erfüllen. Im Folgenden werden die Unterschiede vorhandener Zahlungssysteme kategorisiert und es wird darauf eingegangen, welche Bedingungen sie erfüllen.

2.3.2 Kategorisierung der vorhandenen Zahlungssysteme

Kreditkartenzahlung über das Internet

Die Zahlung mit Kreditkarte ist die einfachste und naheliegendste Zahlungsweise im Internet. Im Vergleich zu gewohnten Kreditkartenzahlungen ändert sich lediglich das Medium, auf dem die Kreditkarteninformationen vom Kunden zum Händler übertragen werden. Der weitere Verlauf der Transaktion verläuft auf herkömmlichem Wege.

Anfänglich wurden die Kreditkarteninformationen unverschlüsselt übertragen, was aber ein automatisches Abhören der Daten sehr einfach macht. Inzwischen gibt es relativ sichere Übertragungsprotokolle für das Internet wie SSL (Secure Socket Layer) und S-HTTP (Secure Hypertext Transfer Protocol). Desweiteren arbeiten VISA und Mastercard an einem offenen Standard für sichere Kreditkartenzahlungen über unsichere Netzwerke mit dem Namen SET (Secure Electronic Transaction), das neben der sicheren Übertragung der Daten auch noch andere Möglichkeiten bietet, wie z.B. die Authentifizierung von Händlern und Kunden.

Zahlung über Kundenkonto

Bei dieser Art Zahlungssystem haben alle beteiligten Parteien (Händler und Kunden) ein Kundenkonto bei einer vermittelnden dritten Partei (Vermittler). Vor dem Einkauf muß sich der Kunde beim Einkaufssystem authentifizieren, z.B. über Kontonummer und Passwort. Die Transaktionen werden dann auf dem Kundenkonto akkumuliert und regelmäßig mit dem konventionellen Bankkonto des Kunden abgeglichen. Dabei muß natürlich ein fester Vertrag mit dem Vermittler bestehen. Dieses System hat den Vorteil, daß die Identität des Kunden nur dem Vermittler bekannt ist. Es muß jedoch ein Vertrauensverhältnis aller beteiligten Parteien gegenüber dem Vermittler bestehen. Das Verfahren benutzt kein gesetzliches Zahlungsmittel und muß deswegen auch nicht die gesetzlichen Anforderungen an ein solches erfüllen.

Digitales Bargeld:

Digitales Bargeld versucht, die Eigenschaften von konventionellem Bargeld auf den digitalen Bereich zu übertragen. Das heißt, daß Transaktionen mit digitalem Bargeld unabhängig von Banken oder Vermittlern getätigt werden können. Mindestens eine Bank muß jedoch die Währung decken, ausgeben und akzeptieren. Bei der Erzeugung des digitalen Bargeldes in Form von sogenannten Tokens wird jedes Token von der Bank mit einem geheimen Schlüssel signiert. Um das digitale Bargeld vor unerlaubtem Kopieren zu schützen, muß digitales Bargeld, nachdem man es bekommen hat, bei der Bank, die das Token ausgegeben hat, auf Gültigkeit geprüft werden. Diese überprüft dabei die Gültigkeit der Signatur. Danach folgt das sogenannte „clearing“. Die Bank, die das Token ausgegeben hat, tauscht es für den Einreicher gegen reelles Geld oder neues digitales Geld.

Bei Transaktionen mit Bargeld besteht nach dem Abschluß der Transaktion keine über das Bargeld nachvollziehbare Relation zwischen Kunde und Händler. Dieses wäre auch der Idealfall für digitales Bargeld, da viele Anwender bei Transaktionen über das Internet den Verlust ihrer Anonymität fürchten. Hierzu wurde das „Blinding“ entwickelt, ein Verfahren, das die Identität des Benutzers, der das Token von der Bank bekommen hat, für andere unerkennbar macht.

Die folgende Abbildung visualisiert noch einmal die drei Kategorien:

1. Kreditkartenzahlung
2. Zahlung über Kundenkonto
3. Digitales Bargeld

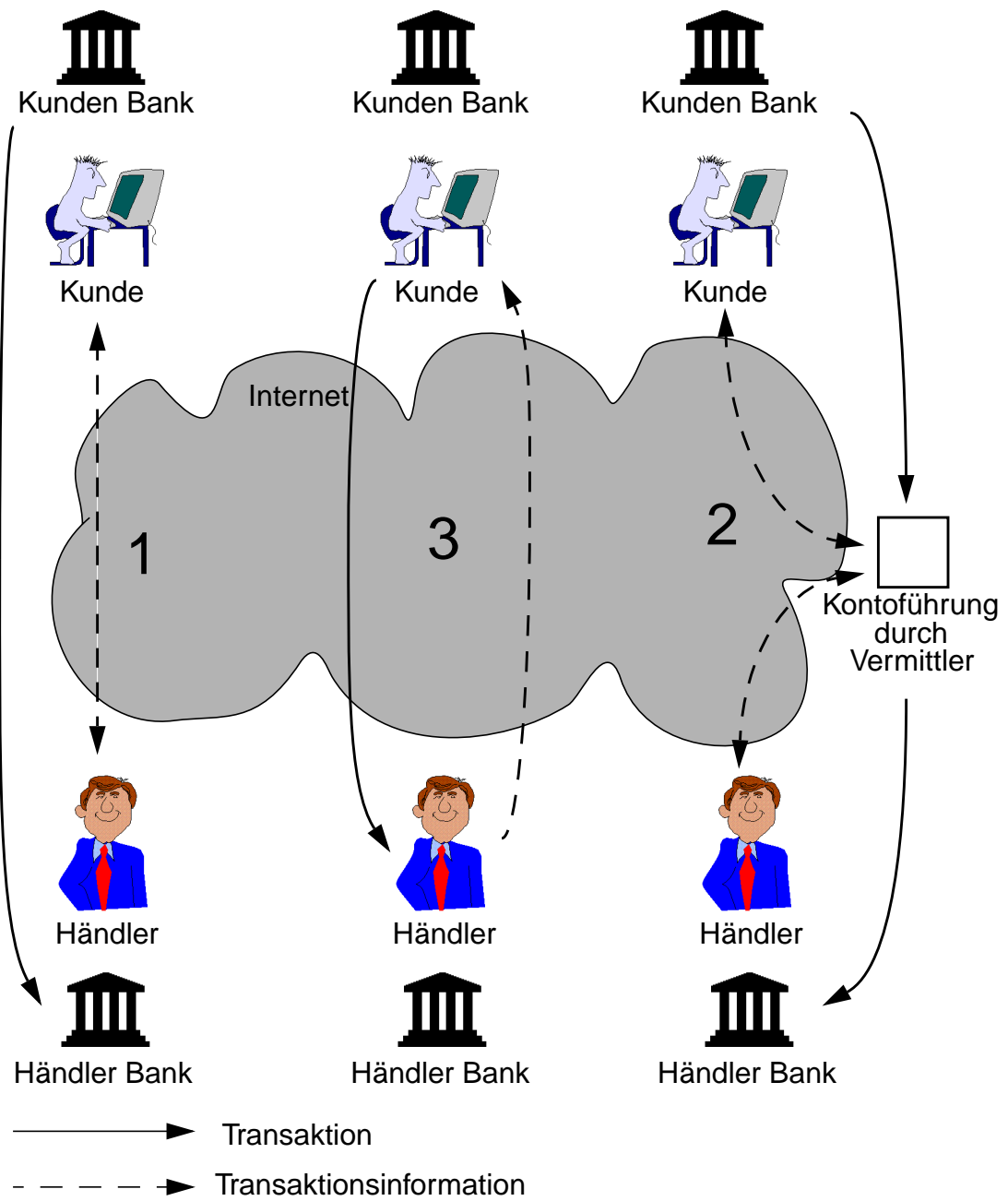


ABBILDUNG 3. Kategorisierung elektronischer Zahlungssysteme im Internet

2.3.3 Bewertung der Kategorien von Zahlungssystemen

Offene und geschlossene Marktplätze:

Während sich die Zahlung über Kundenkonto nur in geschlossenen Marktplätzen realisieren lässt, können die Zahlung per Kreditkarte und per digitalem Bargeld in beliebigen Kontexten genutzt werden.

Mikro- und Makrozahlungen:

Mikrozahlungen sind Zahlungen in der Größenordnung bis ca. 5 DM, Makrozahlungen sind Zahlungen ab ca. 5 DM.

Während Zahlungen über Kundenkonto für alle Zahlungsbeträge rentabel sind, lohnen sich Kreditkartenzahlungen aufgrund der hohen Gebühren nur für Makrozahlungen. Digitales Bargeld eignet sich hingegen sehr gut für Mikrozahlungen, dafür nur beschränkt für Makrozahlungen.

2.3.4 Rechtliche Aspekte

Die Sicherheit der Zahlungssysteme hängt entscheidend von den verwendeten kryptographischen Verfahren ab. In vielen Ländern ist jedoch die Verwendung von starker Kryptographie untersagt oder unterliegt gewissen Exportbeschränkungen. Deswegen sind die bisher zur Verfügung stehenden Systeme nicht optimal gegen Mißbrauch geschützt. Es gibt jedoch Bestrebungen, hier geeignete Kompromißlösungen zu finden.

2.3.5 Existierende Zahlungssysteme in Deutschland

Im folgenden werden zwei Zahlungssysteme vorgestellt, die sich in Deutschland in der Testphase bzw. schon im Einsatz befinden.

CyberCash:

Ein Zahlungssystem der gleichnamigen Firma, das von vielen deutschen Banken, u.a. Commerzbank, Dresdner Bank und Landesbank Baden-Württemberg, angeboten wird.

CyberCash enthält eine Zahlungsvariante über Kreditkarte, die dem SET-Standard angepaßt werden soll und eine Variante für Mikrozahlungen, die CyberCoin. Dabei handelt es sich, etwas widersprüchlich zum Namen, nicht um ein „token-based“, sondern um ein Konto-basiertes („account based“) System. Die CyberCoins existieren nur als Betrag auf einem speziellen CyberCoin-Konto. Bei Transaktionen mit CyberCoin wird der Betrag dann von diesem Konto auf ein anderes verschoben. Dies gibt dem Kunden Anonymität gegenüber dem Händler, aber nicht gegenüber der Bank. CyberCoin wurde entwickelt für Zahlungen von 0.25 USD bis 10.00 USD.

eCash:

Bei eCash handelt es sich um digitales Geld („token based“) der Firma DigiCash. Angeboten wird es in Deutschland von der Deutschen Bank. Ecash erfüllt die typischen Eigenschaften von digitalem Bargeld. Das digitale eCash Bargeld besteht aus verschlüsselten Tokens, die durch das „Blinding“-Verfahren anonymisiert werden. Der Schutz gegen Fälschung bzw. Münzkopierung wird über eine „Double-Spending“-Datenbank realisiert, d.h., daß empfangene Geld-Tokens an die ausgebende Bank zur Überprüfung überstellt werden müssen. Mikrozahlungen sind möglich, für Makrozahlungen ist das System eher ungeeignet. Für letztere bietet die Deutsche Bank ein SET-kompatibles Kreditkartenzahlungssystem.

3 Entwurf

In diesem Kapitel wird ein Entwurf einer Abrechnungskomponente für ein Mobile-Agenten-System erarbeitet. Dieser umfaßt eine Komponente zur Erfassung der Dienstleistungen (Accounting), eine Komponente zur Berechnung der Gebühren (Billing) und die Integration einer Zahlungskomponente in ein Mobile-Agenten-System.

Da der Entwurf dieser Komponenten stark von der Umgebung, in der die Komponenten integriert werden, abhängt, wird er im Folgenden für ein Agentensystem durchgeführt, das in der Sprache Java, Version 1.1 implementiert ist.

Die folgende Graphik zeigt die Interaktion der einzelnen Komponenten untereinander.

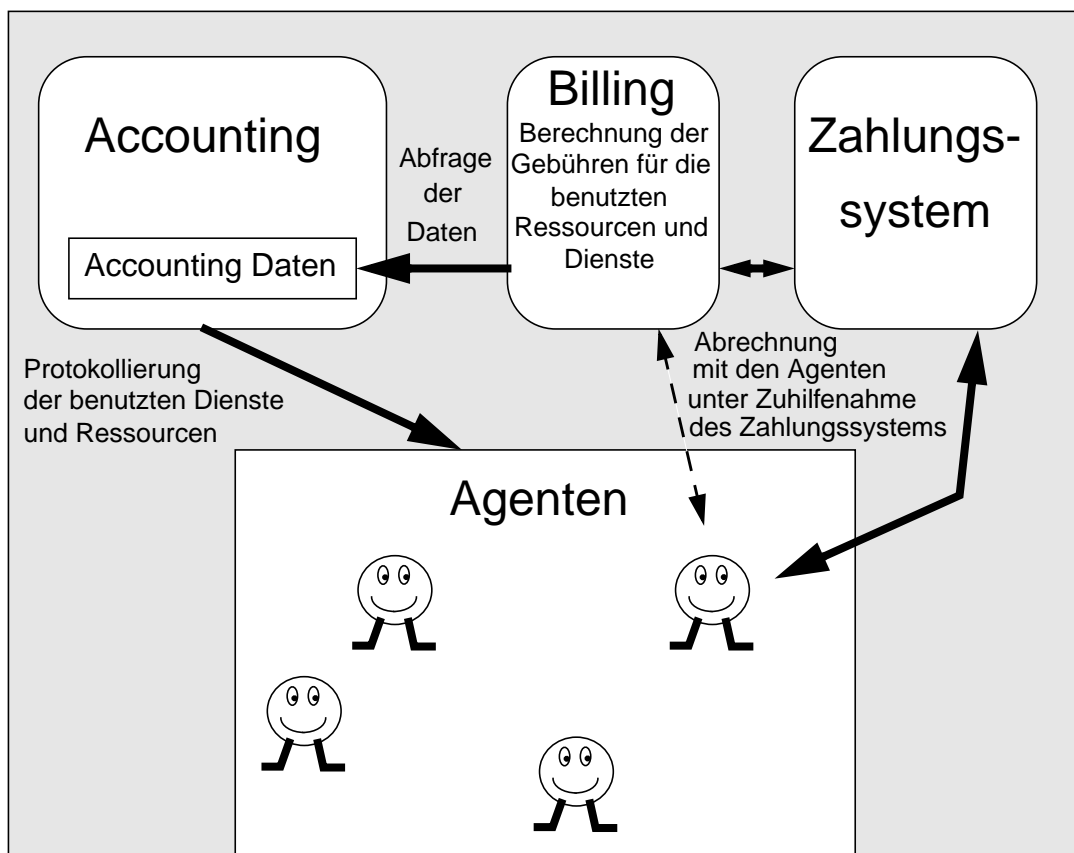


ABBILDUNG 4. Komponenten eines Abrechnungssystems für Mobile-Agenten-Systeme

Die Accounting-Komponente sammelt Daten über die Agenten und speichert sie zentral in einer geeigneten Datenstruktur. Die Billing-Komponente berechnet aus den Accounting-Daten die Gebühren für die einzelnen Agenten und rechnet mit diesen unter Zuhilfenahme des Zahlungssystems ab.

3.1 Komponente zur Erfassung der Dienstleistungen (Accounting)

Die Accounting-Komponente sammelt Daten über verbrauchte Ressourcen und in Anspruch genommene Dienste und speichert diese Daten in einer geeigneten Datenstruktur ab. Dabei beschränkt sie sich auf das Accounting von Agenten, da andere Informationen nicht benötigt werden. Das Accounting dient dem späteren Abrechnen der verbrauchten Ressourcen und benutzten Dienste.

Accounting benötigt grundsätzlich Rechenzeit, d.h. es verlangsamt den Ablauf der restlichen Komponenten. Es hängt dabei von der Effizienz der einzelnen Accounting-mechanismen ab, in welchem Maß das System gebremst wird. Generell ist darauf zu achten, daß der Aufwand des Accountings in einem vernünftigen Verhältnis zum Nutzen steht. Da nicht alle Betreiber eines Agentensystems Wert auf das Accounting derselben Ressourcen bzw. auf die Genauigkeit des Accountings legen, sollte die Granularität des Accountings konfigurierbar sein, d.h., daß für alle überwachten Ressourcen die Feinheit des Accounting einstellbar bzw. das Accounting einzelner Ressourcen ganz ausschaltbar sein sollte.

3.1.1 Überwachte Ressourcen

Ein direktes Accounting aller Ressourcen in Java 1.1 ist grundsätzlich ohne Modifikationen an der virtuellen Java Maschine nicht möglich. Diese Modifikationen an der virtuellen Maschine sind zwar möglich, widersprechen aber dem Gedanken der Plattformunabhängigkeit von Java. Das System müßte mit einer eigenen virtuellen Maschine für jedes zu verwendende Betriebssystem ausgeliefert werden. Dieser Aufwand ist kaum zu bewältigen, deshalb wird das Accounting auf diejenigen Ressourcen beschränkt, deren Überwachung machbar ist. Im Folgenden wird die Machbarkeit des Accounting in Java 1.1 für diverse Ressourcen diskutiert.

- CPU-Zeit:

Die Rechenzeit eines Threads ist in Java 1.1 nicht grundsätzlich abfragbar. Daraus folgt, daß auch die verbrauchte Rechenzeit eines Agenten nicht direkt bestimmbar ist.

Es gibt jedoch eine Möglichkeit, trotzdem an den CPU-Verbrauch eines Threads zu gelangen, nämlich wenn das Agentensystem über einen eigenen Scheduler verfügt. Ein Scheduler ist eine Komponente, die für die Vergabe von Rechenzeit an die einzelnen Threads zuständig ist. Wenn das Agentensystem also die Vergabe der Rechenzeit für einen Teil der Threads selbst vornimmt, können auch Mechanismen integriert werden, die die vergebene Rechenzeit an diese Threads mitprotokollieren. Threads, die ihre Rechenzeit nicht vom Scheduler des Mobile-Agenten-Systems bekommen, sondern direkt vom Scheduler des Laufzeitsystems, bleiben hier jedoch außen vor. Diese Threads sind aber in der Regel Systemthreads, deren Ausschluß vom Accounting nicht weiter tragisch ist.

Bei dieser Lösung ist zu beachten, daß der Scheduler nicht den Agenten Rechenzeit direkt zuweist, sondern allen vorhandenen Threads. Ein Agent besteht in der Regel aus mehreren Threads. Zum Beispiel wird bei jedem Ankommen einer Nachricht

(Message) im Agentensystem Mole ein neuer Thread gestartet, der die Verarbeitung der Nachricht übernimmt. Desweiteren gibt es Systemthreads, die zu keinem bestimmten Agenten gehören. Man muß also sicherstellen, daß jeder Thread, der zu einem bestimmten Agenten gehört, diesem auch zugeordnet werden kann. Wenn diese Voraussetzung erfüllt ist, läßt sich auf diesem Wege dann auch die Rechenzeit, die jeder einzelne Agent verbraucht, mitprotokollieren.

Es gibt außerdem noch die Möglichkeit, durch Modifikation von Java-Klassen (sogenanntes Byte Code Rewriting, das entweder online beim Laden einer Klasse durchgeführt wird oder offline die Klassenbibliothek permanent verändert) und Aufrufe eigener nativer Methoden, erweiterte Funktionalitäten in die Klassen, die Threads erzeugen, zu integrieren und auf diesem Wege unter anderem die CPU-Zeiten vom Betriebssystem abzufragen. Dabei wird bei jeder Erzeugung eines Threads (z.B. beim Aufruf der *run()* Methode) eine native Methode gestartet, die eine Verbindung des Java-Threads zu einem Betriebssystem-Thread herstellt, über die dann die entsprechenden CPU-Zeiten vom Betriebssystem abgefragt werden können. Dieser Ansatz ist jedoch sehr aufwendig. Desweiteren verliert man aufgrund der Notwendigkeit nativer Methoden zur CPU-Zeit-Abfrage vom Betriebssystem die Plattformunabhängigkeit der Implementierungssprache Java.

- Speicherverbrauch:

Wie in der Einführung beschrieben, gibt es prinzipiell zwei Möglichkeiten, den Speicherverbrauch eines Prozesses zu ermitteln.

Die erste Methode ist die exakte Berechnung. Hierzu müssen in den Systemfunktionen zur Speichervergabe und zur Speicherfreigabe Accountingmechanismen integriert werden. Es gibt auch hier die Möglichkeit mittels Byte-Code-Rewriting Mechanismen zu integrieren, die vor jeder Objekt-Allokations-Instruktion ausgeführt werden und den neu angeforderten Speicher zum belegten Speicher hinzuzudieren. Ebenso können Methoden aufgerufen werden, bevor der Garbage-Collector oder andere Mechanismen Speicher wieder frei geben. Die Methoden subtrahieren dann den freigegebenen Speicher vom belegten Speicher. Diese Möglichkeit ist jedoch sehr aufwendig zu implementieren und verlangt genaue Kenntnisse über den Java-Byte-Code. Desweiteren läuft das Agentensystem später dann nur auf entsprechend angepaßten Java-Laufzeitsystemen. Da dies nicht erwünscht ist, wird von dieser Realisierung Abstand genommen.

Die zweite Methode beruht auf dem periodischen Abfragen des belegten Speichers eines Prozesses bzw. in unserem Falle eines Agenten. Hierzu muß eine Methode existieren, den belegten Speicher eines Objektes zu beliebigen Zeiten zu bestimmen. Java 1.1 stellt keine Methode zur Verfügung, die Größe des belegten Speichers eines Objektes festzustellen. Man kann jedoch auf Umwegen den belegten Speicher des Objektes weitestgehend bestimmen.

Dies funktioniert über die sogenannte Serialisierung. In Java ist es möglich, ein komplettes Objekt mit allen Werten in einen Bytestrom zu schreiben. Ein so „serialisiertes“ Objekt kann zu jeder Zeit wieder eingelesen werden und ist mit dem Original-Objekt strukturell identisch. Wenn man nun die Länge des Bytestromes, in dem sich das serialisierte Objekt befindet, bestimmt, was ohne weiteres möglich ist, so bekommt man eine gute Näherung, wieviel Speicher ein Objekt belegt.

Wir haben also nun eine Möglichkeit, die Größe des belegten Speichers eines Objektes ungefähr zu berechnen. Um damit den Speicherverbrauch zu bestimmen, muß man die Messung des belegten Speichers periodisch durchführen. Je kürzer die Periode ist, desto genauer ist die Berechnung. Das Problem dabei ist, daß die Serialisierung eines Objektes relativ aufwendig ist, d.h., daß eine kleine Meßperiode das System stark belasten würde. Hier muß ein geeigneter Mittelweg gefunden werden. Am einfachsten löst man dies, indem man die Periodenlänge konfigurierbar macht. Auf diese Weise kann man sich später für eine geeignete Länge entscheiden. Im Minimalfall wird die Messung des Agenten nur beim Betreten und Verlassen einer Location durchgeführt. Dies ist zwar die ungenaueste Näherung des Speicherverbrauchs eines Agenten, liefert aber immer noch eine brauchbare Näherung.

Die folgende Graphik veranschaulicht dies noch einmal. Im linken Schaubild wurde eine sehr kleine Periode gewählt. Man sieht, daß die schraffierte Fläche relativ genau der Fläche unterhalb der Kurve entspricht. Im rechten Schaubild wird der belegte Speicher nur an zwei Stellen bestimmt. Die Näherung ist nicht so genau, wie die bei kurzer Periodenlänge, gibt aber auch ein ungefähres Bild über den verbrauchten Speicher wieder.

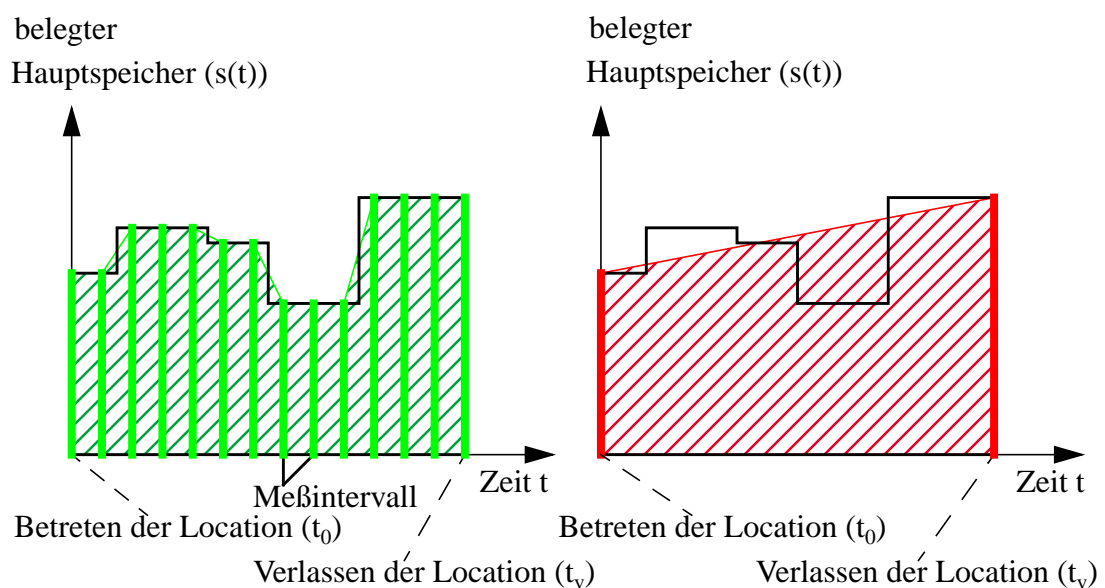


ABBILDUNG 5. Näherungen des Speicherverbrauchs

- Netzwerkkommunikation:

Auch für das Accounting der Netzwerkzugriffe bietet Java 1.1 keine eigenen Sprachelemente. Für eine allgemeine Lösung müßte wiederum die Java-Laufzeitumgebung mittels Byte Code Rewriting so modifiziert werden, daß die Methoden, die direkt aufs Netzwerk zugreifen, Accounting-Mechanismen aufrufen. Diese Lösung fällt aufgrund der schon vorher erwähnten Gründe weg.

Die Netzwerkkommunikation von Agenten beschränkt sich aber zum Großteil auf Kommunikationsmechanismen, die das Agentensystem den Agenten zur Verfügung stellt, wie z.B. das Versenden von Nachrichten. Diese Kommunikationsmechanismen können problemlos um Accountingmechanismen erweitert werden. Lediglich die direkten Netzwerkzugriffe, sofern sie den Agenten überhaupt gestattet sind, bleiben vom Accounting ausgeschlossen. Der Umfang der mitprotokollierten Daten sollte vom Administrator des Systems konfigurierbar sein. Dabei können folgende Informationen, falls gewünscht, mitprotokolliert werden:

- Anzahl der Netzwerkzugriffe
 - Größe der gesendeten / empfangenen Daten (mittels Serialisierung)
 - Netzwerkstart- und Zieladresse
- Dienste:

Das Ausführen von Diensten durch die Location oder ortsfeste Agenten kann leicht mitprotokolliert werden, wenn die Location oder die dienstbringenden Agenten die Inanspruchnahme ihrer Dienste direkt an die Accountingkomponente melden. Die Accountingkomponente speichert diese dann ebenfalls in der zentralen Datenerhaltung.

Zusammenfassung:

Das Accounting kann für erbrachte Dienstleistungen, die Ressourcen CPU-Zeit, Hauptspeicher und einem Großteil der Netzwerkzugriffe durchgeführt werden.

3.1.2 Überwachte Agenten

Das Accounting, das in erster Linie der Gewinnung von Daten zur Berechnung von Gebühren gilt, beschränkt sich ausschließlich auf von Agenten benutzte Ressourcen und Dienstleistungen.

Generell gibt es zwei Klassen von Agenten in Mobile-Agenten-Systemen. Die ortsfesten Systemagenten und die mobilen Benutzeragenten. Es stellt sich hierbei die Frage, ob nur das Accounting von Benutzeragenten notwendig ist oder auch Systemagenten accountet werden sollen.

Es gibt Szenarien, in denen die Systemagenten nur von den Systemverwaltern ins System eingebracht werden. In diesem Fall ist ein Accounting und ebenfalls ein Billing dieser Agenten überflüssig. Andererseits kann man sich auch vorstellen, daß auf Locations fremde Systemagenten Dienste anbieten, die auf die Privilegien von Systemagenten angewiesen sind. In diesem Fall macht ein Accounting der Systemagenten wieder Sinn.

Um beiden Fällen gerecht zu werden und so viel überflüssiges Accounting wie möglich zu vermeiden, sollte das Accounting von Systemagenten über die Konfiguration festlegbar sein.

3.1.3 Datenhaltung der Accounting-Daten

Die beim Accounting gesammelten Daten über den Verbrauch von Ressourcen müssen an einer zentralen Stelle im System in einer geeigneten Datenstruktur gespeichert werden. Am sinnvollsten ist es, wenn sich jede Location ein Datenobjekt für die Accountingdaten hält, da man die Locations, auch wenn sie sich eventuell auf demselben Rechner befinden, als Administrationseinheiten betrachten kann. Sie werden vom Systemadministrator gewartet und installiert. Desweiteren sind sie auch ein guter Ausgangspunkt für die spätere Berechnung der Gebühren für geleistete Dienste gegenüber den Agenten. Diese Datenhaltung gewährleistet, geeignete Kapselung vorausgesetzt, die Kontrolle der Zugriffe auf diese Daten durch andere Objekte, wie etwa die Agenten. Ein Agent sollte, wenn überhaupt, nur auf seine eigenen Accountingdaten zugreifen können. Dies geschieht dann durch geeignete Zugriffsmechanismen. Die Location sollte für jeden Agenten einen Eintrag in der Accountingdatenstruktur anlegen - sinnvollerweise, wenn der Agent die Location betritt. Dieser Eintrag kann dann wieder gelöscht werden, wenn der Agent die Location verläßt und seine Gebühren abgerechnet sind. Die Datenstruktur sollte für alle protokollierbaren Werte Felder besitzen. Sollte der Umfang des Accountings so eingestellt sein, daß nur wenige Informationen gesammelt werden, bleiben diverse Felder unbenutzt. Die Größe einiger Felder, wie z.B. das Feld für den belegten Speicher zu bestimmten Zeiten, muß dynamisch vergrößerbar sein, da die Anzahl der Speichermessungen vorher nicht bestimmbar ist.

Die folgende Graühik zeigt die Datenhaltung der Accountingdaten. Die Struktur ist hierarchisch aufgebaut. Ein übergeordnetes Objekt enthält für jeden Agenten ein anderes Objekt, welches die Accountingdaten aufnimmt. Einige Einträge dieser Objekte verweisen wieder auf andere Objekte, die beliebig viele Objekte einer weiteren Klasse enthalten.

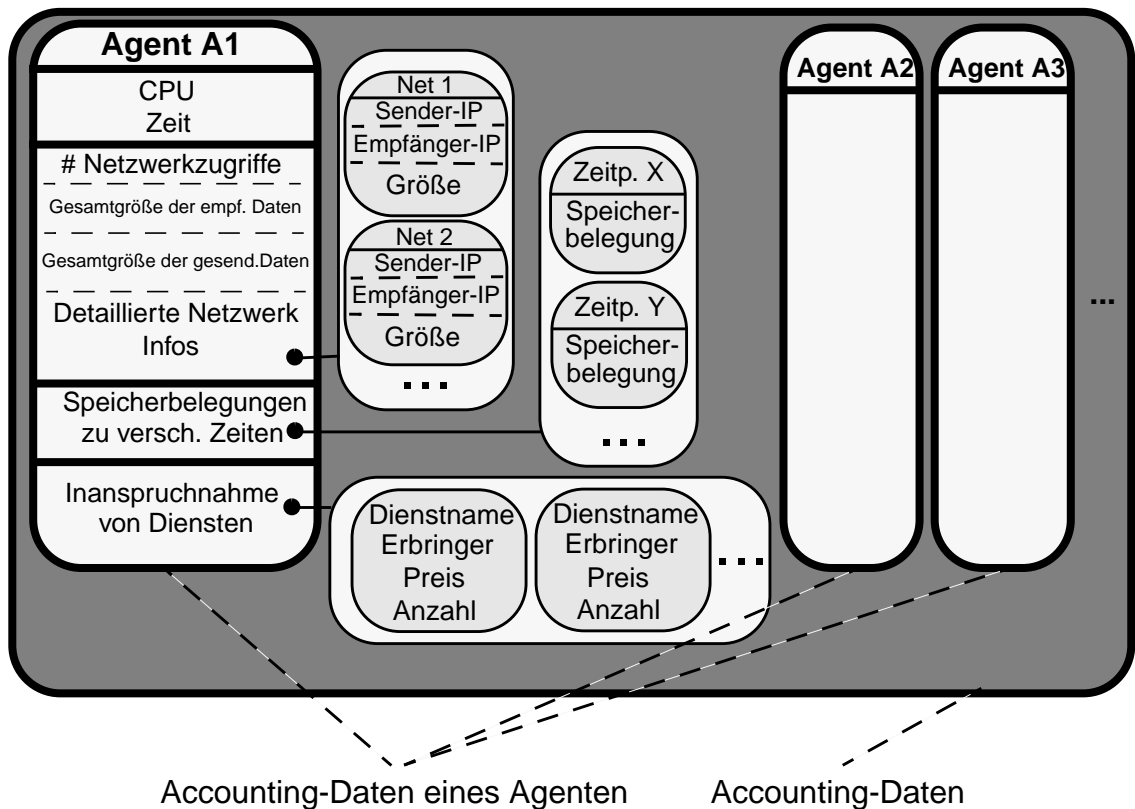


ABBILDUNG 6. Entwurf der Accounting-Datenstruktur

3.2 Komponente zur Berechnung der Gebühren

Die Berechnung der Gebühren, die die Agenten an die Locations zu entrichten haben, basiert auf den vom Accounting gelieferten Daten. Dabei sind alle vom Accounting erfassbaren Ressourcen vom Gebährensystm auch abrechenbar. Die Gebährensstruktur einer Location sollte von überall durch die Agenten abfragbar sein, also auch vor der Migration dorthin. Vor Ort sollte ein Agent jederzeit die von ihm verursachten Gebähren bei der Location abfragen können.

3.2.1 Gebährensrechnung

Die Methoden zur Berechnung der durch die Agenten verursachten Gebährenbeträge können folgendermaßen aussehen:

1. Der Betreiber einer Location kann eine eigene Methode zur Gebährensrechnung zur Verfügung stellen, die die Gebähren für einen Agenten anhand der ihr übergebenen Accountingdaten dieses Agenten nach beliebigen Gesichtspunkten berechnet.

Diese Methode bietet ein Maximum an Flexibilität. Sie läßt den Betreiber einer Location die Gebühren in beliebiger Form festsetzen. Problematisch ist dabei, daß die Freiheit, die bei der Gebührenberechnung herrscht, es erschwert, die Preisstruktur einer Location den interessierten Agenten einheitlich zu präsentieren.

2. Das System stellt festintegrierte Routinen zur Gebührenberechnung zur Verfügung. Diese Routinen sollten dann von außen flexibel konfiguriert werden können. Diese Methode gestattet eine schnelle und einfache Konfiguration der Gebührenstruktur der Location. Es ist keinerlei Programmieraufwand nötig, es müssen lediglich die konfigurierbaren Parameter gesetzt werden. Bei dieser Methode ist eine einheitliche Präsentation der Preisstruktur der Location gegenüber den Agenten möglich. Die Vielfältigkeit ist aber bei weitem nicht so groß wie bei der ersten Methode.

Werden die systemeigenen Routinen zur Gebührenberechnung verwendet, so könnten die Gebühren für die einzelnen Ressourcen folgendermaßen festgesetzt werden:

- CPU:

Es kann entweder ein Betrag festgelegt werden, der pro Millisekunde CPU-Zeit an Gebühren anfällt oder es können CPU-Zeit Intervalle mit den entsprechenden Kosten definiert werden (z.B. 0 - 1000 ms: 10 Geldeinheiten, 1001 - 10000 ms: 20 Geldeinheiten, etc.).

- Speicher:

Die Gebühren für den Speicherverbrauch (in Byte * Millisekunde) können ebenfalls entweder mit einem festen Multiplikator berechnet werden oder für bestimmte Intervalle vordefiniert werden.

- Netzwerkbenutzung:

Die Gebühren für Netzwerkbenutzung können vielfältig festgesetzt werden.

Zuerst einmal kann man je nach IP-Adresse des Kommunikationspartners unterscheiden in lokale Kommunikation oder Kommunikation mit einem bestimmten Subnetz. Je nach Subnetz bzw. lokaler Kommunikation, kann man dann unterschiedliche Gebühren festsetzen.

Die Gebühren werden entweder pro Anzahl der Netzwerkzugriffe oder pro übertragenes Byte festgesetzt. Wiederum entweder mit einem konstanten Kostenfaktor oder in vordefinierten Intervallen.

Beispiel:

lokale Nachrichten: Anzahl: 0-50 kostenlos, 50- ~ 10 Geldeinheiten

Subnetz 1.0.0.0/8: 1 Geldeinheit pro Nachricht

Subnetz 0.0.0.0/0: 0.5 Geldeinheiten pro Byte

(Die Subnetzbezeichnung hier lautet <Netzwerkadresse>/<dezimale Netzwerkmaske>)

- Dienstleistungen (services):

Der Preis für die einzelnen von Agenten erbrachten Dienstleistungen wird von den Erbringern selbst festgelegt und der Location bei jedem erbrachten Dienst gemeldet.

3.2.2 Abrechnungszeitpunkt

Der Zeitpunkt an dem die Gebühren abgerechnet werden, kann auf verschiedene Weisen festgesetzt werden:

- periodisch:

Die Abrechnung erfolgt in festen Intervallen. Nach Ablauf einer konfigurierbaren Zeit wird mit jedem Agenten nacheinander oder parallel abgerechnet.

- nach Ansammlung einer bestimmten Summe:

Sollte ein Agent eine frei konfigurierbare Summe an Gebühren verursacht haben, so wird die Abrechnung mit ihm angestoßen. Dieses Verfahren kann sehr aufwendig werden, da nach jeder Benutzung einer Ressource durch einen Agenten seine zu zahlenden Gebühren neu berechnet werden müssen, um zu überprüfen, ob die konfigurierte Summe überschritten worden ist. Bei dieser Variante gibt es auch noch die Möglichkeit, daß die konfigurierte Summe von den Agenten im Voraus gezahlt werden muß. Nach dem Verbrauch des Geldes muß dann eine erneute Gebührenzahlung vom Agenten gefordert werden.

- bei Verlassen der Location:

Es wird mit dem Agenten abgerechnet, wenn er die Location verläßt. D.h. entweder wenn er auf eine andere Location wechselt oder wenn er terminiert.

In den ersten beiden Fällen muß der fällige Restbetrag der Agenten bei Verlassen der Location ebenfalls abgerechnet werden. Dabei kann es bei der zweiten Variante auch sein, daß der Restbetrag negativ ist, d.h., daß die Location dem Agenten die zuviel bezahlten Gebühren rückerstatten muß.

Welche der drei Varianten am geeignetsten ist, hängt vom Kontext ab. Am vorteilhaftesten wäre es, wenn alle Varianten möglich sind und die im vorhanden Kontext sinnvollste mittels Konfiguration ausgewählt werden kann.

3.2.3 Zahlungsunfähigkeit

Nach Empfangen einer Zahlungsaufforderung von der Location hat ein Agent eine frei definierbare Zeitspanne zur Verfügung, um seine Schulden zu begleichen. Sollte der Agent nach Ablauf der Zeitspanne die Gebühren nicht bezahlt haben, so gibt es zwei Möglichkeiten, wie die Location mit ihm verfahren kann:

- sofortige Terminierung des Agenten:
Der Agent wird vom Agentensystem unwiderruflich beendet.
- Zwangsmigration zur Location, auf der er gestartet wurde:
Falls es möglich ist, die Location zu bestimmen, auf der der Agent erstmals gestartet wurde, wird eine Migration zu dieser Location durchgeführt.

In beiden Fällen sollte ein ausführlicher Eintrag in eine Protokolldatei gemacht werden. Dieser Eintrag beinhaltet, welche Dienste der Agent in Anspruch genommen hat, welchen Betrag er der Location schuldet und auch alle verfügbaren Daten über den Agenten selbst. Hiermit kann dann eventuell später ein Schadensanspruch geltend gemacht werden.

Um mutwillige Zahlungsunfähigkeit zu unterbinden, kann eine Location von den Agenten eine bestimmte Summe an Geld im voraus verlangen, eine Art Pfand. Beim Verlassen der Location bekommen die Agenten das zuviel bezahlte Geld wieder zurückerstattet.

3.3 Zahlungskomponente

Die Zahlungskomponente soll es den Agenten und Locations im Agentensystem ermöglichen, finanzielle Transaktionen zu tätigen.

Dabei sollte ein geeignetes Zahlungssystem folgende Kriterien unterstützen:

- Offener Marktplatz:
Das Agentensystem soll kein eigenes proprietäres Zahlungssystem beinhalten, d.h. es soll kein Zahlungssystem speziell für das Agentensystem implementiert werden. Es sollte vielmehr auf ein standardisiertes Zahlungssystem zurückgegriffen werden, damit es den Agenten auch möglich ist, mit anderen Objekten im Internet, wie z.B. elektronischen Läden o.ä., finanzielle Transaktionen zu tätigen.
Auch das Zahlen per Kundenkonto, das in vielen virtuellen Einkaufszentren üblich ist, basiert auf proprietären Protokollen, die nicht für einen offenen Marktplatz geeignet sind. Diese Kategorie an Zahlungssystemen ist somit für unsere Zwecke ebenfalls nicht geeignet.
- Mikrozahlungen:
Die Gebühren, die ein Agent an die Location zu entrichten hat, sind in vielen Fällen eher kleinere Beträge, besonders wenn der Agent nicht lange auf der Location verweilt. Ein geeignetes Zahlungssystem sollte deshalb hauptsächlich für Mikrozahlungen geeignet sein.
- Anonymität:
Tätigt ein Agent für seinen Besitzer Einkäufe im Internet, so möchte der Besitzer seine Identität oftmals nicht preisgeben. Ein Zahlungssystem, das anonyme Zahlun-

gen ermöglicht, wäre eher geeignet, als eines, das diese Möglichkeit nicht bietet, wobei das Fehlen dieser Eigenschaft keinesfalls ein Ausschlußkriterium für ein Zahlungssystem ist.

Fazit:

Vergleichen wir die Eigenschaften der in Kapitel 2.3 vorgestellten Kategorien von Zahlungssystemen, so läßt sich daraus schließen, daß ein Zahlungssystem, das auf digitalem Bargeld basiert (z.B. das vorgestellte eCash-System der Firma DigiCash), alle obigen Kriterien erfüllt und somit am geeignetsten scheint.

Diese Diplomarbeit legt sich bewußt auf kein derzeit auf dem Markt befindliches Zahlungssystem fest. Viele Zahlungssysteme im Internet sind schnell wieder vom Markt verschwunden. Deswegen scheint es zum jetzigen Zeitpunkt besser zu sein abzuwarten, bis sich ein geeignetes Zahlungssystem am Markt etabliert.

Es soll vielmehr eine offene Schnittstelle zwischen Agentensystem und Zahlungssystem geschaffen werden, die eine leichte Integration eines beliebigen Zahlungssystems in das Agentensystem ermöglicht. Desweiteren sollte sich die Programmierung der am Zahlungsverkehr teilnehmenden Agenten nicht viel aufwendiger gestalten, als die Programmierung der Agenten seither.

Um dies zu erreichen ist eine Klasse in das Agentensystem integriert worden, die den Umgang mit dem Zahlungssystem für die Agenten und Locations vereinfacht und vereinheitlicht. Jeder Agent und jede Location instanziiert sich ein Objekt dieser Klasse. Dieses Objekt stellt den Agenten und Locations feste und standardisierte Funktionalitäten zur Verfügung und setzt diese dann in die proprietären Protokolle und Funktionen des Zahlungssystems um. Soll dann ein anderes Zahlungssystem an das Agentensystem angepaßt werden, muß nur diese Schnittstellenobjekt-Klasse neu implementiert werden. Die vorhanden Agenten funktionieren weiterhin, da die Schnittstelle zu den Agenten gleich geblieben ist.

Das Schnittstellenobjekt enthält je nach verwendetem Zahlungssystem dann z.B. auch Informationen über die Wallet seines Besitzers (darunter versteht man die virtuelle Geldbörse bei Zahlungssystemen mit digitalem Bargeld, in der sich die Geldtokens befinden), Kreditkartendaten (bei Zahlungssystemen über Kreditkarte) oder sonstige vom Zahlungssystem benötigte Informationen. Diese müssen bei der Initialisierung des Schnittstellenobjektes demselben bekannt gemacht werden. Aufgrund des Besitzes dieser sensiblen Daten muß das Schnittstellenobjekt ausreichend gegen fremde Zugriffe und Manipulationen gesichert sein.

Die folgende Graphik zeigt die Integration des Zahlungssystems mittels der Schnittstellenobjekte.

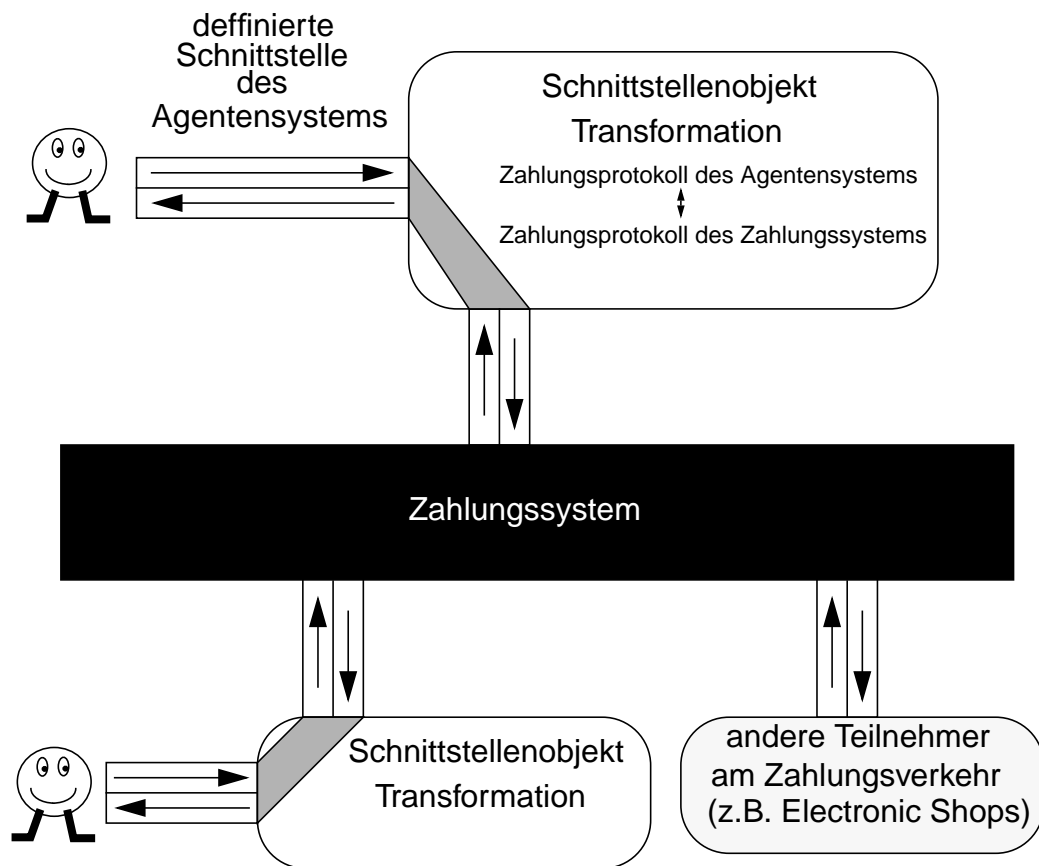


ABBILDUNG 7. Integration des Zahlungssystems

Mittels dieses Ansatzes ist es auch möglich, die Integration mehrerer Zahlungssysteme zu realisieren. Dies kann aus diversen Gründen nützlich sein. Man kann sich z.B. vorstellen, daß ein Zahlungssystem eher für Mikrozahlungen, ein anderes eher für Makrozahlungen geeignet ist. Die Benutzung der Zahlungssysteme durch den Agenten bleibt unverändert. Er bedient sich weiterhin der definierten Schnittstellen. Das Objekt, das die Schnittstelle realisiert, kann dann, z.B. auf grund der Höhe der Zahlung, das geeignetere Zahlungssystem auswählen und verwenden. Diese Realisierung ist außerdem auch vorteilhaft, wenn ein Teilnehmer am Zahlungsverkehr (kein Agent) nur ein bestimmtes Zahlungsmittel akzeptiert. Das Schnittstellenobjekt wählt dann das Zahlungssystem aus, das von allen an der Transaktion beteiligten Parteien unterstützt wird.

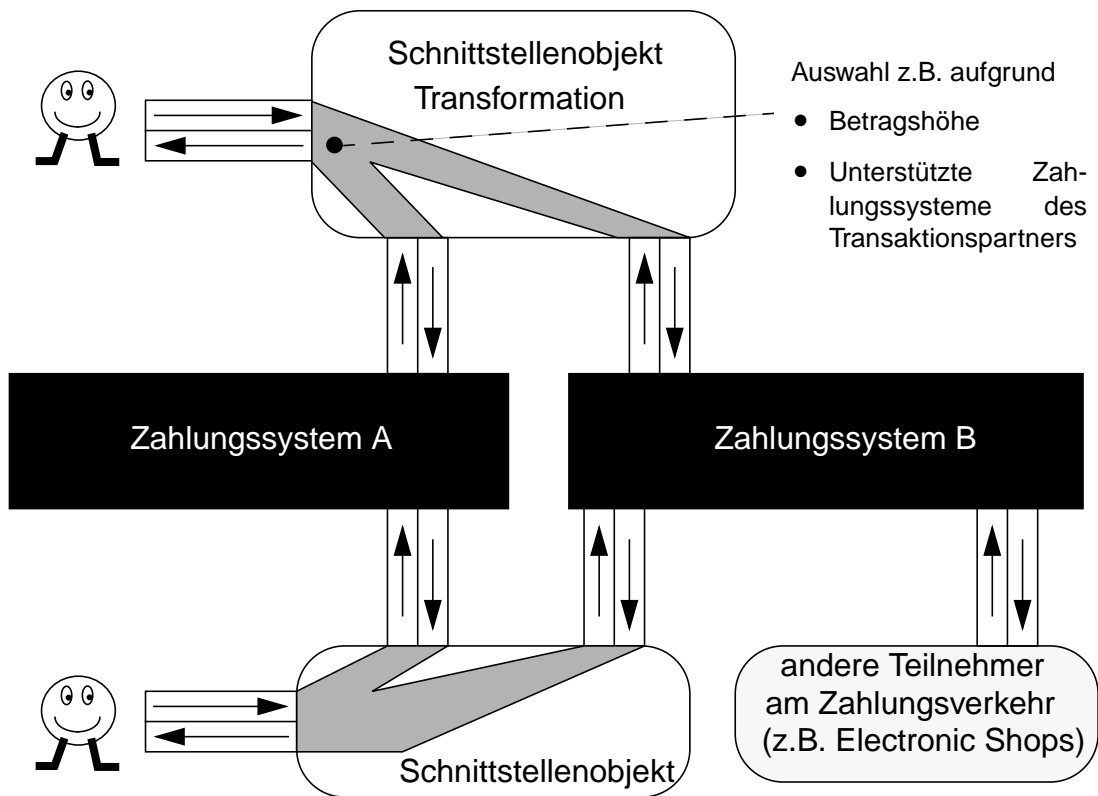


ABBILDUNG 8. Integration mehrerer Zahlungssysteme

Auf diese Weise können dann prinzipiell beliebig viele Zahlungssysteme integriert werden.

3.3.1 Transaktionsnummer

Um Anfragen, Zahlungsaufforderungen und Bezahlungen einander besser zuordnen zu können, muß das System eindeutige Transaktionsnummern generieren können. Diese Nummer gibt man dann bei allen die Transaktion betreffenden Aktionen an und erhält so eine eindeutige Zuordnung.

4 Spezifikation des Prototypen

In diesem Kapitel wird ein Prototyp des in Kapitel 3 entworfenen Systems für das Agentensystem Mole spezifiziert. Mole basiert auf Java 1.1, d.h., daß das Abrechnungssystem, das ja in Mole integriert wird, ebenfalls in Java 1.1 implementiert wird.

4.1 Spezifikation der Accountingkomponente

4.1.1 Ressourcen / Dienste

Das Accounting in Mole soll folgende Ressourcen abdecken:

- CPU-Zeit:

Mole besitzt zur Threadverwaltung einen eigenen Scheduler, das sogenannte MCP (master control program). Über diesen ist es möglich, die Vergabe von Rechenzeit mitzuprotokollieren. Die Klasse *MoleThread*, die die Vaterklasse fast aller Threads in Mole bildet, besitzt bereits Routinen und Variablen, um einen Thread einem Agenten, falls möglich, zuordnen zu können. Es muß dabei aber noch sichergestellt werden, daß das System den Besitzer eines Threads in allen Fällen richtig setzt. Danach kann der MCP so erweitert werden, daß der CPU-Verbrauch aller Agenten mitprotokolliert wird, falls das Accounting von CPU-Zeit mittels Konfiguration eingeschaltet ist.

- Speicherverbrauch:

Falls das Accounting des Speicherverbrauchs per Konfiguration aktiviert ist, so wird die Größe jedes Agenten beim Betreten und beim Verlassen der Location über die Serialisierung bestimmt. Ist außerdem ein periodisches Accounting des Speicherverbrauchs eingestellt, so muß ein neuer Thread gestartet werden, der im konfigurierten Intervall die Größe des belegten Speichers aller Agenten bestimmt und der Accounting-Datenhaltung übermittelt.

- Netzwerkzugriffe:

In Mole gibt es zwei Mechanismen, die gegebenenfalls aufs Netzwerk zugreifen:

- Messages (Nachrichten):

Falls konfiguriert, sollen je nach eingestellter Granularität des Netzwerk-Accountings die Anzahl und/oder die Größe und/oder die Netzwerkstart- und Zieladresse der Nachricht mitprotokolliert werden. Dabei werden diese Daten in der Datenhaltung sowohl beim Sender (als gesendete Nachricht) als auch beim Empfänger (als empfangene Nachricht) gespeichert. Es existieren also verschiedene Einträge für gesendete und empfangene Nachrichten.

- RPCs (Remote Procedure Calls):

Wie bei den Messages werden je nach Einstellung die Anzahl und/oder die Größe der Parameter und Rückgabewerte und/oder die Netzwerkstart- und Zieladresse des RPCs mitprotokolliert und sowohl dem Aufrufer (als aufgerufenen RPC) als auch dem Erbringer (als erbrachten RPC) in der Accounting-

Datenhaltung angerechnet. Es existieren ebenfalls verschiedene Einträge für aufgerufene und erbrachte RPCs.

Für Benutzeragenten sind diese zwei Mechanismen die einzigen Möglichkeiten, auf das Netzwerk zuzugreifen. Alle anderen Netzwerkzugriffe verbietet der Security Manager. Die Netzwerkzugriffe von Benutzeragenten sind also vom Accounting voll abgedeckt. Lediglich die Systemagenten haben die Möglichkeit, direkt aufs Netzwerk zuzugreifen. Diese Zugriffe bleiben vom Accounting dann unbemerkt. Prinzipiell könnten diese Zugriffe durch Eingriff in den Security Manager auf Socketebene auch protokolliert werden.

- Dienste:

Die von Agenten zur Verfügung gestellten Dienste müssen der Location von den Diensterbringern, wie seither auch, bekannt gemacht werden, allerdings noch zusätzlich mit dem Preis der Dienstleistung:

```
Location.registerMeForService(AgentName provider,  
                               String service,  
                               double price)
```

Der Diensterbringer kann danach jede Benutzung des Dienstes mittels der Methode

```
Location.accountService(AgentName provider,  
                        AgentName user,  
                        String service)
```

mitteilen. Diese Mitteilung ist dem Diensterbringer freigestellt. Sie ist aber die Voraussetzung dafür, daß der Agent später die Gebühr für die Dienstleistung bezahlt bekommt. Es sollte auch geprüft werden, ob der Agent, der die Dienstleistung meldet, auch wirklich der Agent ist, der den Dienst anbietet.

In einer Weiterentwicklung des Prototyps könnten Dienste komplett vom System vermittelt werden. Auf diese Weise kann dann sichergestellt werden, daß die diensterbringenden Systemagenten die geforderten Dienste auch zur Zufriedenheit des dienstbenutzenden Agenten ausführen. Auf diesem Wege bekommt das System dann auch mit, welcher Dienst von welchem Agenten in Anspruch genommen wurde. Die obige Meldung einer Dienst-Inanspruchnahme wäre dann überflüssig.

4.1.2 Konfiguration

Die Konfiguration des Accountings wird im Location-spezifischen Teil der Konfigurationsdatei der Engine vorgenommen. Standardmäßig ist das Accounting abgeschaltet.

Folgende Parameter existieren:

- *<location>.ACCOUNTING.SYSTEMAGENTS:*

Gibt an, ob das Accounting auf Benutzeragenten beschränkt werden soll (Wert *FALSE*, default) oder für alle Agenten durchgeführt wird (Wert *TRUE*).

- *<location>.ACCOUNTING.CPU:*
Mittels dieses Parameters kann das Accounting des CPU-Verbrauchs ein- (Wert *TRUE*) oder ausgeschaltet werden (Wert *FALSE*, default).
- *<location>.ACCOUNTING.MEM:*
Der Wert „*TRUE*“ schaltet die Speichermessung bei Betreten und Verlassen der Location ein. Ein numerischer Wert größer als 50 aktiviert zusätzlich noch eine Messung alle x ms, wobei x dem numerischen Wert entspricht. Jeder andere Wert schaltet das Accounting des Speichers ab (Voreinstellung).
- *<location>.ACCOUNTING.MSGS:*
Der Wert „0“ schaltet das Accounting von Nachrichten ab (default). Der Wert „1“ schaltet das Mitprotokollieren der Anzahl an Nachrichten ein, „2“ bedeutet Anzahl und Gesamtgröße der Nachrichten und „3“ bedeutet ein detailliertes Accounting jeder Nachricht mit Größe sowie Start- und Zieladresse.
- *<location>.ACCOUNTING.RPCS:*
Analog zum Nachrichtenaccounting, d.h. „0“ bedeutet aus (default), „1“ bedeutet nur Anzahl der RPCs, „2“ bedeutet Anzahl der RPCs und Größe der dabei übertragenen Daten, „3“ bedeutet Accounting im Detail, inklusive Netzwerkadressen.
- *<location>.ACCOUNTING.SERVICES:*
Dieser Parameter kontrolliert das Accounting von Dienstleistungen. Der Wert „*TRUE*“ schaltet das Accounting ein, jeder andere Wert schaltet es aus (default).

4.1.3 Accounting-Datenhaltung

Die Accounting-Daten werden von jeder Location gesammelt und in einem eigenen Objekt gehalten. Dieses Objekt muß ausreichend gegen Manipulation durch die Agenten oder andere Objekte gesichert sein. Ist dies nicht der Fall, so könnten Agenten die Menge an verbrauchten Ressourcen oder benutzen Dienstleistungen in der Datenhaltung verringern. Gleichzeitig muß es aber auch möglich sein, daß ein Agent seine eigenen Accounting-Daten abfragen kann.

4.2 Spezifikation der Gebührenberechnungskomponente

Die Gebührenberechnungskomponente wird in das Mole System integriert. Sie berechnet aufgrund der vom Accounting gelieferten Daten die anfallenden Gebühren der Agenten und rechnet diese dann mit den Agenten ab.

Das Gebührensystem wird, wie auch das Accounting, im Location-spezifischen Teil der Engine-Konfigurationsdatei konfiguriert. Es gibt zwei Möglichkeiten der Gebührenberechnung. Die erste ist die Berechnung mittels systemeigener Methoden, die viel-

seitig konfigurierbar sind. Die zweite Möglichkeit ist die Abrechnung mittels einer bestimmten Methode einer eigens bereitgestellten Klasse. Die Klasse kann einen beliebigen Namen tragen. Die Methode muß *computeFee()* heißen und ein Objekt der Klasse *AccountingDataEntry* als Parameter übernehmen. Dieses Objekt enthält alle Informationen, die das System über den Ressourcenverbrauch und die Dienstinnanspruchnahmen eines Agenten besitzt. Als Rückgabewert muß die Methode dann die berechneten Gebühren liefern.

Diese Eigenschaften sind sinnvollerweise in einem Interface zu definieren. Damit kann die Klasse, die die Gebühren berechnet, auch andere Funktionen haben. Sie muß lediglich das definierte Interface implementieren. Das Interface soll unter dem Namen *FeeComputer* definiert werden.

Die Gebührenberechnung ist standardmäßig deaktiviert. Mittels des Parameters *<Location>.BILLING* und des Wertes *SYSTEM* wird die Gebührenberechnung mit den systemeigenen Methoden aktiviert. Jeder andere Wert als *SYSTEM* oder *FALSE* wird als Name einer Klasse interpretiert, die eine vom Benutzer bereitgestellte Methode des Namens *computeFee()* mit oben genannten Eigenschaften zur Gebührenberechnung enthält.

4.2.1 Berechnung der Gebühren über eine dafür bereitgestellte Klasse

Wie schon erwähnt wird diese Variante mittels des Parameters *<Location>.BILLING* aktiviert, wobei der Wert ungleich „*SYSTEM*“ und „*FALSE*“ sein muß. Die in der Konfiguration angegebene Klasse muß das Interface *FeeComputer* implementieren. Jedesmal wenn das Gebührensystem die Gebühren für einen Agenten berechnen will, wird dann die Methode *computeFee* dieser Klasse aufgerufen und die Accounting-Daten des Agenten übergeben. Das System erwartet daraufhin als Rückgabewert die angefallenen Gebühren.

Beispiel:

Der Eintrag

```
testloc.BILLING PrivateBilling
```

in der Konfigurationsdatei der Engine bewirkt, daß bei jeder anfallenden Gebührenberechnung die Methode *computeFee* der Klasse *PrivateBilling* mit dem Accounting-Daten des Agenten als Parameter aufgerufen wird:

```
PrivateBilling.computeFee(accountingDataOfAgentX);
```

Die Methode muß statisch sein, da der Name der Klasse dem System erst zur Laufzeit, beim Einlesen der Konfiguration, bekanntgemacht wird. Ein statischer Aufruf ist dann einfacher zu realisieren als die Klasse zu instanziiieren.

4.2.2 Konfiguration und Berechnung der Gebühren mit systemeigenen Methoden

Bei dieser Variante werden die Gebühren für die einzelnen Ressourcen, ebenfalls im Location-spezifischen Teil der Engine-Konfigurationsdatei, wie folgt festgesetzt:

- CPU: Der Schlüssel zur Konfiguration des Tarifs für den CPU-Verbrauch lautet:

<location>.BILLING.CPU

Der Wert des Schlüssels sieht wie folgt aus:

EBNF Syntax:

cpu_eintrag := <intervall>:<preis>[„x“]{„,“<cpu_eintrag>}

intervall := <Start>„-“[<Ende>]

Anmerkungen:

- Sollte der Preisangabe ein „x“ folgen, so gilt der Preis nicht pauschal für das angegebene Intervall, sondern pro Millisekunde.
- Sollte das Ende eines Intervalls nicht spezifiziert werden, so ist das Intervall nach oben offen.
- Die einzelnen Intervalle sollten sortiert sein und keine Lücken aufweisen.

Beispiel:

`bsplocation.BILLING.CPU 1-100:1;101-200:2;201-:0.01x`

d.h. 1-100 ms CPU-Zeit kosten 1 Einheit, 101-200 ms kosten 2 Einheiten und jede ms über 200ms kostet 0.01 Einheiten.

- Speicher: Der Tarif für den Speicherverbrauch wird mittels des Schlüssels

<location>.BILLING.MEM

festgelegt:

EBNF Syntax:

mem_eintrag := <intervall>„,“<preis>[„x“]{„,“<mem_eintrag>}

intervall := <Start>„-“[<Ende>]

Anmerkungen:

- Sollte der Preisangabe ein „x“ folgen, so gilt der Preis nicht pauschal für das angegebene Intervall, sondern pro byte*ms.
- Sollte das Ende eines Intervalls nicht spezifiziert werden, so ist das Intervall nach oben offen.
- Die einzelnen Intervalle sollten sortiert sein und keine Lücken aufweisen.

Beispiel:

`bsplocation.BILLING.MEM 1-:0.001x`

d.h. Ein Byte Speicher kostet pro Millisekunde grundsätzlich 0.001 Einheiten.

- Nachrichten: Der Preis für die Kommunikation mittels Nachrichten wird über die Schlüssel

<location>.BILLING.MSGn

festgesetzt, wobei n zwischen 1 und 9 liegt und der Wert des Schlüssels wie folgt auszusehen hat:

EBNF Syntax:

```
msg_eintrag := [(<subnetz>| „local“) „:“ <msg_teileintrag>{ „:“ <msg_teileintrag>}
msg_teileintrag := („#“ | „B“) „:“ <intervall> „:“ <preis>[ „x“ ]
subnetz := <Netzwerk IP Adresse> „/“ <dezimale Subnetzmaske>
intervall := <Start> „-“ [<Ende>]
```

Anmerkungen:

- Das führende „#“ besagt, daß der folgende Eintrag pro Nachricht gilt; ein führendes „B“ besagt, daß der Eintrag pro übertragenes Byte gilt.
- Die Subnetzangabe bzw. die Angabe „local“ können auch weggelassen werden, dann gilt der Eintrag für alle Nachrichten.
- Sollte der Preisangabe ein „x“ folgen, so gilt der Preis nicht pauschal für das angegebene Intervall, sondern pro Anzahl bzw. Byte.
- Sollte das Ende eines Intervalls nicht spezifiziert werden, so ist das Intervall nach oben offen.
- Die einzelnen Intervalle sollten sortiert sein und keine Lücken aufweisen.
- Die Nummer n am Ende des Schlüssels legt auch die Reihenfolge fest, nach der die einzelnen Einträge beim Berechnen der Kosten nach Übereinstimmung geprüft werden. D.h., daß speziellere Einträge vor allgemeineren kommen sollten, da sie sonst nicht berücksichtigt werden.

Beispiel:

```
bsplocation.BILLING.MSG1 local:#:0-50:0;#:51-:0.1x
bsplocation.BILLING.MSG2 1.0.0.0/8:B:0-:0.25x
bsplocation.BILLING.MSG3 0.0.0.0/0:B:0-:0.5x
```

d.h. lokale Nachrichten sind bis zu einer Anzahl von 50 kostenlos, jede lokale Nachricht über 50 kostet 0.1 Einheiten. Nachrichten ins Subnetz 1.x.x.x kosten 0.25 Geldeinheiten pro Byte; Nachrichten in alle anderen Subnetze kosten 0.5 Einheiten pro Byte.

Beispiel2:

```
bsplocation.BILLING.MSG1 :#:0-50:0.1x:#:51-:0.09x
```

d.h. der Tarif ist für alle Nachrichten gleich, nämlich 0.1 Geldeinheiten pro Nachricht für die ersten 50 Nachrichten und 0.09 Geldeinheiten für jede weitere.

- RPCs: der Tarif für RPCs wird in derselben Art wie der von Nachrichten festgesetzt. Die Schlüssel lauten

<location>.BILLING.RPCn ,

wobei n zwischen 1 und 9 liegt.

Beispiel:

```
bsplocation.BILLING.RPC1 local:#:0-50:0;#:51-:0.1x  
bsplocation.BILLING.RPC2 1.0.0.0/8:B:0-:0.25x  
bsplocation.BILLING.RPC3 0.0.0.0/0:B:0-:0.5x
```

- Dienstleistungen von Agenten: Der Preis von durch Agenten erbrachte Dienstleistungen wird bereits beim Accounting dem Verzeichnis der Dienstleistungen (Service Dictionary) entnommen.

4.2.3 Datenhaltung

Die Gebühren werden unter Verwendung der aus dem Accounting gewonnenen Daten und der konfigurierten Tarife bzw. der *computeFee()* Methode einer vom Betreiber bereitgestellten Klasse berechnet und zentral gespeichert.

4.2.4 Abrechnung

Abrechnungszeitpunkt:

Der Abrechnungszeitpunkt wird mittels des Schlüssels *<location>.BILLING.TIME* in der Konfigurationsdatei der Engine festgesetzt. Dabei gibt es folgende Möglichkeiten:

- bei Verlassen der Location (Wert „0“):
In diesem Fall werden die Gebühren berechnet und mit dem Agenten abgerechnet, wenn dieser terminiert oder von der Location wegmigriert.
- periodisch (Wert „1“):
Hier werden die Gebühren nach Ablauf einer mittels des Schlüssels *<location>.BILLING.PERIOD* konfigurierten Zeitspanne berechnet und mit allen Agenten abgerechnet. Verläßt ein Agent die Location, so wird ihm noch der Restbetrag in Rechnung gestellt.
- nach Ansammlung einer bestimmten Summe (Wert „2“):
Bei dieser Variante wird mit einem Agenten abgerechnet, wenn dieser die mittels des Schlüssels *<location>.BILLING.SUM* konfigurierbare Gebührensumme überschreitet. Um dies zu realisieren, müssen bei jedem Zugriff auf die Accountingdaten die Gebühren der Agenten neu berechnet werden. Sollte ein Agent die Gebührensumme überschreiten, wird mit ihm abgerechnet. Bei Verlassen der Location wird mit dem jeweiligen Agenten der Restbetrag abgerechnet.
Durch Setzen des Schlüssels *<location>.BILLING.PREPAY* auf *true* kann veranlaßt werden, daß die Agenten die konfigurierte Summe jeweils im Voraus zahlen müssen. Es muß also jeder Agent beim Betreten der Location die konfigurierte Summe bezahlen. Ist der Betrag dann aufgebraucht, muß er erneut zahlen.

Abrechnungsvorgang:

Der Abrechnungsvorgang verläuft folgendermaßen:

1. Die Location sendet dem Agenten eine Zahlungsaufforderung (siehe auch Kapitel 4.3.1, "Definition der Schnittstellen: Mole <-> Zahlungssystem").
2. Die Location wartet den mittels des Schlüssels `<location>.BILLING.TIMEOUT` konfigurierten Zeitraum darauf, daß der Agent bezahlt.
3. Sollte die Zahlung in diesem Zeitraum eintreffen so läuft alles normal weiter. Sollte der Agent nicht bezahlen, so wird er je nach Wert des Schlüssels `<location>.BILLING.PENALTY` entweder terminiert (Wert „0“, default) oder zur Location, auf der er erstmals gestartet wurde, zurückgeschickt (Wert „1“).

Desweiteren wird ein Eintrag in die Logdatei „debtors.log“ gemacht. Dieser Eintrag enthält neben dem Namen des Agenten und der Höhe der Schulden zusätzlich noch genaue Informationen über die verbrauchten Ressourcen.

4.2.5 Währungseinheit der Geldbeträge

Alle Mole-Systeme benutzen dieselbe Währungseinheit, den EURO. Alle in Mole konfigurierten, geforderten oder bezahlten Geldbeträge sind in dieser Währung. Zahlungen, die das Mole-System verlassen oder von extern in das Mole-System gelangen, muß die Mole-Komponente zur Integration der Zahlungssysteme von EURO in die benötigte bzw. von der externen Einheit in EURO umrechnen.

Eine feste Währungseinheit wurde gewählt, um den Umgang mit dem Gebühren- und Zahlungssystem von Mole für die Agenten möglichst einfach zu gestalten. Ein lästiges Umrechnen von Einheiten entfällt. Mißverständnisse, die auf unterschiedlichen Währungen beruhen, werden vermieden.

Der EURO wurde der D-Mark aufgrund des baldigen Ausscheidens der D-Mark vorgezogen.

4.2.6 Abfrage der Gebühren durch Agenten

Den Agenten soll es möglich sein, sowohl ihre eigenen zu zahlenden Gebühren als auch die Gebührenstruktur, d.h. die Preise für die einzelnen Ressourcen, von den Locations abzufragen. Das Abfragen der Preise einer Location soll schon vor Migration auf dieselbe möglich sein, um eine Migration von den eventuell entstehenden Kosten abhängig machen zu können.

Bei der Abfrage der angefallenen Gebühren durch einen Agenten (Klasse *Location*, Methode *getFees()*), muß sichergestellt sein, daß nur der Agent selbst seine Gebühren abfragen kann. Dies kann realisiert werden, indem man den Namen des Agenten, dem der aktuelle Thread zugeordnet ist, mit dem übergebenen Agentennamen vergleicht. Sind sie identisch, so ist sichergestellt, daß der Agent seine eigenen Gebühren abfragt.

Die Abfrage der Preise der einzelnen Ressourcen einer Location erfolgt immer über die eigene Location (*Location.getPrice(locationname)*). Diese leitet die Anfrage dann weiter (bei Locations derselben Engine direkt, bei entfernten Locations per RMI).

Problematisch sind hierbei Locations mit eigener Gebührenberechnungsmethode. Man könnte in diesem Fall in der Klasse, die die Gebührenberechnungsmethode enthält auch eine Methode zur Abfrage der Gebühren voraussetzen. Da durch die freie Programmierung der Gebührenberechnungsmethode keinerlei Vorgaben für die Berechnung existieren, ist es aber nicht möglich die Preisstruktur in eine feste Syntax zu fassen. Die Rückgabe müßte also beliebig sein. Agenten, die die Location nicht kennen, ist es dann prinzipiell nicht möglich die Semantik des Rückgabewertes zu verstehen. Agenten, die mit der Preisstruktur der Location vertraut sind, haben aber kein Interesse diese abzufragen.

Aufgrund dieser Problematik liefert die Abfragemethode der Preisstruktur bei Locations mit eigener Gebührenberechnungskomponente keine Informationen.

Die Rückgabe der Tarifinformationen von Locations, die systemeigene Methoden zur Gebührenberechnung verwenden, erfolgt in einem Array:

```
{<CPU Gebühren>, <Speicher Gebühren>, <Message Gebühren>, <RPC Gebühren>}
```

Die einzelnen Elemente des Arrays sind jeweils Vektoren, die spezielle Objekte zur Gebührenrepräsentation enthalten.

Die Abfrage der Gebühren für Dienstleistungen erfolgt über den Aufruf *Location.priceOfService(service, provider)*. Ein Agent, der ein Dienstleistung anbietet, muß diese vorher bei der Location mit Preis anmelden (*Location.registerMeForService(provider, service, price)*).

4.3 Spezifikation der Zahlungskomponente

Im folgenden Kapitel werden die Schnittstellen zur Integration eines Zahlungssystems in Mole definiert. Danach wird eine prototypische Zahlungskomponente spezifiziert.

4.3.1 Definition der Schnittstellen: Mole <-> Zahlungssystem

Wie in Kapitel 3.3 entworfen erfolgt die Integration eines Zahlungssystems mittels einer für jedes Zahlungssystem zu entwickelnden Klasse: der *PaymentObject*-Klasse. Diese Klasse setzt die in Mole standardisierten Aufrufe zur Benutzung des Zahlungssystem in die proprietären Protokolle des Zahlungssystem um. Desweiteren enthalten die Instanzen dieser Klasse Zahlungssystem-spezifische Informationen über den Benutzer, wie z.B. die Kreditkartennummer.

Die Schnittstellen sehen folgendermaßen aus:

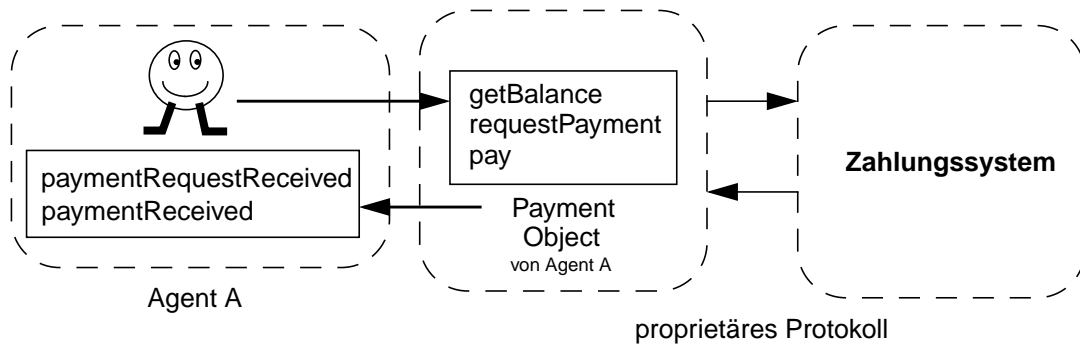


ABBILDUNG 9. Schnittstellen zwischen Agent (auch Location) und dem Schnittstellenobjekt PaymentObject

Jedes am Zahlungsverkehr teilnehmende Objekt muß sich eine neue Instanz der *PaymentObject*-Klasse erzeugen. Dies erfolgt mittels dem Konstruktor der *PaymentObject*-Klasse, der folgende Form hat:

- **Konstruktor:** *PaymentObject(Object owner, Object initData)*
Bei der Erzeugung des Payment-Objekts muß der Besitzer übergeben werden sowie ein Objekt, das Initialisierungsinformationen des Benutzers für das Zahlungssystem enthält. Dieses Objekt kann je nach implementierten Zahlungssystem Informationen über die elektronische Geldbörse oder Kreditkarteninformationen enthalten. Diese Initialisierungsdaten sind die einzigen Zahlungssystem-spezifischen Daten, d.h. je nach implementierten Zahlungssystem müssen die Benutzer dem Payment-Objekt unterschiedliche Informationen übergeben.

Die folgenden Methoden, die das Payment-Objekt dem Agenten bzw. den Locations zur Verfügung stellen muß, sind für alle potentiellen Zahlungssysteme gleich. Diese Methoden rufen die Agenten oder die Locations auf, um das Zahlungssystem zu nutzen:

- ***double getBalance():***
liefert dem Agenten bzw. der Location seinen aktuellen Geldstand (bei Zahlungssystemen, die auf digitalem Bargeld basieren, entspricht dies dem Inhalt der Geldbörse, ansonsten dem Kontostand des dazugehörigen Kontos).
- ***requestPayment(AgentName payer, LocationName ln, double amount, String description, String id):***
Diese Methode wird vom Agenten oder einer Location aufgerufen, um einem anderen Agenten mitzuteilen, daß man eine Zahlungsforderung an ihn stellt. Dabei muß der Methode der Empfänger, sein Aufenthaltsort, die Höhe der Geldforderung und eine Beschreibung des Grundes der Zahlungsaufforderung übergeben werden. Als

letzter Parameter kann noch eine Transaktions-Identifikation angegeben werden, um die Ereignisse einer Transaktion einander besser zuordnen zu können.

- *pay(AgentName payer, LocationName ln, double amount, String description, String id):*
Diese Methode tätigt eine Zahlung für den Agenten. Der Agent bzw. die Location teilt dem Payment-Objekt mit, welchen Betrag er/sie an wen bezahlen möchte und mit welchem Kommentar. Auch hier ist der letzte Parameter eine Transaktionsidentifikation.

Zur Signalisierung von Zahlungssystem-spezifischen Ereignissen ruft das Payment-Objekt folgende Methoden seines Besitzers (Agent oder Location) auf. Verfügt der Besitzer eines Payment-Objekts nicht über diese Methoden, kann keine Benachrichtigung über eingegangene Ereignisse erfolgen:

- *paymentRequestReceived(AgentName sender, LocationName senderlocation, Double amount, String description, String id):*
Das Payment-Objekt ruft diese Methode beim Agenten/Location auf, wenn es von einem anderen Agenten/Location eine Zahlungsaufforderung bekommt. Dabei übermittelt sie den Absender, die Höhe des Betrages sowie den Grund der Zahlungsforderung und eine Transaktionsidentifikation an den Agenten. Dieser kann sich daraufhin entschließen, die Zahlung zu tätigen oder nicht.
- *paymentReceived(AgentName sender, LocationName senderlocation, Double amount, String description, String id):*
Diese Methode des Agenten oder der Location wird vom Payment-Objekt aufgerufen, wenn eine Zahlung für den Agenten bzw. die Location eingetroffen ist. Der Agent oder die Location bekommen vom Payment-Objekt dabei Absender, Höhe der Zahlung, Beschreibung des Zahlungsgrundes und die Transaktionsidentifikation mitgeteilt.

Transaktionsidentifikation:

Mittels der Methode *String Engine.getId()* kann man von der Engine eine systemweit eindeutige Transaktionsidentifikation bekommen und bei finanziellen Transaktionen nutzen.

4.3.2 Prototypische Zahlungskomponente

Zu Test- und Demonstrationszwecken soll es möglich sein, finanzielle Transaktionen zu tätigen. Dafür ist ein Payment-Objekt notwendig, das die definierten Schnittstellen implementiert. Für unsere Zwecke reicht es, nur die Schnittstellen zu Mole hin zu implementieren. Es ist nicht notwendig, die Kommunikation zu einem Zahlungssystem zu implementieren. Die Payment-Objekte können dann direkt miteinander kommunizieren, was die Implementierung stark vereinfacht (siehe Abbildung 10). Die Abstim-

mung auf ein Zahlungssystem oder die Implementierung eines eigenen ist also für die Test- und Demonstrationszwecke überflüssig.



ABBILDUNG 10. Direkte Kommunikation der PaymentObjects zu Test- und Demonstrationszwecken

4.4 Interaktion der einzelnen Komponenten in einem Beispiel

Die folgende Graphik und der Text veranschaulichen die Interaktion der einzelnen spezifizierten Komponenten in Mole beispielhaft anhand einer gebührenpflichtigen Dienstleistung, die von Agent A angeboten und von Agent B in Anspruch genommen wird.

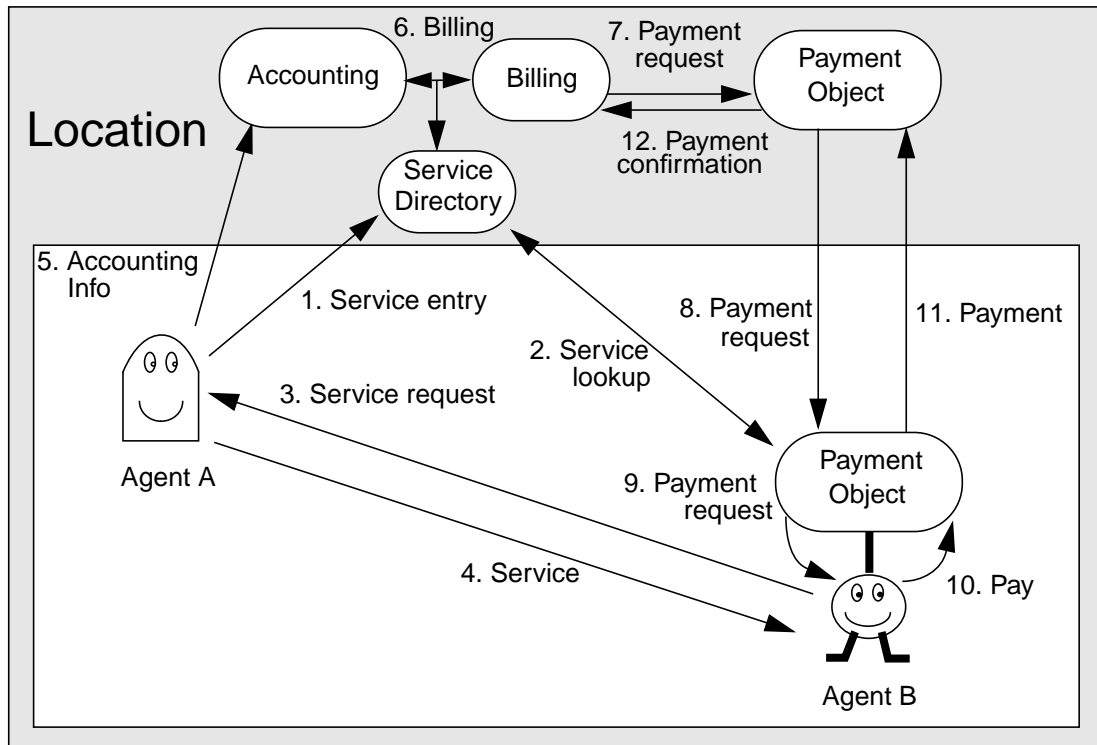


ABBILDUNG 11. Ablauf einer Dienstleistung mit Bezahlung

1. Agent A macht seine Dienstleistung und den Preis dafür bei der Location bekannt.
2. Agent B fragt bei der Location nach dem Anbieter einer bestimmten Dienstleistung sowie nach dem dazugehörigen Preis und bekommt Agent A und den Preis mitgeteilt.
3. Agent B fordert die kostenpflichtige Dienstleistung bei Agent A an.
4. Agent A erbringt die Dienstleistung
5. Agent A meldet das Erbringen der Dienstleistung für Agent B an die Accounting-Komponente der Location
6. Bei der nächsten Abrechnung der Location mit Agent B wird auch die von Agent A erbrachte kostenpflichtige Dienstleistung mitabgerechnet.
7. Die Location beauftragt ihr Payment-Objekt, Agent B eine Zahlungsaufforderung zu senden.
8. Das Payment-Objekt der Location sendet Agent Bs Payment-Objekt die Zahlungsaufforderung (im proprietären Protokoll des verwendeten Zahlungssystems).
9. Agent Bs Payment-Objekt teilt die eingegangene Zahlungsaufforderung seinem Agenten mit.
10. Agent B entschließt sich zu bezahlen und beauftragt sein Payment-Objekt, die Zahlung durchzuführen.
11. Bs Payment-Objekt führt die Zahlung an das Payment-Objekt der Location durch (im proprietären Protokoll des verwendeten Zahlungssystems).
12. Das Payment-Objekt der Location informiert die Location über die eingegangene Zahlung.

5 Implementierung

Die im Kapitel 4 entworfenen prototypischen Komponenten wurden im Rahmen dieser Diplomarbeit vollständig implementiert. Das folgende Kapitel beschäftigt sich mit den Details und Problemen dieser Implementierung.

5.1 Implementierung der Accountingkomponente

Die implementierten Mechanismen der Accountingkomponente sind über diverse Mole-Klassen verteilt. Sie befinden sich an den Stellen, an denen ein Accounting der jeweiligen Ressource möglich ist. Im nächsten Kapitel werden die Implementierungen des Accounting der einzelnen Komponenten beschrieben.

5.1.1 Accounting der einzelnen Ressourcen

- CPU-Zeit:

Zuerst wurde analysiert, ob das Besitzer-Attribut der einzelnen Mole-Threads in allen Fällen richtig gesetzt wird. Das Besitzer-Attribut steht in der Variable *owner-Agent* der Klasse *SecurityInfo*, von der jeder Mole-Thread eine Instanz besitzt und gibt an, für welchen Agenten der Thread Programmcode ausführt. Der Besitzer-agent des Mole-Threads kann über diverse Methoden der Klasse *MoleThread* gesetzt bzw. geändert werden. Die Analyse ergab, daß das Besitzer-Attribut vom System konsequent gesetzt und geändert wird. Dies ist eine wichtige Voraussetzung für das Accounting der CPU-Zeit.

Das Accounting der CPU-Zeit wurde dann in das Master Control Program *MCP* integriert. Ein schon vorhandener Mechanismus, der die CPU-Zeiten aller Mole-Threads auf Threadebene mitprotokollierte, wurde so erweitert, daß der Besitzer des Mole-Threads ermittelt wird und die CPU-Zeit sowie der Name des Besitzers an die Accounting-Datenhaltung mittels der Methode *addMCPTime(AgentName anAgent, long time)* der *AccountingData*-Klasse übermittelt wird, vorausgesetzt, daß es sich um keinen System-Thread handelt oder der Besitzer der Moleview-Agent ist. Desweiteren muß vorher die Konfiguration daraufhin überprüft werden, ob das Accounting der CPU-Zeit überhaupt aktiviert wurde bzw. ob Systemagenten auch accountet werden sollen. Dabei ist darauf zu achten, daß man die Konfigurationsdaten derjenigen Location auswählt, auf der sich der Agent befindet, da das Master Control Program Rechenzeiten für die Threads aller Locations der Engine vergibt.

- Speicher:

Voraussetzung für das Accounting des Speichers ist eine Methode, die den belegten Speicher eines Objekts zu einem beliebigen Zeitpunkt mittels der Serialisierung mißt. Hierzu wurde die Klasse *Accounting* implementiert. Die Methode *sizeOf(Object obj)* dieser Klasse bestimmt die Größe des Objekts *obj*, indem sie es in einen *ByteArrayOutputStream* serialisiert und danach die Größe des Streams bestimmt. Diese Vorgehensweise funktioniert nur für Objekte, die das Interface

java.io.Serializable implementiert haben. Die Agenten, deren Größen damit gemessen werden sollen, haben dieses Interface aber zwangsläufig implementiert, da alle Agenten von der Klasse *Agent* abgeleitet sind, die bereits das Interface *Serializable* implementiert. Außerdem funktioniert der Migrationsmechanismus ebenfalls unter Zuhilfenahme der Serialisierung, d.h., daß nur serialisierbare Agenten migrieren können. Desweiteren gibt es die Möglichkeit, über das Schlüsselwort *transient* Klassenvariablen zu markieren, die nicht serialisiert werden. Der Speicher, den so markierte Variablen belegen, bleibt dann vom Accounting unbemerkt.

Mittels der *sizeOf*-Methode wird die Größe jedes Agenten, falls das Accounting des Speichers konfiguriert ist, beim Betreten der Location, genauer gesagt, wenn ein Eintrag in der Accounting-Datenhaltung für ihn eingerichtet wird, bestimmt. Dies geschieht ebenfalls beim Verlassen der Location, bevor das letzte Mal mit ihm abgerechnet wird. Zusammen mit der aktuellen Systemzeit im absoluten Millisekunden-Format (Millisekunden seit dem 1.1.1970 0 Uhr) werden die Messungen in der Accounting-Datenhaltung gespeichert. Dies geschieht mittels der Methode *addMemInfo(AgentName anAgent, long size)* der *AccountingData* Klasse. Das absolute Millisekunden-Format wurde gewählt, da damit das Subtrahieren von Systemzeiten extrem einfach ist.

Für das periodische Messen der Speicherbelegung wurde die neue Klasse *MemAccThread* implementiert. Diese Klasse, die von *MoleThread* abgeleitet ist, mißt in einer Endlosschleife die Speicherbelegung aller Agenten auf der Location und übergibt sie der Accounting-Datenhaltung, die die aktuelle Systemzeit hinzufügt. Danach legt sich der Thread für die konfigurierte Zeit (Intervall der Speicherüberprüfung) schlafen, bevor er wieder von vorne anfängt.

Ist ein periodisches Abfragen der Speicherbelegung konfiguriert, so wird beim Start der Location ein *MemAccThread* gestartet und die Länge des Meßintervalls übergeben.

Bevor ein Agent die Location verläßt (Migration, Terminierung), wird, wie oben beschrieben, ein letztes Mal die Größe des belegten Speichers gemessen. Danach erfolgt die Berechnung des Speicherverbrauchs. Gemäß der Formel für die Näherung des Speicherverbrauchs aus Kapitel 2.2 wird jeweils der Zeitabstand zweier aufeinanderfolgenden Messungen bestimmt und mit dem Mittelwert der beiden gemessenen Speicherbelegungen multipliziert. Alle so gewonnenen Werte werden aufsummiert und ergeben dann die Näherung des Speicherverbrauchs in Bytes pro Millisekunde.

- Messages:

Das Accounting von Messages erfolgt an zwei Stellen. Alle Messages, die für Agenten einer anderen Location bestimmt sind, werden mittels der Methode *lowlevelsend()* der Klasse *Location* verschickt. Dies ist eine geeignete Stelle, um von lokalen Agenten verschickte Messages zu accounten, mit Ausnahme derjenigen Messages, die die Location nicht verlassen, sondern an Agenten derselben Location verschickt werden. Auf den ersten Blick könnte man meinen, daß die Methode *syncmessage()* besser geeignet wäre, da diese Methode für lokale und nicht lokale Messages aufgerufen wird. In dieser Methode findet aber noch keine Überprüfung der Ziel-Location statt, d.h., daß Messages, die gar nicht versendet werden, weil ihre Ziel-Location unbekannt ist, trotzdem als Netzwerkzugriff accountet würden (siehe hierzu Abbildung 12).

Lokal verschickte Messages und solche, die von Agenten anderer Locations kommen, werden in der Methode *deliverMessage()* der Klasse *Location* mitprotokolliert. Diese Methode stellt ankommende Messages den gewünschten Empfänger-Agenten zu.

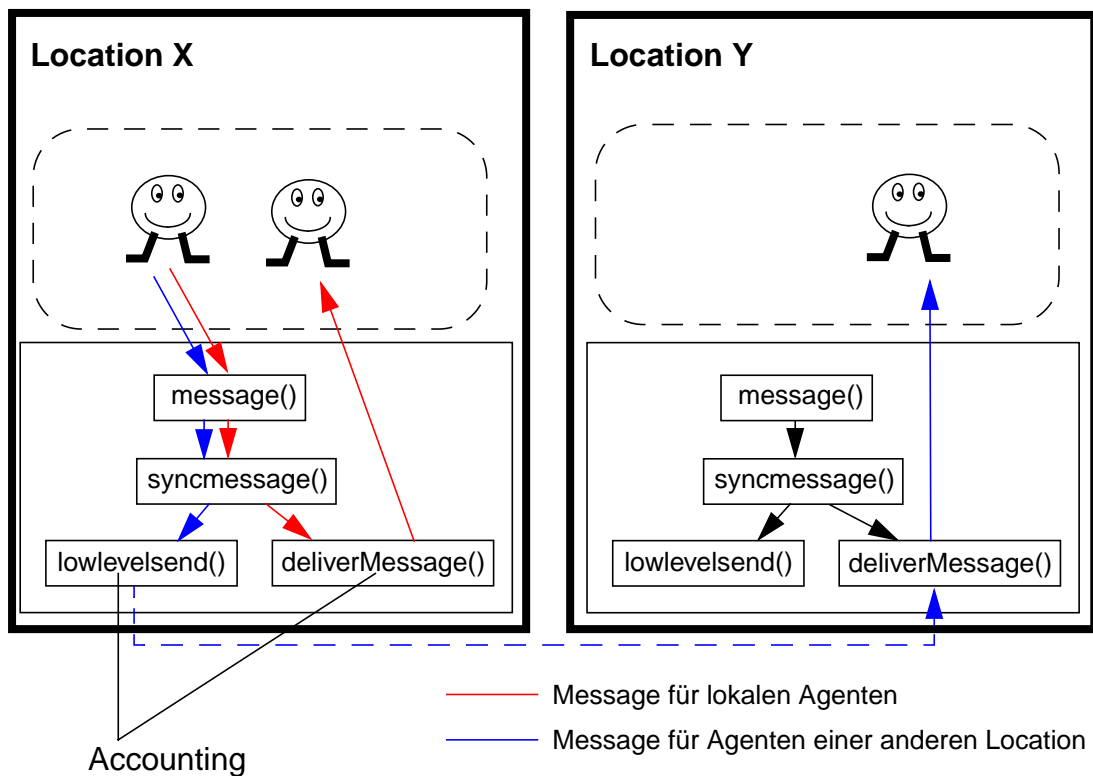


ABBILDUNG 12. Zustellung einer Message in Mole

Je nach eingestelltem Umfang des Message-Accountings werden verschiedene Informationen mittels der Methode *addMsg(...)* der Klasse *AccountingData* an die Accounting-Datenhaltung übermittelt. Entweder wird nur das Versenden bzw. Empfangen einer Message gemeldet (Wert „1“ des Konfigurationsparameters `<location>.ACCOUNTING.MSGS`) oder das Versenden bzw. Empfangen und die Größe der Nachricht (Wert „2“, die Größe der Nachricht wird in der Methode *sizeOf()* der Klasse *Accounting* mittels Serialisierung bestimmt) oder zusätzlich noch die Start- und Zieladresse der Message, die mittels der Resolver-Klasse *ResolverFake* aus Start- und Ziel-Location der Message aufgelöst wird (Wert „3“). Desweiteren werden der Accounting-Datenhaltung der Name des Agenten übermittelt, der die Message empfangen bzw. gesendet hat sowie die Richtung der Message - ankommend oder abgehend.

- RPCs:

Das Accounting der RPCs erfolgt ebenfalls an zwei Stellen. In der Methode *call()* der Klasse *Location*, mit der ein RPC-Aufruf gestartet wird, werden alle abgehenden RPCs mitprotokolliert. In der Methode *dispatchRPC()* der Klasse *Location*, die die RPCs an die entsprechenden Agenten weiterleitet, werden alle ankommenden RPCs mitprotokolliert (siehe Abbildung 13).

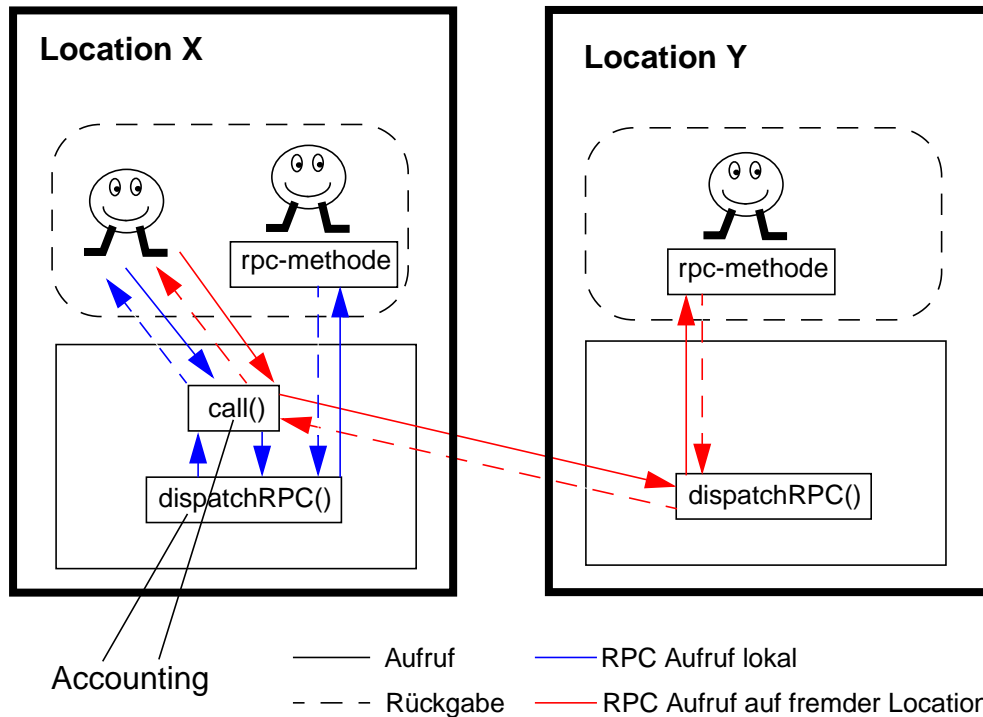


ABBILDUNG 13. Ablauf eines RPCs

Die mitprotokollierten RPCs werden mittels der Methode *addRPC()* der Klasse *AccountingData* an die Accounting-Datenhaltung weitergegeben. Dabei hängt es vom konfigurierten Grad des RPC-Accountings ab, welche Daten an die Datenhaltung übergeben werden. Im einfachsten Fall wird nur das Aufrufen bzw. das Ausführen eines RPCs, der Name des Agenten, der den RPC aufgerufen bzw. ausgeführt hat und ob es sich um einen aufgerufenen oder ausgeführten RPC gehandelt hat, übermittelt (Wert „1“ des Konfigurationsparameters *<location>.ACCOUNTING.RPCS*). Die nächst genauere Stufe des RPC-Accountings (Wert „2“) übermittelt zusätzlich noch die Größe der übermittelten Parameter und die Größe des Rückgabewertes. Beide werden mittels Serialisierung über die Methode *Accounting.sizeOf()* bestimmt. Die genaueste Stufe des RPC-Accounting (Wert „3“) übergibt der Accounting-Datenhaltung außerdem die Start- und Zieladresse des RPCs. Diese werden aus Start- und Ziel-Location des RPCs mittels der Resolver-Klasse *ResolverFake* aufgelöst.

5.1.2 Accounting von Diensten

Wie im Kapitel 4.1.1 spezifiziert, wird das Accounting der Dienste von den dienstleistenden Agenten selbst durchgeführt. Hierfür wurde die Methode `accountService(AgentName provider, AgentName user, String service)` in der Klasse `Location` implementiert (siehe Abbildung 14). Diese Methode übermittelt die Benutzung des Dienstes `service` durch den Agenten `user`, bereitgestellt vom Agenten `provider` an die Accounting-Datenhaltung mittels der Methode `addServiceInfo()`. Den Preis der Dienstleistung entnimmt die Methode dem Dienstleistungsverzeichnis (`ServiceDictionary`) mittels der Methode `priceOfService()` der Klasse `Location`. Hierzu muß der Dienst und der Preis des Dienstes vom Agenten an das Service Dictionary durch Aufrufen der Methode `registerMeForService()` der Klasse `Location` übermittelt worden sein.

Vor der Übermittlung an die Datenhaltung wird überprüft, ob die Meldung der Dienstbenutzung auch tatsächlich vom Dienstbringer stammt. Hierzu wird der Benutzer des aktuellen Threads ermittelt. Ist er identisch mit dem Agentennamen des Dienstbringers (`provider`), so erfolgt die Übermittlung an die Accounting-Datenhaltung. Andernfalls wird die Meldung ignoriert.

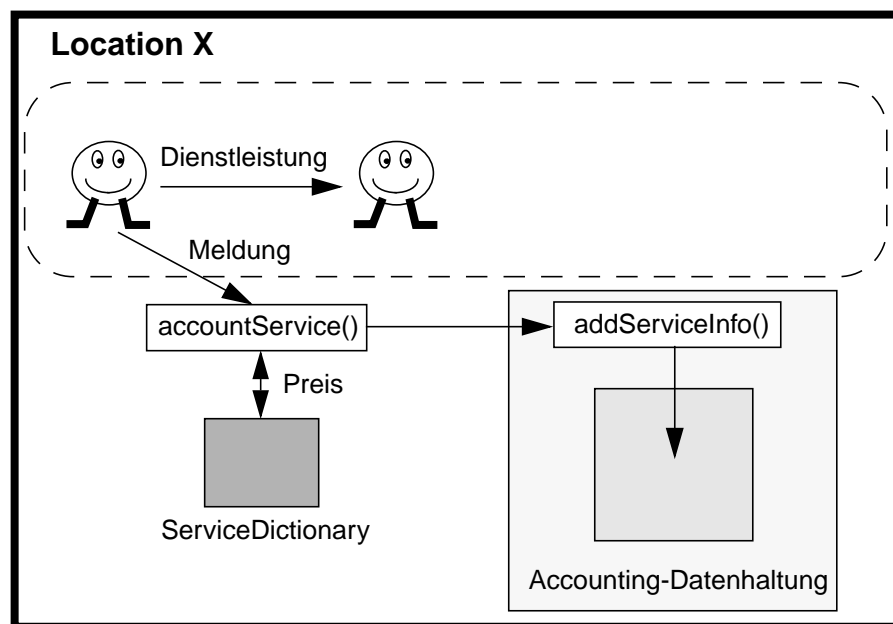


ABBILDUNG 14. Meldung einer Dienstleistung an die Accounting-Komponente

5.1.3 Implementierung der Accounting-Datenhaltung

Die Klasse `AccountingDataEntry` wurde zur Accounting-Datenhaltung neu implementiert. Sie enthält alle Accounting Daten für einen Agenten. Die Klassenvariablen sind:

- `agentName`: der Name des Agenten dessen Accounting-Daten gehalten werden
- `mcpTime`: die bisher verbrauchte CPU-Zeit in ms

- *nrOfSendMsgs*: die Anzahl der versendeten Nachrichten
- *nrOfRecvMsgs*: die Anzahl der empfangenen Nachrichten
- *sizeOfSendMsgs*: die Gesamtgröße aller gesendeten Nachrichten in Byte
- *sizeOfRecvMsgs*: die Gesamtgröße aller empfangenen Nachrichten in Byte
- *detailedMsgInfo*: eine Referenz auf ein anderes Objekt der Oberklasse *java.util.Vector* (*AccountingMsgData*), das detaillierte Informationen über alle gesendeten und empfangenen Nachrichten hält (*AccountingMsgDataEntry*):
 - *sourceIP*: Adresse des Senders der Nachricht
 - *destIP*: Adresse des Empfängers
 - *size*: Größe der Nachricht in Byte
- *nrOfServedRPCs*: die Anzahl der erbrachten RPCs
- *nrOfCalledRPCs*: die Anzahl der aufgerufenen RPCs
- *sizeOfServedRPCs*: die Gesamtgröße aller bei erbrachten RPCs verschickten und empfangenen Daten in Byte
- *sizeOfCalledRPCs*: die Gesamtgröße aller bei gerufenen RPCs verschickten und empfangenen Daten in Byte
- *detailedRPCInfo*: Analog zu *detailedMsgInfo* eine Referenz auf ein Objekt der Oberklasse *java.util.Vector* (*AccountingRPCData*), das detaillierte Informationen über alle erbrachten und aufgerufenen RPCs hält (*AccountingRPCDataEntry*):
 - *sourceIP*: Adresse des Aufrufers des RPCs
 - *destIP*: Adresse des Erbringers des RPCs
 - *size*: Größe der übermittelten Daten (Parameter und Rückgabewerte) in Byte
- *memoryUsage*: Eine Referenz auf ein Objekt der Objektklasse *java.util.Vector* (*AccountingMemData*), das die Speicherbelegung zu verschiedenen Zeiten enthält (*AccountingMemDataEntry*):
 - *time*: die Systemzeit der Messung
 - *size*: die Speicherbelegung zur Zeit *time* in Byte
- *usedServices*: Eine Referenz auf ein Objekt der Oberklasse *java.util.Vector* (*AccountingServiceData*), das Informationen über die in Anspruch genommenen Dienstleistungen enthält (*AccountingServiceDataEntry*):
 - *provider*: der Dienstbringer der Dienstleistung
 - *service*: der Name der Dienstleistung
 - *price*: der Preis der Dienstleistung

Der Konstruktor sieht folgendermaßen aus:

- *AccountingDataEntry(agentName)*: erzeugt ein neues Objekt und setzt den Agentennamen

Folgende Methoden wurden implementiert:

- *addMCPTime(time)*: addiert *time* zur bisher verbrauchten CPU-Zeit
- *addMsg(direction)*: addiert je nach Richtung (*direction*) eine Nachricht auf die Anzahl der gesendeten oder empfangenen Nachrichten
- *addMsg(direction, size)*: wie oben, zusätzlich wird noch die Größe der Nachricht auf die Gesamtgröße addiert
- *addMsg(direction, size, sourceIP, destIP)*: wie oben, zusätzlich wird noch ein detaillierter Eintrag vorgenommen
- *addRPC(direction)*: addiert je nach Richtung (*direction*) einen RPC auf die Anzahl der erbrachten oder aufgerufenen RPCs
- *addRPC(direction, size)*: wie oben, zusätzlich wird noch die Größe der beim RPC übertragenen Daten auf die Gesamtgröße addiert
- *addRPC(direction, size, sourceIP, destIP)*: wie oben, zusätzlich wird noch ein detaillierter Eintrag vorgenommen
- *addMemoryInfo(size)*: Fügt die Speicherbelegung *size* mit der aktuellen Systemzeit in das *memoryUsage*-Objekt ein
- *addServiceInfo(provider, service, price)*: fügt eine Inanspruchnahme des Dienstes *service* mit dem Preis *price*, erbracht durch den Agenten *provider* zu den in Anspruch genommenen Diensten hinzu. Der Preis für die Dienstleistung wird dem *ServiceDictionary* mittels der Methode *priceOfService()* der Klasse *Location* entnommen.

Die *AccountingDataEntry*-Objekte werden ebenfalls in einer Vektorklasse (*AccountingData*) gesammelt. Diese Klasse stellt alle Methoden der Klasse *AccountingDataEntry* zur Verfügung, erweitert um den Parameter *agentName*.

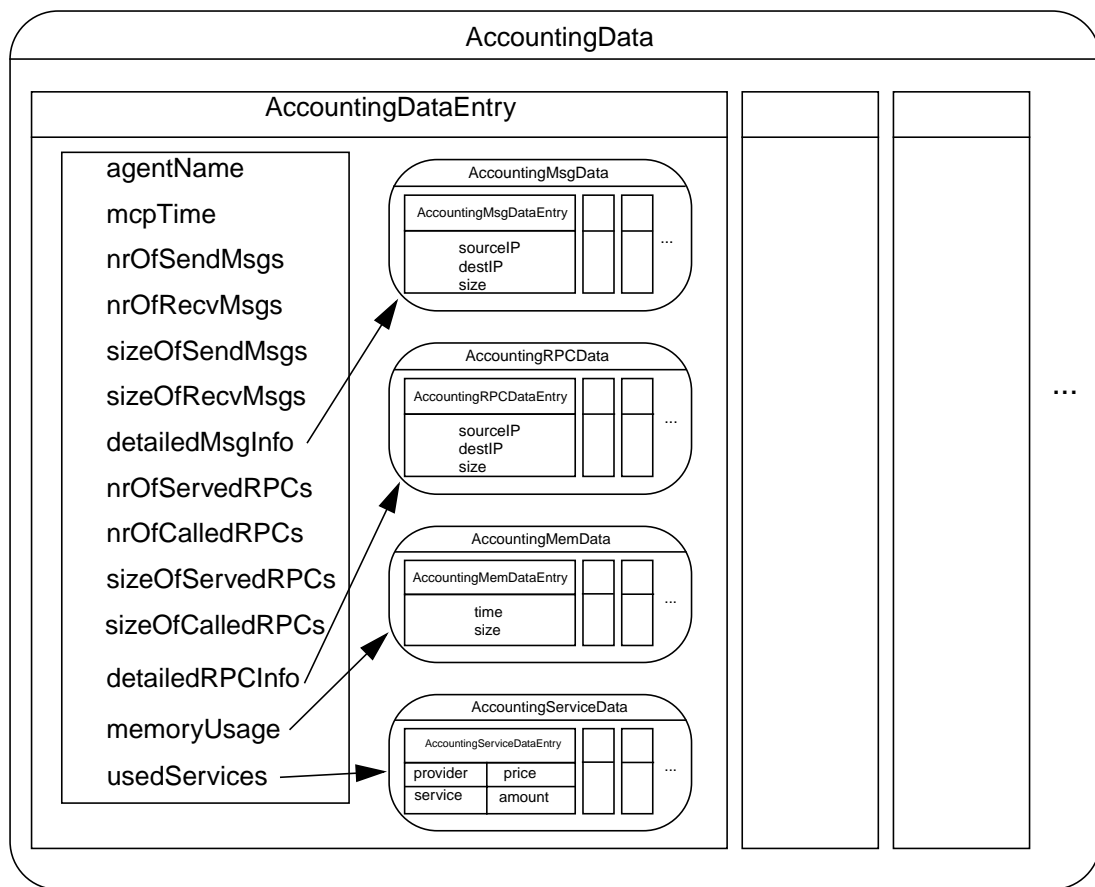


ABBILDUNG 15. Die Accounting-Datenstruktur

Beim Betreten der Location durch einen Agenten, wird für ihn ein Objekt der Klasse *AccountingDataEntry* instanziiert und im Vektor *AccountingData* eingetragen. Dies findet beim ersten Start der Location für die Agenten, die von der Location automatisch gestartet werden, in der Methode *startAgents()* der Klasse *Location* statt. Alle Agenten, die beim Start der Location nicht automatisch mitgestartet werden, durchlaufen die Methode *_arrive()* der Klasse *Location*, egal ob sie durch Migration oder Erzeugung durch einen anderen Agenten auf die Location gelangt sind. Für diese Agenten wird das *AccountingDataEntry*-Objekt dann in dieser Methode angelegt.

Beim Verlassen der Location wird das *AccountingDataEntry*-Objekt eines Agenten wieder gelöscht. Die Agenten können auf verschiedene Weise die Location verlassen. Sie können terminieren, terminiert werden oder auf eine andere Location migrieren. In allen drei Fällen wird die Methode *removeAgent()* der Klasse *Location* aufgerufen, die den Agenten aus dem System entfernt. In dieser Methode wird er dann auch vom Accounting abgemeldet.

Abfrage der Accounting-Daten:

Jedem Agenten ist es gestattet, über die Methode *getAccEntry()* der Klasse *Location* seine Accounting-Daten abzurufen. Dabei wird zuerst überprüft, welcher Agent der Besitzer des aktuellen Threads ist, um sicherzustellen, daß ein Agent nur seine eigenen Accounting-Daten bekommt. Damit der Agent die Accounting-Daten nicht abändern kann, wird sein Eintrag in der Accounting-Datenhaltung zunächst kopiert. Diese Kopie bekommt der Agent dann als Rückgabewert. Dies bedeutet auch, daß der Agent die Methode jedesmal erneut aufrufen muß, wenn er die aktuellen Daten wünscht.

5.2 Implementierung der Gebührenberechnungskomponente

5.2.1 Gebührenberechnung

Die Berechnung der Gebühren findet immer nur bei Bedarf statt, da die Berechnung, abhängig von der gewählten Konfiguration der Gebührenberechnung, komplex sein kann. Die Notwendigkeit der Aktualität der fälligen Gebühren besteht unter anderem bei Abrechnung, bei Abfrage der Gebühren durch den Agenten oder bei der Überprüfung, ob die Gebühren ein bestimmtes Limit überschritten haben.

Zur Berechnung der Gebühren wurde die Methode *computeFee()* der Klasse *Location* implementiert. Sie ruft abhängig von der gewählten Konfiguration die systemeigene Methode zur Gebührenberechnung (*computeFeeStandard()*) oder die vom Administrator bereitgestellte Methode (*computeFee()*), wenn sie das Interface *FeeComputer* implementiert, auf. Die berechneten Gebühren werden daraufhin in dem Accounting-Datenobjekt des jeweiligen Agenten gespeichert, das somit auch der Gebühren-Datenhaltung dient.

5.2.2 Gebühren-Datenhaltung

Die Accounting-Datenobjekte wurden so erweitert, daß sie auch die Gebühren-Daten der einzelnen Agenten aufnehmen. Dabei handelt es sich um folgende Klassenvariablen der Klasse *AccountingDataEntry*:

- *fees*: die Gesamtsumme der berechneten Gebühren
- *cpuFees*: die berechneten Gebühren für CPU-Zeit
- *memFees*: die berechneten Gebühren für Speicherverbrauch
- *msgFees*: die berechneten Gebühren für Messages
- *rpcFees*: die berechneten Gebühren für RPCs
- *serviceFees*: die berechneten Gebühren für Dienstleistungen
- *paidFees*: die bereits bezahlten Gebühren

Abfrage der angefallenen Gebühren durch die Agenten:

Die Abfrage der Gebühren erfolgt durch den Aufruf der Methode *getFees()* der Klasse *Location*. Diese Methode bestimmt den Agenten, dem der aktuelle Thread zugeordnet ist, und liefert die Gebührendaten (Gesamtsumme, Gebühren für CPU-Zeit, Gebühren für Speicher, Gebühren für Nachrichten, Gebühren für RPCs, Gebühren für Dienstleistungen, bereits bezahlte Gebühren) für diesen Agenten zurück.

5.2.3 Gebührenabrechnung

Die Methode *billAgent()* der Klasse *Location* sorgt für die Abrechnung der Gebühren eines Agenten.

Zuerst werden die aktuellen Gebühren neu berechnet (*computeFee()*). Von den aktuellen Gebühren werden die bereits bezahlten Gebühren abgezogen. Sollte das Zahlen einer gewissen Summe im Voraus konfiguriert sein (Parameter *<location>.BILLING.PREPAY* und *<location>.BILLING.SUM*), so wird dieser Betrag noch zu der zu bezahlenden Summe addiert. Ist der Gesamtbetrag größer als null, so sind abrechenbare Gebühren vorhanden. In diesem Fall wird dem Agenten eine Zahlungsaufforderung in Höhe der Gebühren mit einem Kommentar und einer Transaktionsidentifikation geschickt (*requestPayment()* der Klasse *PaymentObject*). Draufhin wartet die Methode auf das Eintreffen der Zahlung, indem sie periodisch die Höhe der zu zahlenden Gebühren und die Höhe der bezahlten Gebühren vergleicht. Sind sie identisch oder die Höhe der bezahlten Gebühren sogar größer als die Höhe der zu zahlenden Gebühren, so hat der Agent die Rechnung beglichen. Sollten unter den bezahlten Gebühren auch Gebühren für benutzte Dienstleistungen anderer Agenten gewesen sein, so werden diese Gebühren den Diensterbringern mittels der Methode *pay()* der *PaymentObject*-Klasse ausbezahlt.

Trifft die Zahlung nicht innerhalb der konfigurierten Maximalzeit (Parameter *<location>.BILLING.TIMEOUT*) ein, so wird zur Bestrafung des Agenten die Methode *punishDebtor()* der Klasse *Location* aufgerufen.

Je nach konfigurierter Bestrafungsstrategie (Parameter *<location>.BILLING.PENALTY*) wird der zahlungsunwillige Agent sofort terminiert oder auf seine Heimat-Location bzw. die Location, auf der er erzeugt wurde, zwangsmigriert. In beiden Fällen wird ein Eintrag in die Log-Datei gemacht.

Zur Bestimmung der Heimat-Location eines Agenten ist eine kleine Erweiterung der Klasse *Agent* notwendig. In der neuen Klassenvariablen *homeLocationName* wird beim ersten Betreten einer Location die Heimat-Location gesetzt. Über die Methode *getHomeLocationName()* der Klasse *Agent* kann die Heimat-Location dann abgefragt werden.

Wenn ein Agent zum letzten Mal abgerechnet wird, z.B. wenn er die Location verläßt oder terminieren will, wird ihm, falls seine bezahlten Gebühren über den zu zahlenden Gebühren liegen (z.B. durch Vorauszahlung), der Differenzbetrag in der Abrechnungsmethode über den Aufruf der Methode *pay()* der Klasse *PaymentObject()* zurückerstattet.

Synchronisation:

Während des Ablaufs einer Abrechnung mit einem Agenten dürfen dessen Gebühren nicht neu berechnet werden. Dies kann z.B. durch Abfrage der aktuellen Gebühren des Agenten selbst oder durch den periodischen Abrechnungsprozeß auftreten. Im ungünstigsten Fall werden dabei die zu zahlenden Gebühren erhöht, nachdem bereits die Zahlungsaufforderung an den Agenten gesendet wurde. Wenn der Agent nun die Zahlung tätigt, so sind seine zu zahlenden Gebühren immer noch höher als die bezahlten Gebühren. Nach Ablauf des Timeouts würde der Agent dann unberechtigter Weise bestraft werden.

Aus diesem Grund ist die Abrechnungsmethode auf das Accounting-Objekt (*AccountingDataEntry*) des jeweiligen Agenten synchronisiert, ebenso wie alle anderen Methoden, die die Neuberechnung der Gebühren eines Agenten (*computeFee()*) anstoßen. Auf diese Weise ist sichergestellt, daß sich die zu zahlenden Gebühren eines Agenten während des Abrechnungsvorganges mit ihm nicht ändern.

Eingehende Zahlungen:

Das *PaymentObject* teilt der Location das Eingehen einer Zahlung über den Aufruf der Methode *paymentReceived()* der Location mit. Die Location versucht daraufhin, die Zahlung über den Agentennamen des Zahlenden und der Transaktionsidentifikation folgendermaßen zuzuweisen:

1. Ist der zahlende Agent bekannt und wurde die verwendete Transaktionsidentifikation bei einer Zahlungsaufforderung an diesen Agenten benutzt, so wird diesem Agenten auch der eingezahlte Betrag gutgeschrieben.
2. Ist der zahlende Agent (A) nicht bekannt, die Transaktionsidentifikation aber einer Zahlungsaufforderung an einen anderen Agenten (B) zuzuordnen, so wird diesem Agenten (B) der gezahlte Betrag gutgeschrieben. Dies ermöglicht, daß andere Agenten Zahlungen eines bestimmten Agenten übernehmen.
3. Ist die verwendete Transaktionsidentifikation unbekannt, so wird der Betrag dem zahlenden Agenten, falls bekannt, gutgeschrieben.

Um diesen Mechanismus zu implementieren, ist es notwendig, alle bei Zahlungsaufforderungen benutzen Transaktionsidentifikationen mit dem dazugehörigen Agentennamen zu speichern. Dies geschieht im Accounting-Objekt des entsprechenden Agenten.

Ist die Zuordnung der Zahlung zu einem Agenten durchgeführt, so wird die bezahlte Summe im Accounting-Objekt des Agenten zu den bereits bezahlten Gebühren hinzugefügt. Die *billAgent()* Methode, die auf die Zahlung wartet, registriert dann beim nächsten Abfragen der zu zahlenden und der gezahlten Gebühren, daß die Zahlung eingetroffen ist.

5.2.4 Abrechnungszeitpunkt

Je nach Konfiguration (Parameter *<location>.BILLING.TIME*) findet die Abrechnung mit einem Agenten zu verschiedenen Zeitpunkten statt.

In allen Konfigurationen ist es notwendig, einen Agenten abzurechnen, bevor er wegmigriert oder terminiert. Zu diesem Zweck wird in den entsprechenden Methoden (Methode *die(...)*, *kill(...)* oder *goTo(...)* in *mole.Location*) eine Abrechnung mit dem Agenten angestoßen.

Ist nur eine einzige Abrechnung jedes Agenten bei Verlassen der Location erwünscht, so ist dies der einzige Zeitpunkt, an dem die Abrechnung ausgeführt wird.

Ist ein periodisches Abrechnen mit den Agenten konfiguriert, so wird in regelmäßigen Abständen die Abrechnung angestoßen. Dies geschieht in einem separaten Thread (Klasse *BillingThread*), der beim Start der Location erzeugt wird. Der Thread schläft in einer Endlosschleife die konfigurierte Zeitspanne und nimmt danach die Berechnung aller Gebühren und die Abrechnung mit allen Agenten vor.

Die dritte Möglichkeit für den Abrechnungszeitpunkt ist nach Anfallen einer konfigurierten Summe an Gebühren. Hierzu müssen nach jeder in Anspruch genommenen Dienstleistung und bei jeder Benutzung einer Ressource durch einen Agenten seine angefallenen Gebühren neu berechnet werden. Übersteigt die zu zahlende Summe den konfigurierten Betrag, so wird der Agent mittels der Methode *billAgent()* der Klasse *Location* abgerechnet.

Dieses Verfahren ist wegen der ständigen Neuberechnungen der Gebühren sehr Rechenzeit-intensiv und sollte nur bei Bedarf verwendet werden. Besonders die Ressource CPU-Zeit, die in Timeslices je 20 Millisekunden accountet wird, würde fast alle 20 Millisekunden eine Neuberechnung der Gebühren eines Agenten anstoßen. Um zumindest in diesem Fall etwas Abhilfe zu schaffen, wurde die Neuberechnung der Gebühren bei Änderung der benutzten CPU-Zeit auf maximal alle 10 Sekunden benutzter CPU-Zeit beschränkt. Desweiteren wurde die Berechnung der Gebühren für die einzelnen Ressourcen getrennt. Bei Benutzung einer Ressource werden nur die Gebühren für diese Ressource neu berechnet.

Implementiert wurde diese Variante in der Methode *billIfLimitReached()* der Klasse *Location*.

5.2.5 Abfrage der Gebührenstruktur

Die Methode *getPrices(locationname)* der Klasse *Location* kann von den Agenten aufgerufen werden, um die Preisstruktur einer beliebigen Location abzufragen. Diese Methode funktioniert nur für Locations, die die Gebühren über die systemeigenen Routinen berechnen. Die Rückgabe erfolgt in Form eines Arrays, das vier Elemente des Typs *Vector* enthält. Jeder Vektor repräsentiert die Gebührenstruktur für eine Ressource:

```
{<CPU-Gebühren-Vektor>, <Speicher-Gebühren-Vektor>,
 <Message-Gebühren-Vektor>, <RPC-Gebühren-Vektor>}
```

Der CPU-Gebühren-Vektor und der Speicher-Gebühren-Vektor enthalten Objekte der Klasse *MoleFeel*. Diese Klasse besitzt folgende Klassenvariablen:

- *long start*: Beginn des Intervalls, für das die folgenden Eigenschaften gelten.
- *long end*: Ende des Intervalls, für das die folgenden Eigenschaften gelten.

Das Intervall gibt an, für welche benutzten Einheiten einer Ressource die folgenden Eigenschaften gelten. Bei der Ressource CPU ist das CPU-Zeit (in ms), beim Speicher ist es belegter Speicher pro Zeit (in byte/ms), bei Messages sind es entweder Anzahl oder Gesamtgröße der Nachrichten (in byte), bei RPCs sind es ebenfalls Anzahl oder Gesamtgröße der RPC-Daten (in byte).

- *boolean mul*: Gibt an, ob der folgende Preis pauschal für das gesamte Intervall gilt (false) oder pro benutzter Einheit.
- *double price*: der Preis für das Intervall, entweder pauschal oder pro Einheit.
- *double lastprice*: Wenn der Preis des Intervalls pro Einheit gilt, steht in dieser Variablen, wieviel alle Einheiten bis zum Start dieses Intervalls kosten.
Bsp.: der Preis für eine Millisekunde CPU-Zeit zwischen 1000 und 2000 Millisekunden beträgt 0.01. Die Variable *lastprice* enthält den Gesamtpreis der ersten 1000 Millisekunden.

Der Message-Gebühren-Vektor und der RPC-Gebühren-Vektor enthalten Objekte der Klasse *MoleFee2*. Diese Klasse ist von der Klasse *MoleFee1* abgeleitet und enthält zusätzlich folgende Klassenvariablen:

- *boolean nr*: Gibt an, ob die Gebühren für Messages bzw. RPCs pro Anzahl (true) oder pro übertragenes Byte (false) berechnet werden.
- *String subnet*: gibt das Ziel-Subnetz der Messages bzw. RPCs an.
- *int mask*: gibt die Ziel-Subnetzmaske an.

5.2.6 Transaktionsidentifikation

Die Vergabe der eindeutigen Transaktionsidentifikationen wurde in der Klasse *Engine* implementiert (Methode *getId()*). Sie funktioniert analog zur Vergabe der eindeutigen Agentennamen durch die Engine folgendermaßen:

Die Transaktionsidentifikation hat den Syntax:

```
<fortlaufende Nr.>". "<Crash-Counter>". "<ip-adresse>". "<port>
```

Bei der fortlaufenden Nummer handelt es sich um einen Zähler, der bei jeder Vergabe einer Transaktionsnummer inkrementiert wird. Der Crash-Counter ist ein Zähler, der bei jedem Neustart der Engine inkrementiert wird und dauerhaft auf Festplatte hinterlegt wird. Übersteigt die fortlaufende Nummer die Maximalgröße des dazugehörigen Datentyps, so wird der Crash-Counter ebenfalls inkrementiert. Die letzten beiden Variablen sind die IP-Adresse und die Portnummer der Mole-Engine. Das Überlaufen der Crash-Counter Variable wird nicht abgefangen, trotzdem sollte diese Logik für den Zweck ausreichend sein.

5.3 Implementierung der Zahlungskomponente

Im Rahmen dieser Diplomarbeit wurde zu Test- und Demonstrationszwecken eine beispielhafte Zahlungskomponente implementiert. Diese besteht aus der *PaymentObject*-Klasse und einigen dafür notwendigen Modifikationen am System.

Die *PaymentObject* Klasse simuliert einen „token-based“ bzw. auch „cash-like“ genannten Typ eines Zahlungssystems, bei dem das Geld direkt in elektronischer Form vorhanden ist. Jedes *PaymentObject* hält also das Geld seines Agenten bzw. seiner Location in digitaler Form. Alle für den Zahlungsverkehr notwendigen Schnittstellenmethoden wurden implementiert (*getBalance()*, *requestPayment()*, *pay()*). Die Interaktion mit dem Agenten bzw. der Location erfolgt ebenfalls über die definierten Methoden (*paymentRequestReceived()*, *paymentReceived()*)

Die Implementierung dient ausschließlich Demonstrations- und Testzwecken. Sie enthält keine Mechanismen zum Schutz gegen Manipulation oder zur Verifikation des digitalen Geldes auf Gültigkeit. Es existiert auch keine Schnittstelle zu Banken oder anderen Institutionen, bei denen ein Wandel des digitalen Geldes in echtes Geld möglich wäre.

Die Kommunikation zwischen den einzelnen Payment-Objekten wird über Nachrichten abgewickelt. Nachrichten werden normalerweise von den Agenten empfangen und müßten dann von denselben an ihre Payment-Objekte weitergeleitet werden. Dieser Mechanismus könnte in der Nachrichtenempfängsmethode (*receiveMessage()*) jedes Agenten integriert werden. Diese Methode wird aber meistens von abgeleiteten Agentenklassen überschrieben, d.h. in jeder weiteren Agentenklasse müßte der Mechanismus zur Weiterleitung neu integriert werden, was sehr umständlich ist. Deswegen wird die Methode *dispatchMessage()* der Klasse *Agent*, die für den Aufruf der *receiveMessage()* Methode verantwortlich ist, so modifiziert, daß Nachrichten, deren Inhalt (*Message.content*) ein Objekt der Klasse *PaymentMsgContent* ist, nicht an die Nachrichtenempfängsmethode des Agenten (*receiveMessage()*), sondern an die Methode *receivePaymentMsg()* des jeweiligen Payment-Objektes, falls vorhanden, weitergeleitet werden (siehe Abbildung 16).

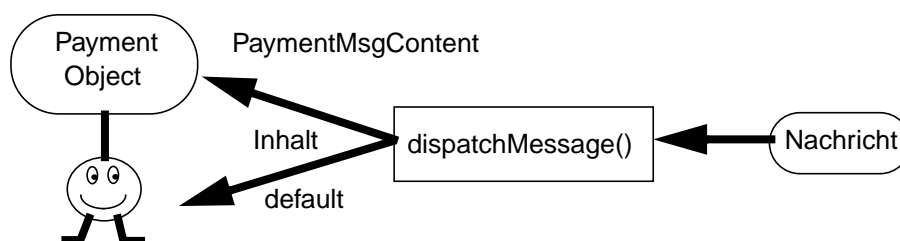


ABBILDUNG 16. Weiterleitung der Nachrichten

Da auch die Locations am Zahlungsverkehr teilnehmen sollen, muß die Zustellung der Zahlungsnachrichten auch an Locations sichergestellt sein.

Jede Location besitzt unter anderem zum Empfangen von Messages einen sogenannten *LocalAgent*. Dies ist ein spezieller Agent, der seiner Location gewisse Kommunikationsmöglichkeiten zur Verfügung stellt. Alle Messages für diesen *LocalAgent* werden an die Methode *automessage()* der Klasse *Location* weitergeleitet. Diese Methode wurde, wie die Methode *dispatchMessage()* der Agenten, so erweitert, daß Messages,

deren Inhalt Instanzen der Klasse *PaymentMsgContent* sind, an das Payment-Objekt der Location weitergeleitet werden.

Das Payment-Objekt ruft, wie spezifiziert, zwei Methoden ihres zugehörigen Objekts (Agent oder Location) auf (*paymentReceived()*, *paymentRequestReceived()*). Für die Location wurde, wie bereits im Kapitel 5.2.3 beschrieben, die Methode *paymentReceived()* implementiert. Zur Zeit ist nicht vorgesehen, daß eine Location eine Zahlungsaufforderung bekommt. Deswegen bleibt die Methode *paymentRequestReceived()* der Klasse *Location* erst einmal leer. Die zwei dazu analogen Methoden, die das zu einem Agenten gehörige Payment-Objekt beim Agenten aufruft, sind Aufgabe des Agentenprogrammierers und hängen in ihrer Funktionalität von der Anwendung ab.

6 Schnittstellenbeschreibung

Das folgende Kapitel beschreibt die zu Mole neu hinzugekommenen Schnittstellen für Entwickler zur Integration von Zahlungssystemen, für Administratoren zur Verwendung eigener Gebührenberechnungsmethoden und für Agentenprogrammierer zum Umgang mit dem Accounting, Billing und dem Zahlungssystem.

6.1 Schnittstellenbeschreibung für Zahlungssysteme

Um ein neues Zahlungssystem in Mole zu integrieren, muß die Klasse *PaymentObject* neu implementiert werden. Dabei müssen folgende Methoden existieren:

- Konstruktor *public PaymentObject(Object owner, Object initData):*

Der Konstruktor wird bei der Instanziierung der *PaymentObject*-Klasse von jedem am Zahlungsverkehr teilnehmenden Objekt aufgerufen. Dabei wird in der Variable *owner* das Besitzerobjekt des *PaymentObject* übergeben, dem später Zahlungen und Zahlungsaufforderungen signalisiert werden sollen. Die Variable *initData* kann ein Objekt beliebigen Typs sein, das für das Zahlungssystem wichtige Informationen enthält.

- *public double getBalance():*

Diese Methode muß das aktuelle Guthaben des Besitzers des *PaymentObject* zurückliefern.

- *public int requestPayment(AgentName payer, LocationName ln, double amount, String description, String id):*

Diese Methode soll dem Agenten mit dem Namen *payer*, der sich auf der Location mit dem Namen *ln* befindet, eine Zahlungsaufforderung der Höhe *amount* senden. Dabei kann in der Variablen *description* eine Beschreibung und in der Variablen *id* eine Transaktionsidentifikation übergeben werden.

Ein negativer Rückgabewert signalisiert einen Fehler.

- *public int pay(AgentName dest, LocationName ln, double amount, String description, String id):*

Diese Methode muß eine Zahlung tätigen. Die Geldsumme *amount* wird an den Agenten mit dem Namen *dest*, der sich auf der Location mit dem Namen *ln* befindet, bezahlt. Dabei kann in der Variablen *description* eine Beschreibung und in der Variablen *id* eine Transaktionsidentifikation übergeben werden.

Ein negativer Rückgabewert signalisiert einen Fehler.

Eingehende Zahlungsaufforderungen werden dem Besitzer-Objekt des *PaymentObject* durch Aufrufen der Methode

public void paymentRequestReceived(AgentName sender, LocationName senderloc, double amount, String description, String id)

signalisiert. Der Agent mit dem Namen *sender*, der sich auf der Location mit dem Name *ln* befindet, stellt eine Zahlungsforderung in Höhe von *amount*. Die Variable *description* enthält eine Beschreibung des Zahlungsgrundes, die Variable *id* enthält eine Identifikation der Transaktion.

Eine eingehende Zahlung wird dem Besitzer-Objekt des *PaymentObject* durch aufrufen der Methode

public void paymentReceived(AgentName sender, LocationName senderloc, double amount, String description, String id)

signalisiert. Der Agent mit dem Namen *sender*, der sich auf der Location mit dem Name *ln* befindet, hat den Betrag *amount* bezahlt. Die Variable *description* enthält eine Beschreibung des Zahlungsgrundes, die Variable *id* enthält eine Identifikation der Transaktion.

6.2 Schnittstellenbeschreibung für eigene Gebührenberechnung

Zur eigenen Gebührenberechnung muß eine neue Klasse implementiert werden bzw. eine vorhandene Klasse erweitert werden. Diese Klasse muß das Interface *FeeComputer* implementieren und eine Methode

public static double computeFee(AccountingDataEntry a)

besitzen, die als Rückgabewert die berechneten Gebühren liefert.

Beispiel:

```
public class MyFeeComputer extends Object implements FeeComputer
{
    public static double computeFee(AccountingDataEntry a)
    {
        return 5;
    }
}
```

Durch setzen des Parameters *<location>.BILLING* in der Konfigurationsdatei auf den Wert *MyFeeComputer* wird zur Gebührenberechnung die obige *computeFee*-Methode aufgerufen.

6.3 Schnittstellenbeschreibung für Agentenprogrammierer

Für Agentenprogrammierer existieren folgende neue Schnittstellen zum Agentensystem:

Dienstleistungen von Agenten (Klasse *Location*):

```
public void registerMeForService(AgentName provider, String aService, double price)
```

Diese Methode meldet eine Dienstleistung mit Preis bei einer Location an.

Beispiel:

```
getCurrentLocation().registerMeForService(getName(), "party service",  
                                         0.5);
```

Dieser Aufruf, der z.B. in der *start*-Methode eines Agenten stehen kann, meldet der Location, daß dieser Agent den Dienst „party service“ zur Verfügung stellt und pro Aufruf 0.5 Euro verlangt.

```
public void accountService(AgentName provider, AgentName user, String service)
```

Diese Methode wird aufgerufen, wenn eine Dienstleistung erbracht wurde. Die Location speichert daraufhin die Inanspruchnahme des Dienstes in der Accounting-Datenhaltung des Dienstbenutzers. Bei Abrechnung des Dienstbenutzers mit der Location wird die Gebühr für die Dienstleistung mitberechnet und danach dem Diensterbringer ausgezahlt.

Beispiel:

```
getCurrentLocation().accountService(getName(), serviceUser,  
                                    "party service");
```

Dieser Aufruf, der unmittelbar dem Quellcode zur Ausführung einer Dienstleistung folgen sollte, meldet dem Accounting die Benutzung seiner Dienstleistung „party service“ durch den Agenten *serviceUser*.

```
public double priceOfService(AgentName provider, String aService)
```

Über diese Methode kann der Preis einer bei der Location registrierten Dienstleistung erfragt werden.

Beispiel:

```
double price=getCurrentLocation().priceOfService(provider,  
                                                  "party service");
```

Dieses Statement weist der Variablen *price* den Preis für die Benutzung des Dienstes „party service“, der vom Agenten *provider* zur Verfügung gestellt wird, zu.

Abfrage der Accounting-Daten (Klasse *Location*):

```
public AccountingDataEntry getAccEntry()
```

Mittels dieser Methode, kann ein Agent seine Accounting-Daten bei der Location abfragen. Die Klassenvariablen der *AccountingDataEntry* Klasse sind im Kapitel 5.1.3 detailliert beschrieben.

Beispiel:

```
AccountingDataEntry ade=getCurrentLocation().getAccEntry();  
Engine.info("My CPU usage so far : " + ade.mcpTime);
```

Dieses Beispiel, das an einer beliebigen Stelle des Agentenquellcodes stehen kann, fragt die aktuellen Accountingdaten des Agenten von seiner Location ab und gibt die bisher verbrauchte CPU-Zeit aus.

Abfrage der Gebührenstruktur einer Location (Klasse *Location*):

```
public Vector[] getPrices(LocationName loc)
```

Diese Methode liefert die Gebührenstruktur einer Location zurück. Der Rückgabewert besteht aus einem Array, das 4 Elemente vom Typ *Vector* enthält. Der erste Vektor enthält die Gebühren für CPU-Zeit als Objekte der Klasse *MoleFee1*. Der zweite Vektor enthält ebenfalls Elemente des Typs *MoleFee1* und repräsentiert die Preise für die Speicherbenutzung. Die letzten beiden Vektoren enthalten Elemente der Klasse *MoleFee2* und geben Informationen über die Preise von Messages und RPCs. Die Variablen der Klassen *MoleFee1* und *MoleFee2* sind im Kapitel 5.2.5 detailliert beschrieben.

Beispiel:

```
Vector[] prices=getCurrentLocation().
    getPrices(getCurrentLocation().locationName());
Vector cpuPrices=prices[0];
Vector memPrices=prices[1];
Vector msgPrices=prices[2];
Vector rpcPrices=prices[3];
int cpuPricesMaxIndex;
int fees;
try
{
    cpuPricesMaxIndex = cpuPrices.indexOf(cpuPrices.lastElement());
}
catch (Exception e)
{
    cpuPricesMaxIndex = -1;
}
for (int i=0; i<= cpuPricesMaxIndex; i++)
{
    cpuFee = (MoleFee1) cpuFees.elementAt(i);
    if (300 <= cpuFee.end && 300 >= cpuFee.start)
    {
        if (cpuFee.mul) //price is per ms
        {
            fees=cpuFee.lastprice+((300 - cpuFee.start) + 1) *
                cpuFee.price);
        }
        else // price is for a range
        {
            fees=cpuFee.price;
        }
        break;
    }
}
Engine.info("300 ms of CPU-Time are " + fees + "Euros");
```

Dieses Beispiel holt sich die Gebührenstruktur der aktuellen Location und berechnet daraus den Preis für die Benutzung von 300 Millisekunden CPU-Zeit, indem es das Gebührenobjekt (*MoleFee1*) sucht, in dessen Intervall die 300 Millisekunden fallen. Dabei ist zu beachten, daß bei Gebühren, die pro Millisekunde berechnet werden

(*mul==true*), der Preis nur für den Teil des CPU-Verbrauchs gilt, der im Intervall des Gebührenobjektes liegt ($(300\text{-cpuFee.start})+1$). Die Gebühren für den Rest der CPU-Zeit stehen in der Variablen *lastprice*, die zu den Gesamtgebühren addiert wird.

Abfrage der Gebühren eines Agenten (Klasse *Location*):

```
public double[] getFees()
```

Diese Methode dient der Abfrage der entstandenen Gebühren eines Agenten. Der Rückgabewert ist ein Array von acht *double*-Werten, die der Reihe nach Folgendes bedeuten: ausstehende Gebühren, Gesamtsumme der Gebühren, bereits bezahlte Gebühren, Gesamtsumme der Gebühren für CPU-Zeit, Gesamtsumme der Gebühren für Speicherverbrauch, Gesamtsumme der Gebühren für gesendete und empfangene Nachrichten und Gesamtsumme der Gebühren für aufgerufene und geleistete RPCs.

Beispiel:

```
double[] fees=getCurrentLocation().getFees();
double notPaidFees=fees[0];
double totalFees=fees[1];
double paidFees=fees[2];
double cpuFees=fees[3];
Engine.info("I owe the location " + notPaidFees +
            " which should be the same as " + (totalFees-paidFees));
```

Dieses Beispiel fragt die Gebühren bei der Location ab und gibt die noch nicht bezahlten Gebühren aus.

Generierung einer eindeutigen Transaktionsidentifikation (Klasse *Engine*):

```
public static String getId()
```

Diese Methode liefert eine eindeutige Transaktionsidentifikation zurück.

Beispiel:

```
String id=getCurrentLocation().getId();
Engine.info("The ID " + id + "is system wide unique");
```

Dieses Beispiel holt sich von der Engine eine eindeutige Transaktionsidentifikation und gibt diese aus.

Benutzung der Zahlungskomponente (Klasse *PaymentObject*):

Bevor ein Agent die Methoden der Zahlungskomponente benutzen kann, muß er der Klassenvariablen *paymentObject* eine Instanz der Klasse *PaymentObject* zuweisen:

```
paymentObject=new PaymentObject(this,initData);
```

Danach können die im Kapitel 6.1 beschriebenen Methoden zur Interaktion mit dem Zahlungssystem benutzt werden.

Beispiele:

```
paymentObject.pay(newAgent, ln, 100, "Begrüßungsgeld",
                 Engine.getId());
```

Dieser Aufruf führt eine Zahlung in Höhe von 100 Euro an den Agenten *newAgent* auf der Location *ln* mit der Beschreibung „Begrüßungsgeld“ und einer neuen Transaktionsidentifikation durch.

```
paymentObject.requestPayment (AgentName.la(),
                               getLocation().locationName(),
                               "Begrüßungsgeld", Engine.getId());
```

Dieser Aufruf sendet eine Zahlungsaufforderung in Höhe von 100 Euro an die aktuelle Location (über den *localAgent* der Location) mit der Beschreibung „Begrüßungsgeld“ und einer neuen Transaktionsidentifikation. Das Beispiel ist rein fiktiv. Eine Location würde auf diese Forderung in der jetzigen Implementation nicht reagieren.

```
public void paymentRequestReceived (AgentName sender, LocationName ln,
                                    double amount, String description,
                                    String id)
{
    Engine.info("Agent " + sender.toString() + " wants " + amount +
               " Euros from me for " + description);
}
```

Dies ist eine Beispielmethode, die bei eingehenden Zahlungsaufforderungen ausgeführt wird. Sie gibt lediglich die Zahlungsaufforderung aus.

```
public void paymentReceived (AgentName sender, LocationName ln,
                             double amount, String description,
                             String id)
{
    message(new Message (getName(),
                        getLocation().locationName(),
                        sender, ln, 0, "Thanks for the money"));
}
```

Dies ist ein Beispiel für die Methode, die bei eingehenden Zahlungen aufgerufen wird. Sie bedankt sich nach Eingang einer Zahlung beim zahlenden Agenten mit einer Nachricht.

7 Messungen

Um den Performanzverlust, der mit dem Accounting und Billing einhergeht, ungefähr zu bestimmen, wurden Messungen am erweiterten Mole-System mit verschiedenen Konfigurationen sowie am unmodifizierten Mole (Version 3.0) durchgeführt. Hierzu wurde ein Testagent geschrieben, der u.a. 100 RPCs aufruft und 100 Messages verschickt. Der Testagent sendet sowohl die RPC-Aufrufe als auch die Messages an sich selbst, d.h., daß jede Message und jeder RPC als abgehend/gerufen und ankommend/geleistet accountet wird. In der RPC-Methode werden sehr große Fakultäten berechnet. Nach Beendigung gibt der Testagent seine Gesamtlaufzeit aus.

Der Quelltext des Agenten befindet sich im Anhang A. Das Testsystem bestand aus einem PC mit Intel Pentium II (300 MHz) Prozessor und dem Betriebssystem Linux 2.2. Während der Tests arbeiteten keine weiteren Benutzer am System. Die einzelnen Messungen wurden mehrfach durchgeführt.

Es folgen die gemessenen durchschnittlichen Laufzeiten des Testagenten:

Unmodifiziertes Mole (Version 3.0)	25,2 s
Erweitertes Mole, Accounting und Billing in der Konfiguration ausgeschaltet	25,2 s
Erweitertes Mole, nur CPU-Accounting	25,3 s
Erweitertes Mole, nur Speicher-Accounting (jede Sekunde)	25,6 s
Erweitertes Mole, nur Speicher-Accounting (alle 0.1 Sekunden)	26,9 s
Erweitertes Mole, nur Message-Accounting (maximale Informationen, 100 verschickte + 100 empfangene Messages, Größe des Inhalts jeweils 10 Byte)	26,0 s
Erweitertes Mole, nur Message-Accounting (maximale Informationen, 100 verschickte + 100 empfangene Messages, Größe des Inhalts jeweils 50 KByte)	70,0 s
Erweitertes Mole, nur RPC-Accounting (maximale Informationen, 100 gerufene + 100 geleistete RPCs)	25,3 s
Erweitertes Mole, komplettes Accounting (Speicher-Accounting jede Sekunde, Größe des Message-Inhaltes : 10 Byte)	27,2 s
Erweitertes Mole, komplettes Accounting (s.o.), Billing bei Verlassen der Location	27,2 s
Erweitertes Mole, komplettes Accounting (s.o.), Billing alle 5 Sekunden	27,4 s
Erweitertes Mole, komplettes Accounting (s.o.), Billing jede Sekunde	27,7 s
Erweitertes Mole, komplettes Accounting (s.o.), Billing bei Überschreitung eines konfigurierten Betrages	27,4 s

Ergebnisse:

- Bei ausgeschaltetem Accounting und Billing hat der Testagent dieselbe Laufzeit wie in einem Mole-System ohne Abrechnungskomponente.
- Ein vollständiges Accounting (Speicher-Accounting jede Sekunde) bei nicht zu großen Messages verringert die Performanz nur um 8 %.
- Das Accounting von großen Messages bremst das System stark. Dies gilt ebenfalls für RPCs mit großen Übergabe- oder Rückgabeobjekten sowie für das Speicheraccounting von großen Agenten.

Dies liegt an der Größenbestimmung von Objekten über die Serialisierung, die in allen drei Fällen zum Einsatz kommt. Messungen auf dem Testsystem haben ergeben, daß die Serialisierung von einem Kilobyte ganz grob 5 Millisekunden in Anspruch nimmt. Dieser Wert ist natürlich abhängig von Rechnerarchitektur und Java-Implementierung. Die Speichermessung eines Agenten mit 5 Kilobyte dauert dann ca. 25 Millisekunden, ebenso wie das Accounten einer Message der Größe 5 Kilobyte.

- Der komplette Abrechnungsvorgang mit Gebührenberechnung (systemeigene Methoden) geht sehr schnell vonstatten (ca. 20 ms).
- Entgegen den Erwartungen belastet das Abrechnen bei Ansammlung einer gewissen Summe das System im Vergleich zum Abrechnen bei Verlassen der Location kaum (< 1 %). Zur Erinnerung: Hier müssen bei jedem Zugriff auf einen Dienst oder eine Ressource die kompletten Gebühren neu berechnet werden.

8 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde eine Abrechnungskomponente entwickelt, die das Mole-System um diverse wichtige Fähigkeiten erweitert.

Es wurde eine Accounting-Komponente in Mole integriert, die auf Agentenebene verbrauchte Ressourcen und benutzte Dienste mitprotokolliert.

Eine weitere Komponente berechnet aufgrund der vom Accounting gelieferten Daten die Gebühren für Ressourcen- und Dienstinanspruchnahmen und rechnet diese mit den Agenten ab. Dieses geschieht unter Zuhilfenahme der neu definierten und prototypisch implementierten Schnittstellen zu einem Zahlungssystem.

Die durch die neu integrierten Komponenten auftretenden zusätzlichen Systembelastungen sind dabei durchaus akzeptabel.

Das entworfene und implementierte System bietet aber immer noch Erweiterungs- und Änderungsmöglichkeiten, die in Zukunft realisiert werden können.

Sollte sich die Mole-Projektgruppe entschließen, bei der Weiterentwicklung des Agentensystems Mole eine eigene unabhängige Java Virtual Machine zu verwenden, so würde sich eine komplette Reimplementierung der Accounting-Komponente anbieten, bei der ein Großteil der Accounting-Mechanismen in die Virtuellen Maschine integriert sind. Die Java-Systemklassen müßten dann so modifiziert werden, daß sie mit den Accounting-Mechanismen der Virtuellen Maschine kooperieren. Auf diesem Wege wäre ein effizienteres und flächendeckendes Accounting in Mole möglich.

Eine sinnvolle Erweiterung im Bereich der von Agenten zur Verfügung gestellten Dienstleistungen wäre die vollständige Vermittlung dieser Dienste durch das System. Ein Agent, der einen Dienst benutzen möchte, wendet sich mit seinem Wunsch an das System. Dieses vermittelt den Wunsch an einen geeigneten Dienstbringer. Darüber hinaus kümmert sich das System dann darum, daß der Dienst sowohl ordnungsgemäß erbracht wird als auch ordnungsgemäß bezahlt wird. Hiermit könnte die Problematik der nicht bezahlten, aber geleisteten Dienste sowie die Problematik der bezahlten, aber nicht geleisteten Dienste gelöst werden.

Anhang

Anhang A: Quellcode des für die Messung verwendeten Testagenten

Der Quelltext des Testagenten dient zum einen zur Verdeutlichung des Meßvorganges und zum anderen als Beispiel zum Umgang mit den neuen Funktionalitäten.

```
/**
 * AccTester.java
 * an agent for testing and measurement
 *
 * System: Java JDK 1.1
 * Autor: traenksn@orgel.informatik.uni-stuttgart.de
 * Datum: 5.6.99
 *
 */

package mole.apps;

import mole.*;
import java.util.*;
import java.io.*;

public class AccTester
    extends UserAgent implements MobileAgent
{
    boolean done=false;
    int counter=0;

    // -----
    // constructor
    // -----
    public AccTester()
    {
        super("Accounting tester and measurer");
        /* initialize paymentObject */
        paymentObject = new PaymentObject(this,new Double(1000000));
    }

    public boolean init(Hashtable parameters)
    {
        // no arguments needed
        if( parameters.isEmpty() )
            return true;
        else
            return false;
    }

    // -----
    // start
    // -----
    public void start()
    {
        int i=0;
        Engine.info ("AccTester: awaking at " +
                    getLocation().locationName());
        long currentTime1 = System.currentTimeMillis();
    }
}
```

```

Long ret = new Long(0);
for (i=1; i <= 100; i++)
{
    try
    {
        ret = (Long) call("rpcFac", getName(),
                        getCurrentLocation().locationName(),
                        new Integer(i));
    }
    catch (Exception e)
    {
        Engine.error("AccTester: exception " + e );
    }
    message(new Message(getName(),
                        getCurrentLocation().locationName(),
                        getName(),
                        getCurrentLocation().locationName(),
                        0, new String("Faculty123")));
}
while (counter<100)
{
    try
    {
        Thread.sleep(100);
    }
    catch (Exception e) {}
}
try
{
    Thread.sleep(1000);
}
catch (Exception e) {}
Engine.info ("AccTester: dieing ");
long currentTime2 = System.currentTimeMillis();
System.out.println("Overall Time in millis = " +
                  (currentTime2 - currentTime1));
die();
}
// -----
public void receiveMessage(Message m)
{
    counter++;
    Engine.info("AccTester: received Message : "+counter);
}
public void paymentRequestReceived(AgentName sender, LocationName ln,
                                   double amount, String description,
                                   String id)
{
    if (sender.equals(AgentName.la()) &&
        getCurrentLocation().locationName().equals(ln) &&
        description.startsWith("location fees"))
    {
        Engine.info("AccTester: paying fees -> " + amount);
        paymentObject.pay(sender, ln, amount, description, id);
    }
}

```

```

}
public void paymentReceived(AgentName sender, LocationName ln,
                           double amount, String description,
                           String id)
{
    Engine.info("AGENT: got payment " + amount + " for "+description);
}
public Long rpcFac(Integer x)
{
    long lreturn=1;
    for (int i=2 ; i <=x.intValue()*10000 ; i++)
    {
        lreturn*=i;
    }
    return new Long(lreturn);
}
}
}

```

Anhang B: Quellcode der Klasse Accounting

Die Klasse *Accounting* beinhaltet die Funktion zur Berechnung des durch ein Objekt belegten Speichers mittels der Serialisierung.

```

/*****
 * Accounting.java
 *
 * duty: methods for accounting
 *
 * created: 18.3.99
 * Author: traenksn@orgel.informatik.uni-stuttgart.de (Sven Traenkle)
 *
 *****/

package mole;

import java.lang.*;
import java.io.*;

public class Accounting extends Object
{
    public Accounting()
    {
        super();
    }

    public static long sizeof(Object obj)
    {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oas = null;
        try
        {
            oas = new ObjectOutputStream(baos);
        }
        catch (Exception e)
        {
            System.err.println("Error opening ObjectOutputStream : " +
                               e.toString());
        }
    }
}

```

```
        new Exception("Accounting");
    }
    try
    {
        oas.writeObject(obj);
    }
    catch (Exception e)
    {
        System.err.println("Error writing Object : " + e.toString());
        new Exception("Accounting");
    }
    return (long) baos.size();
}
}
```

Literaturverzeichnis

[AJS97]

N. Asokan, P. Janson, M. Steiner, M. Waidner. *Electronic Payment Systems*. 211ZR019, IBM Zurich Research Laboratory, April 1997, final version: IEEE COMPUTER, September 1997.
<http://www.semper.org/info/index.html#211ZR019>

[BKR98]

J. Bredin, D. Kotz, D. Rus. *Market-based Ressource Control for Mobile Agents*. In: Proceedings of „Autonomous Agents“, pp. 197-204. Association of Computing Machinery (ACM). May 1998.

[BY98]

S.G. Belmon, B.S. Yee. *Mobile Agents and Intellectual Property Protection*. In: Proceedings of the Second International Workshop on Mobile Agents (MA'98), K. Rothermel und F. Hohl (Eds.), Lecture Notes in Computer Science Nr. 1477, pp. 172-182. Springer-Verlag. 1998.

[CE97]

G. Czajkowski, T.v. Eicken. *JRes: a resource accounting interface for Java*. In: OOPSLA '98. Proceedings of the conference on Object-oriented programming, systems, languages, and applications. pp 21-35. ACM SIGPLAN Notices Vol. 33, No. 10. Oktober 1998.

[DB98]

Deutsche Bank. *ecash*.
Produktionformationsbroschüre. März 1998.

[Du89]

Das neue Dudenlexikon. Meyers Lexikonredaktion 1989.

[Ec98]

B. Eckel. *Thinking in Java*.
Prentice Hall. März 1998.

[Fo93]

L.N. Foner. *What's An Agent ? A Sociological Case Study*. Agents Memo 93-01. Agent Group, MIT Media Lab. 1993.

[GK94]

M.R. Genesereth, S.P. Ketchpel. *Software Agents*.
In: Communications of the ACM. June 1993. Vol.36, No.4. pp.67-77.

[GKC98]

R.S. Gray, D. Kotz, G. Cybenko, D. Rusniela. *D'Agents: Security in a Multiple-Language, Mobile-Agent System*.
In: Giovanni Vigna (Ed.). *Mobile Agents and Security*. pp 154-187. Springer-Verlag. 1998.

[GM94]

General Magic. *Telescript Technology: The Foundation for the Electronic Marketplace*.
General Magic White Paper. 1994.

[Ho95]

F. Hohl. *Konzeption eines einfachen Agentensystems und Implementation eines Prototyps*.

Universität Stuttgart, Fakultät Informatik. Diplomarbeit Nr. 1267. 1995.

[Ho96]

F. Hohl. *Mole Alpha 1.0 Documentation*.

Universität Stuttgart, Abteilung Verteilte Systeme des IPVR. 1996.

[Ku97]

W. Kuhn. *Ressourcenkontrolle in einem Mobile-Agenten-System*.

Universität Stuttgart, Fakultät Informatik. Diplomarbeit Nr. 1468. 1997.

[Ou95]

J. Ousterhout. *Scripts and Agents: The New Software High Ground*.

Invited Talk, USENIX Conference 1995.

[Sc97]

B. Schoenmakers. *Basic Security of the ecashTM Payment System*.

In: *Computer Security and Industrial Cryptography: State of the Art and Evolution*. ESAT Course, Leuven, Belgium, 1997. Lecture Notes in Computer Science. Springer Verlag. 1997.

[Se94]

T. Selker. *Coach: A Teaching Agent that Learns*.

In: *Communications of the ACM*. July 1994. Vol.37, No.7. pp.92-99.

[SFE97]

R. Schuster, J. Färber, M. Eberle. *Digital Cash*.

Springer Verlag. 1997.

[We98]

R. Weber. *Chablis - Market Analysis of Digital Payment Systems*.

Version 1.0. TU Munich. Technical Report TUM-I9819. August 1998.

[WJ95]

M.J. Wooldridge, N.R. Jennings. *Intelligent Agents*.

ECAI-94 Workshop on Agent Theories, Architectures and Languages. Amsterdam, The Netherlands August 8-9, 1994.

Proceedings. Springer 1995.

ERKLÄRUNG

Ich versichere, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

(Sven Tränkle)