UNIVERSITÄT PASSAU

Diplomarbeit

Migration und Sicherheit in AutO, einem verteilten System autonomer Objekte

Stefan Seltzsam
Eduard-Hamm-Straße 14
94036 Passau

Erstgutachter:	Prof. Alfons Kemper, Ph. D.
Zweitgutachter:	Prof. Christian Lengauer, Ph. D.
Betreuerin:	Dipl. Inform. Natalija Krivokapić

8. Januar 1999

Prof. A. Kemper, Ph. D.
Lehrstuhl für Dialogorientierte Systeme
Fakultät für Mathematik und Informatik
Universität Passau

Zusammenfassung

AutO ist ein verteiltes System autonomer Objekte, das ein ACID-konformes Transaktionssystem, Persistenz und Recovery bietet. Sicherheit ist in AutO in Form von rollenbasierter Autorisierung und sicherer Kommunikation gegeben. Durch die Migration von Objekten kann sich das System dynamisch an Lastverschiebungen anpassen und somit Kommunikationskosten einsparen. Da AutO vollständig in Java implementiert wurde, kann es auch in heterogenen Systemen wie dem Internet eingesetzt werden.

Diese Diplomarbeit betrachtet verschiedene Aspekte der Migration in AutO. Neben dem Vergleich zweier Migrationsstrategien im Hinblick auf Rechenzeit und Kommunikationskosten wurde auch die Anbindung eines verteilten Nameservice an AutO realisiert. Das Hauptaugenmerk dieser Arbeit liegt auf der Erweiterung der vorhandenen Sicherheitskomponente um Konzepte, die sich der Sicherheitsaspekte der Migration annehmen. Den Schutz der Rechner, auf denen autonome Objekte laufen, übernehmen nun ein spezieller ClassLoader und ein SecurityManager. Um Objekte daran zu hindern, auf unsichere Rechner zu migrieren, wurde eine Kategorisierung der Rechner und Objekte vorgenommen. Diese wird dazu verwendet, festzustellen, ob ein Rechner für ein Objekt sicher genug ist oder nicht. Eine weitere Aufgabenstellung dieser Diplomarbeit war es, eine Beispielanwendung für AutO zu entwickeln. Diese wurde in Form eines einfach gehaltenen Support-Centers realisiert, das durch eine Visualisierungskomponente Einblick in die internen Arbeitsabläufe gewährt.

Ein großes Dankeschön...

an Frau Natalija Krivokapić und Professor Alfons Kemper für die hervorragende Betreuung dieser Diplomarbeit...

an die Mannschaft des AutO–Teams — Stefan Grießer, Markus Islinger, Natalija Krivokapić, Stefan Pröls und Markus Keidl — für die tatkräftige Unterstützung beim Entwurf und der Implementierung von AutO...

an Herrn Markus Keidl, der bei den vielen Diskussionen nie mit kritischen Bemerkungen sparte...

an Frau Natalija Krivokapić, Herrn Markus Keidl, Herrn Bernhard Stegmaier und Herrn Bernhard Zeller für das Korrekturlesen dieser Arbeit...

an Frau Michaela Reitberger für die vielen aufmunternden Worte...

an meine „Leidensgenossen" im Rechnerraum, die die Diskussionen über AutO meistens ohne Murren ertrugen...

und schließlich an das Wahn's Inn, das nach arbeitsreichen Tagen immer wieder für die nötige Entspannung sorgte.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Mit zunehmender weltweiter Vernetzung wächst auch die Verbreitung verteilter Datenbanken. Diese ermöglichen es, sehr große, auf mehrere Rechner verteilte Datenmengen konsistent zu verwalten. Um trotz der Verteilung effizient arbeiten zu können, ist es wichtig, Daten möglichst dort zur Verfügung zu stellen, wo sie am häufigsten gebraucht werden. In vielen realen Anwendungsfällen ist es nicht ausreichend, die Daten statisch zu positionieren, sondern es ist nötig, daß die Positionierung der Daten von der Datenbank selbst dem Zugriffsverhalten dynamisch angepaßt wird. Ermöglicht wird eine derartige „Selbstoptimierung“ des Systems durch die Integration einer Migrationskomponente, die die Repositionierung der Daten nach einer festgelegten Strategie vornimmt und dadurch die innerhalb des Systems anfallenden Kommunikationskosten reduziert. Natürlich entstehen bei der Migration selbst Kommunikationskosten durch das Versenden der Daten. Die Strategie muß also so ausgelegt sein, daß die durch eine Migration verursachten Kosten im Normalfall viel geringer sind als die Einsparungen, die dadurch erzielt werden.

Ein Anwendungsszenario einer derart verteilten Datenbank ist eine weltweit agierende Firma, die ihre Unternehmensdaten nicht lokal für jede Filiale, sondern global in einer einzigen verteilten Datenbank verwalten will. Die Nützlichkeit und Notwendigkeit der Migration wird hier deutlich: Zum einen können Objekte, die häufig global referenziert werden, dem natürlichen Tag/Nacht-Zyklus der Sonne folgen. Dadurch tragen sie dazu bei, die Kommunikationskosten im System gering zu halten, da sie sich immer in der Zeitzone aufhalten, in der Angestellte der Firma gerade arbeiten. Zum anderen können Objekte, die eher lokal verwendet werden, auf einen Rechner migrieren, auf dem sie geringere Kommunikationskosten verursachen und dort verweilen, da dies einen schnellen Zugriff auf diese Objekte ermöglicht. Damit stehen filialspezifische Daten lokal zur Verfügung, während Daten, die von allen Filialen gleichermaßen referenziert werden, stets so positioniert werden, daß sie von den „aktiven“ Filialen mit möglichst geringen Kommunikationskosten verwendet werden können.

Wenn ein Datenbanksystem für eine solche Anwendung eingesetzt wird, ist allerdings nicht nur die Effizienz des Systems wichtig, auch die Sicherheit der Daten vor unbefugten

Zugriffen muß gewährleistet sein: Daten dürfen weder unerlaubt eingesehen, noch manipuliert werden können. Deshalb erfordert ein solches System eine – im Vergleich zu einem System ohne Migration – leistungsfähigere Sicherheitskomponente, um die Migration nicht zum Sicherheitsrisiko werden zu lassen. Mobiler Code, wie ihn migrierende Objekte darstellen, erfordert einerseits, daß man das Datenbanksystem vor unerlaubten Zugriffen und Manipulationen durch die Objekte schützt, andererseits müssen auch die Objekte davor geschützt werden, auf Rechner zu migrieren, auf denen sie nicht sicher sind.

Ein Beispiel eines verteilten Datenbanksystems, das Migration unterstützt, ist AutO, ein System autonomer Objekte. AutO verfügt über sichere Kommunikation und ein rollenbasiertes Autorisierungsverfahren [Kei99], um den Zugriff auf die Objekte innerhalb des Systems kontrollieren zu können. Es bietet aber keine Möglichkeit, den durch die Migration entstehenden Sicherheitsproblemen entgegenzuwirken.

1.2 Aufgabenstellung

Das AutO-System ist ein persistentes, verteiltes System autonomer Objekte, das vollständig in Java [AG98] implementiert ist. Es bietet eine leistungsfähige Migrationskomponente und ein Sicherheitskonzept, das den Zugriff auf Objekte innerhalb des Systems regelt.

Ziel dieser Arbeit ist es, verschiedene Aspekte der Migration in AutO zu untersuchen und das existierende System um eine Komponente zu erweitern, die sich der spezifischen Sicherheitsprobleme, die durch die Migration verursacht werden, annimmt. Die erste Aufgabe besteht darin, zwei verschiedene Migrationsstrategien vorzustellen und zu vergleichen. Um die Effizienz des Systems zu steigern, soll außerdem ein verteilter Nameservice an das System angebunden werden, der an die Stelle des bisher verwendeten zentralen Nameservers treten soll. Den zentralen Punkt dieser Arbeit bildet die Erweiterung der bestehenden Sicherheitskomponente von AutO um Konzepte, die es erlauben, auch Migration zu nutzen, ohne Abstriche in der Sicherheit des Systems hinnehmen zu müssen. Die abschließende Aufgabe ist die Realisierung einer Beispielanwendung für das AutO-System.

1.3 Überblick

Der Rest dieser Arbeit ist wie folgt gegliedert: Im zweiten Kapitel wird das AutO-System beschrieben, um einen Einblick in dessen Arbeitsweise zu gewinnen. In Kapitel 3 werden dann zwei verschiedene Migrationsstrategien genau untersucht und miteinander verglichen. Das darauffolgende Kapitel behandelt die Integration eines verteilten Nameservice in das AutO-System. Kapitel 5 beschäftigt sich mit der existierenden Sicherheitskomponente von AutO und beschreibt deren Erweiterung, die auch Sicherheitsaspekte der Migration berücksichtigt. Eine Beispielanwendung des AutO-Systems, die auch die Visualisierung der Vorgänge im System erlaubt, behandelt Kapitel 6. Den Schluß dieser Arbeit bilden eine Zusammenfassung und ein kurzer Ausblick.

Kapitel 2

Einführung in AutO

Diese Arbeit basiert auf AutO, dem das in [KLMW94] vorgestellte Modell autonomer Objekte zugrunde liegt. Durch die Erweiterung dieses Modells um Datenbankkonzepte wie Persistenz, Recovery und ein Transaktionssystem, wurde AutO zu einem persistenten, objektorientierten und verteilten System autonomer Objekte ausgebaut. Außerdem wurde eine Migrationskomponente in AutO integriert, die es Objekten erlaubt, sich im System dynamisch zu positionieren und dadurch die Kommunikationskosten zu senken.

Dieses Kapitel gibt einen kurzen Einblick in die Arbeitsweise von AutO, soweit dies für das Verständnis der Arbeit erforderlich ist. Eine genauere Beschreibung findet sich in [Gri97] und [Isl97]. Verschiedene in AutO integrierte Kontrollmechanismen für verteilte Objektbanken werden in [Kri98] detailliert beschrieben.

2.1 Ein Modell autonomer Objekte

Autonome Objekte besitzen wie Objekte im herkömmlichen, objektorientierten Sinn eine Struktur, die durch die Daten des Objektes repräsentiert wird, und ein Verhalten, das durch die Methoden, die das Objekt zum Zugriff und zur Manipulation seiner Struktur zur Verfügung stellt, definiert wird. Zusätzlich weisen autonome Objekte aber ein aktives Verhalten auf, das es ihnen ermöglicht, auf folgende Reize zu reagieren:

- Nachrichten, die Objekte von anderen Objekten oder Transaktionen erhalten,
- Zeitereignisse, also das Erreichen eines bestimten Zeitpunktes, und
- Veränderungen ihres eigenen Zustandes.

Das aktive Verhalten autonomer Objekte wird in einer sogenannten *behavioural map* beschrieben, die aus einer Menge von *Guards* besteht. Ein Guard kann dabei als *event-condition-action*-Regel (ECA-Regel) aufgefaßt werden. Tritt ein gewisses Ereignis ein, wird eine Aktion ausgeführt, wenn die Bedingung der entsprechenden Regel erfüllt ist. Einem Guard wird außerdem noch eine Priorität zugewiesen, die angibt, welchen Vorrang die Ausführung dieses Guards gegenüber anderen genießt.

In dem hier vorgestellten Modell gibt es drei Arten von Guards: on-Guards, if-Guards und at-Guards. On-Guards werden durch Nachrichten von anderen Objekten oder Transaktionen ausgelöst und beschreiben dadurch das Verhalten eines Objektes auf Stimulationen durch die Außenwelt. If-Guards dagegen können auf Veränderungen des internen Zustandes eines Objektes reagieren. Mit Hilfe von at-Guards kann ein Objekt zu bestimmten vorgegebenen Terminen beliebige Aktionen ausführen. At-Guards werden durch eine Timer-Komponente in AutO ausgelöst, die den Objekten zu den entsprechenden Zeitpunkten eine Nachricht schickt.

Die Nachrichten, die ein autonomes Objekt empfängt, werden in der Reihenfolge ihrer Ankunft in einer Warteschlange gespeichert. Die Guards, die durch diese Nachrichten ausgelöst werden können, werden als „aktiv“ gekennzeichnet. Guards, für die keine Nachrichten vorliegen, sind dadurch „passiv“ und brauchen nicht betrachtet zu werden. Da if-Guards nicht durch Nachrichten ausgelöst werden, sondern durch die Änderung des internen Zustandes eines Objektes, werden durch solche Zustandsänderungen alle if-Guards aktiviert. Wird später ein aktiver if-Guard ausgewählt, dessen Bedingung nicht erfüllt ist, wird er deaktiviert.

Das Objekt ermittelt nichtdeterministisch unter Berücksichtigung der Prioritäten einen aktiven Guard. Falls es sich dabei um einen on-Guard handelt, wird in der Warteschlange nach einer Nachricht gesucht, die den gewählten Guard auslösen kann. Existiert eine solche Nachricht oder handelt es sich nicht um einen on-Guard, wird die Aktion des Guards ausgeführt, sofern dessen Bedingung erfüllt ist.

2.2 Systemarchitektur

AutO wurde für den Einsatz in einem Wide-Area-Network (WAN) konzipiert. Da ein derartiges Netzwerk selten homogen ist, wurde bei der Entwicklung von AutO sehr auf Plattformunabhängigkeit geachtet. Aus diesem Grund wurde die Programmiersprache Java [AG98] verwendet. Die aktuelle Version des Systems ist somit auf allen Plattformen lauffähig, für die eine Java Virtual Machine (JVM) der Version 1.2 Beta 4 existiert. Mit der Verfügbarkeit der endgültigen Version des Java Development Kit (JDK) 1.2 wird auch dieses von AutO unterstützt werden.

Auf jedem Rechner im AutO-System – im folgenden als Knoten oder Node bezeichnet – muß ein *Knotenmanager* (Node-Manager) und mindestens ein *AutO-Prozeß* laufen, wie in Abbildung 2.1 dargestellt.

Der Knotenmanager eines Rechners kümmert sich um die Kommunikation mit anderen Rechnern, genauer gesagt mit deren Knotenmanagern. Der oder die AutO-Prozesse eines Knotens übernehmen die eigentliche Arbeit und verwalten eine Menge von autonomen Objekten und Transaktionen. Diese werden in AutO über logische Objektidentifikatoren (OIDs) eindeutig referenziert. OIDs betehen aus einer Seriennummer und dem Namen des Geburtsrechners, der nur dazu dient, eindeutige OIDs zu generieren. In AutO existieren verschiedene Arten von Prozessen:

- Generische AutO-Prozesse (`ObjectRepository`). Diese werden vom Knotenmana-

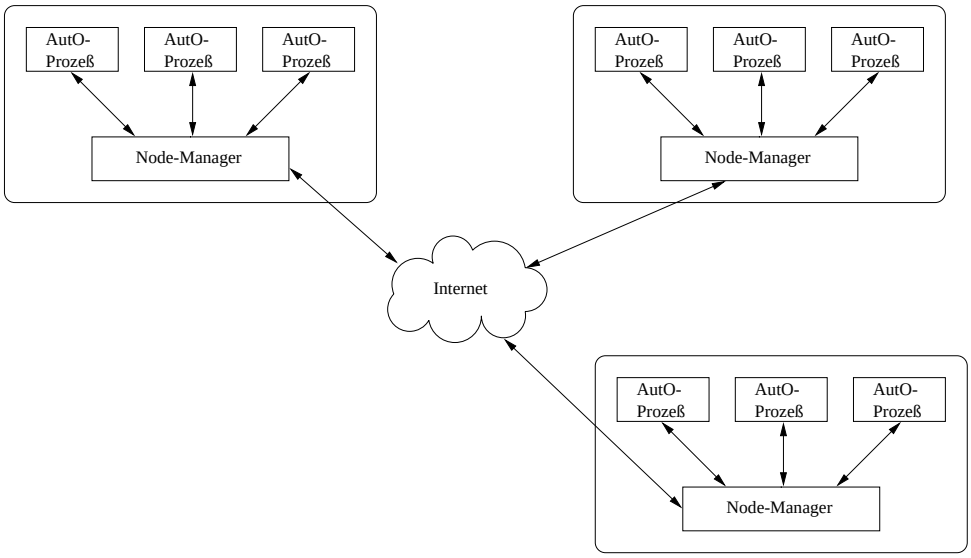


Abbildung 2.1: Schematischer Aufbau des AutO-Systems

ger bei Bedarf gestartet und dienen dazu, autonome Objekte und Transaktionen zu verwalten. Alle anderen Arten von AutO-Prozessen bieten zusätzliche Funktionalität.

- Interaktive Prozesse (z. B. `TerminalServer`). Diese Art von Prozessen bietet die Möglichkeit, das AutO-System interaktiv zu steuern. Der Terminal-Server bietet darüber hinaus die Möglichkeit, AutO über eine Netzverbindung mit der Hilfe von Terminal-Clients zu bedienen.

- Anwendungsspezifische Prozesse, die konkrete Anwendungen für das AutO-System ausführen.

Der Aufbau eines Prozesses ist schematisch in Abbildung 2.2 dargestellt.

Der *Communication-Agent* ist für die korrekte Weiterleitung von ankommenden und ausgehenden Nachrichten zuständig. Bei ausgehenden Nachrichten versucht der er zuerst, diese lokal durch den Object-Manager innerhalb des Prozesses zustellen zu lassen. Scheitert dieser Versuch, sendet er die Nachricht an den Knotenmanager, der sie entweder an einen anderen AutO-Prozeß oder zu einem anderen Knoten weiterleitet, in Abhängigkeit davon, wo sich das Empfängerobjekt befindet.

Der *Object-Manager* verwaltet eine Tabelle, die alle autonomen Objekte und Transaktionen dieses Prozesses beinhaltet. Er bietet Methoden an, um autonome Objekte oder Transaktionen zu erzeugen bzw. zu löschen und Objekte (aus der Datenbank) zu laden. Außerdem gehört die Zustellung von Nachrichten innerhalb des Prozesses zu seinen Aufgaben. Stellt er einem Objekt eine Nachricht zu, d. h. er schreibt sie in die Warteschlange des Objektes, wird diesem außerdem ein Thread zugewiesen, falls es noch keinen besitzt. Dieser übernimmt die Bearbeitung der Nachricht. Transaktionen besitzen stets zwei Threads:

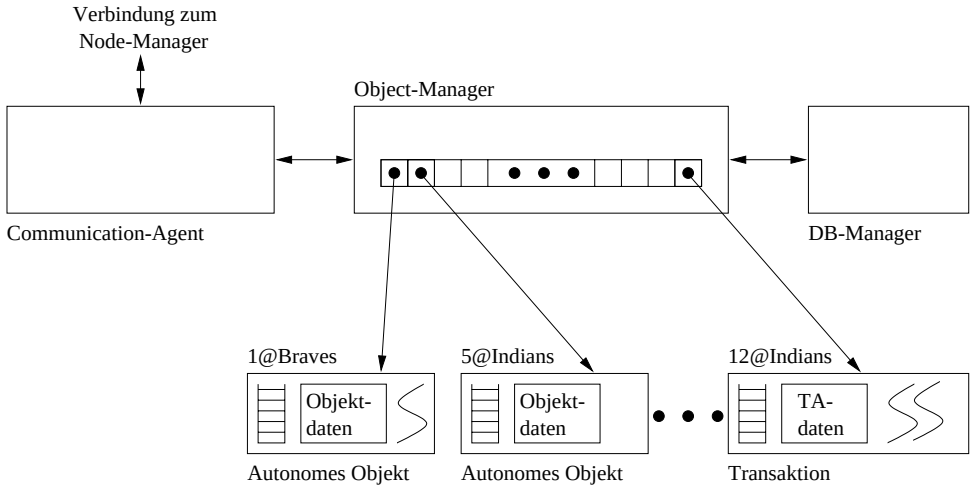


Abbildung 2.2: Aufbau eines AutO-Prozesses

einen für die eigentliche Arbeit der Transaktion und einen zweiten, der die Verwaltungsaufgaben der Transaktion wahrnimmt. Eine Beschreibung des in AutO implementierten Transaktionskonzeptes, das die ACID-Eigenschaften [KE96, KS91] erfüllt und auf semantischem Locking basiert, findet sich in [Kri97]. Verklemmungen werden durch *Deadlock Detection Agents* (DDAs) [KKG96, Prö97] aufgelöst. Mit Hilfe des *Datenbank-Managers* (DB-Manager) kann der Object-Manager auf die zugrundeliegende Datenbank[1] zugreifen, die als Objektspeicher dient, und sorgt so für die Persistenz des Systems. Außerdem kann der DB-Manager in Zusammenarbeit mit der Recovery-Komponente von AutO das System im Falle eines Systemausfalls beim Wiederanlauf in einen transaktionskonsistenten Zustand versetzen.

Weitere Komponenten, die bisher nicht erwähnt worden sind, sind der *Nameservice* und der *Informationservice*. Der Nameservice hat die Aufgabe, die aktuellen Aufenthaltsorte aller autonomen Objekte im System zu verwalten. Versucht ein Knoten einem Objekt eine Nachricht zuzustellen, dessen aktueller Aufenthaltsort ihm nicht bekannt ist, wird der Nameservice befragt. Die Kommunikation mit diesem läuft dabei über den Knotenmanager.

Mit Hilfe des Informationservice können allgemeine Daten über Objekte bzw. Objekttypen im System gespeichert werden. So ist es möglich, Objekten global gültige Namen zuzuweisen und z.B. Beschreibungen und Schlüsselworte für die verschiedenen Typen autonomer Objekte zu hinterlegen. Außerdem kann beim Informationservice angefragt werden, welches ihm bekannte Objekt eines bestimmten Typs einem gegebenen Rechner am nächsten liegt. Dazu greift der Informationservice auch auf den Nameservice zu, um die Aufenthaltsorte der in Frage kommenden Objekte zu erfahren.

[1] Dabei steht entweder AutOShore (das speziell für AutO auf der Basis von Shore [CDF 94] implementiert wurde) oder jede JDBC-fähige Datenbank [HCF97] zur Auswahl.

2.3 Migration

Da AutO für den Einsatz in einem WAN ausgelegt ist, unterstützt es die Migration von Objekten durch eine in AutO integrierte Migrationskomponente [Isl97], die die Kommunikationskosten auf eine für den Benutzer transparente Weise senkt: Die Objekte migrieren autonom, wenn die dadurch erwartete Kostenersparnis groß genug ist. Die Strategie, nach der die Objekte ihre Migrationsentscheidung fällen, ist nicht fest vom System vorgegeben, sondern kann vom Benutzer implementiert werden und ist auch während des Betriebs dynamisch austauschbar. Dabei kann jedem Objekt eine eigene Strategie zugewiesen werden. Ihr stehen für die Migrationsentscheidung die Informationen über die vergangenen Zugriffe auf das Objekt zur Verfügung.

Das Migrationssystem besteht aus drei Hauptteilen, die im folgenden beschrieben werden: *Migrationsinstanz* (`MigratingInstance`), *Migrationsüberwacher* (`MigrationSupervisor`) und *Migrationsassistent* (`MigrationAssistant`). Jedes autonome Objekt in AutO besitzt eine eigene Migrationsinstanz, die die Zugriffe von Transaktionen auf das Objekt in Statistiken verwaltet. Das Objekt wertet diese nach einer gewissen (konfigurierbaren) Anzahl neu empfangener Nachrichten mit der ihm zugewiesenen Strategie aus und überprüft so, ob eine Migration vorteilhaft ist. Wenn dies der Fall ist, leitet das Objekt die Migration ein.

Im Gegensatz zur Migrationsinstanz existiert der Migrationsassistent nur einmal pro AutO-Prozeß. Die Hauptaufgabe dieses Assistenten ist es, Objekte, die in den Prozeß migrieren, zu instanziieren und Nachrichten wegmigrierender Objekte zu puffern, solange diese noch nicht einsatzfähig sind. Außerdem überwacht der Migrationsassistent die Last in seinem Prozeß, die durch die Anzahl von Nachrichten in einer festgelegten Zeitspanne definiert ist, und informiert periodisch den Knotenmanager darüber. Durch dieses Verfahren ist es möglich, eine gewisse Lastverteilung im System zu erzielen, da migrationswillige Objekte bei zu hoher Last abgelehnt werden können.

Der Migrationsüberwacher schließlich ist für die Überwachung der Migrationen verantwortlich und ist in den Knotenmanager integriert. Basierend auf den Lastinformationen, die ihm über die Prozesse vorliegen, entscheidet der Migrationsüberwacher, ob ein Objekt angenommen oder abgelehnt wird. Läuft mehr als ein Prozeß, wählt er auch aus, in welchen lokalen Prozeß ein Objekt migrieren soll. Darüber hinaus informiert der Migrationsüberwacher den Nameservice über die Migration des Objektes und puffert ankommende Nachrichten für das Objekt, bis das Objekt nach erfolgreicher Migration wieder instanziiert wurde. Sind nicht alle für das migrierende Objekt nötigen Klassen auf dem Zielrechner der Migration vorhanden, kümmert sich der Migrationsüberwacher darum, daß ihm die benötigten Klassen bei der Migration zugeschickt werden und installiert diese im System. Da es nicht immer möglich ist, alle Klassen festzustellen, die ein Objekt benötigt, wird dieses Verfahren durch einen speziellen *ClassLoader* (siehe Abschnitt 5.1) unterstützt, der bei Bedarf fehlende Klassen aus dem Netzwerk nachlädt.

Nachdem nun alle an der Migration beteiligten Komponenten vorgestellt worden sind, kann das Migrationsprotokoll am Beispiel einer erfolgreich verlaufenden Migration eines Objektes O von Rechner A auf Rechner B erklärt werden. Der Nachrichtenfluß einer

derartigen Migration ist in Abbildung 2.3 dargestellt.

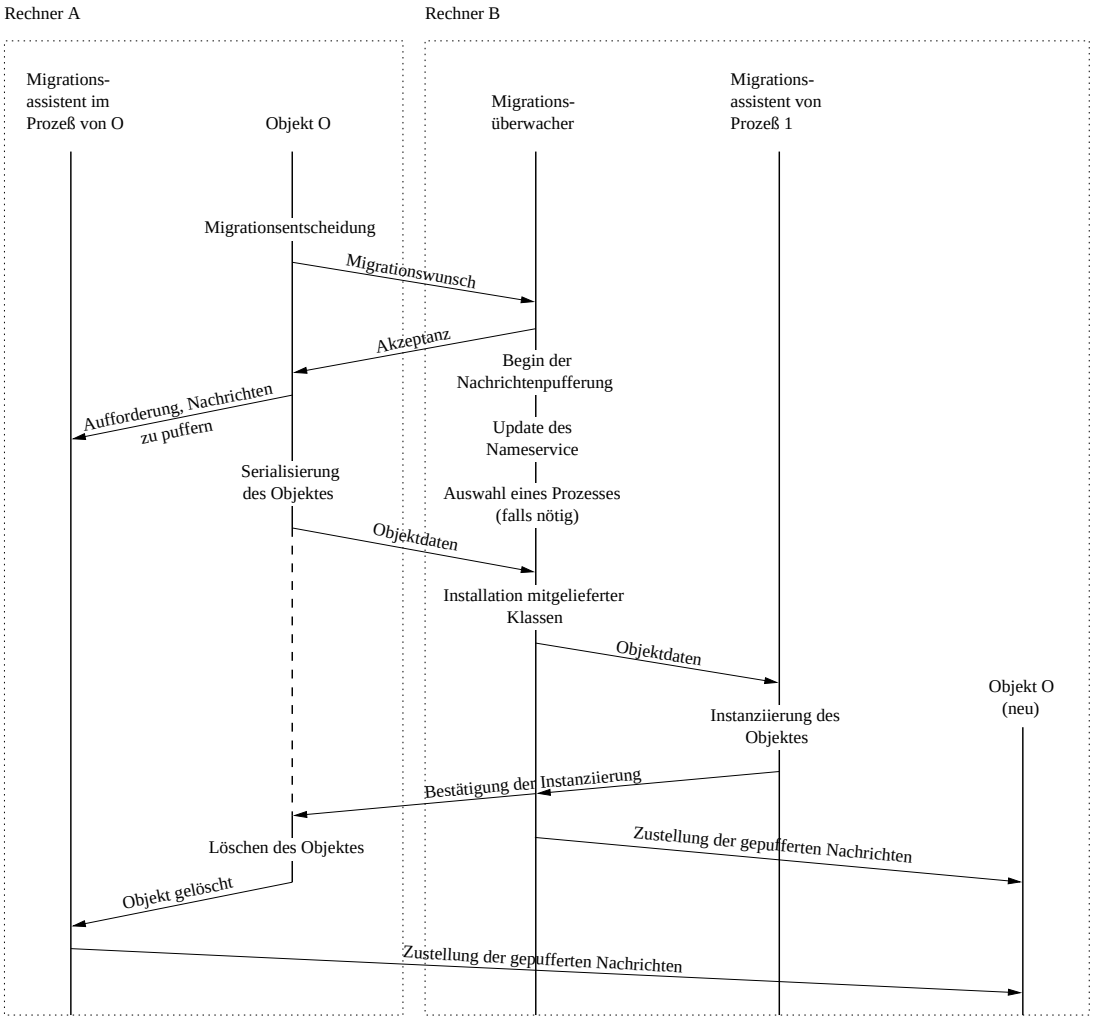


Abbildung 2.3: Migrationsprotokoll (vgl. [KIK98])

Hier entscheidet sich Objekt O, von Rechner A auf Rechner B zu migrieren. Deshalb sendet es eine entsprechende Nachricht, die auch Informationen über die vom Objekt benötigten Klassen enthält, an den Migrationsüberwacher auf Rechner B und wartet auf dessen Antwort. In dieser Zeit arbeitet O normal weiter und ist nicht blockiert. Der Migrationsüberwacher stimmt der Migration entweder zu oder lehnt sie ab (z.B. wegen zu hoher Last auf Rechner B). In diesem Beispiel stimmt er dem Migrationswunsch zu und schickt eine entsprechende Nachricht zurück. Diese enthält auch die Namen der von O benötigten Klassen, die auf Rechner B nicht zur Verfügung stehen. Daraufhin beginnt der Migrationsüberwacher von Rechner B, Nachrichten für O zu puffern, informiert den Nameservice über den neuen Aufenthaltsort des Objektes und wählt einen der existierenden Prozesse aus, in den das Objekt geladen werden soll (in diesem Beispiel ist das Prozeß 1). Existiert kein Prozeß, wird ein generischer AutO-Prozeß gestartet.

Nach Erhalt der Zusage informiert das Objekt O den zuständigen Migrationsassistenten

auf Rechner A, daß Nachrichten für Objekt O ab sofort gepuffert werden sollen. Dann beginnt das Objekt, sich selbst zu serialisieren, also eine Bytefolge zu erstellen, mit dessen Hilfe das Objekt wieder exakt rekonstruiert werden kann. Dann schickt es eine Nachricht mit diesem serialisierten Abbild an den Migrationsüberwacher von Rechner B. Ab diesem Zeitpunkt arbeitet Objekt O nicht mehr. Rechner B installiert daraufhin die Klassen, die von Objekt O benötigt werden, und gibt die Nachricht weiter an den Migrationsassistenten des Prozesses, der für Objekt O ausgewählt wurde. Dieser instanziiert das Objekt und meldet diesen Vorgang dem lokalen Migrationsüberwacher, der alle gepufferten Nachrichten an die neue Instanz von O schickt und die Nachricht über die Instanziierung von O an die alte Instanz von O weiterleitet, die sich daraufhin löscht. Dieser Löschvorgang wird dem zuständigen Migrationsassistenten auf Rechner A gemeldet, der seinerseits der neuen Instanz des Objektes O alle gepufferten Nachrichten nachsendet.

An diesem kleinen Beispiel wurde nur eine erfolgreich ablaufende Migration ohne Zwischenfälle demonstriert. Das Migrationssystem ist allerdings so ausgelegt, daß keine Objekte verdoppelt werden oder verloren gehen, wenn Fehler z. B. durch Nachrichtenverlust auftreten. Eine genaue Beschreibung des Protokolls, die auch auf Maßnahmen im Fehlerfall eingeht, ist in [Isl97] enthalten.

Die Entscheidung, wann und wohin ein Objekt migriert, ist abhängig von der Strategie, die ein Objekt verwendet. Wie eine solche aussehen kann, zeigt das nächste Kapitel, das zwei Migrationsstrategien vorstellt und bezüglich Leistungsfähigkeit und Rechenaufwand miteinander vergleicht.

Kapitel 3

Analyse verschiedener Migrationsstrategien

In diesem Kapitel werden zwei verschiedene Migrationsstrategien bezüglich Leistungsfähigkeit und Rechenaufwand verglichen. Die hier vorgestellten Messungen wurden in Zusammenarbeit mit Natalija Krivokapić, Markus Islinger und Professor Alfons Kemper durchgeführt. Die Ergebnisse können im Technischen Bericht [KIK98] nachgelesen werden. Dieses Kapitel stellt eine Zusammenfassung dieses Berichtes und der Ergebnisse der verschiedenen Benchmarks dar.

3.1 Netzstruktur eines WANs

Die Migrationsstrategie eines autonomen Objektes sollte dieses veranlassen, auf denjenigen Rechner zu migrieren, auf dem die Zugriffe auf das Objekt voraussichtlich am wenigsten Kommunikationskosten verursachen werden. Innerhalb eines LANs würde es dabei ausreichen, festzustellen, von welchem Rechner aus in der Vergangenheit am häufigsten auf ein Objekt zugegriffen worden ist, da die Kosten zwischen je zwei Rechnern innerhalb eines LANs nicht sehr voneinander abweichen. Basierend auf der Annahme, daß das Zugriffsverhalten in der Zukunft etwa dem in der Vergangenheit entspricht, kann davon ausgegangen werden, daß dieser Rechner auch in der näheren Zukunft ein optimaler Aufenthaltsort für dieses Objekt bezüglich der Kommunikationskosten im System ist.

In einem weitverteilten System wie AutO, das für den Einsatz in einem WAN ausgelegt ist, ist eine derart einfache Zählstatistik nicht mehr ausreichend. Die Zugriffszeiten zwischen verschiedenen Rechnern innerhalb eines WANs können stark differieren und müssen deshalb bei der Berechnung des optimalen Rechners beachtet werden. Beispielsweise sind die Kommunikationskosten zwischen LANs, die sich auf demselben Kontinent befinden, im Normalfall sehr viel geringer, als die für transkontinentale Kommunikation. Die Kosten zwischen den Rechnern innerhalb eines LANs können im Vergleich dazu vernachlässigt werden. Aus diesem Grund bilden die durchschnittlichen Kosten zwischen LANs die Berechnungsgrundlage bei der Suche nach dem besten Aufenthaltsort für ein Objekt.

Ein kleines Beispiel einer derartigen Netzwerkstruktur zeigt Abbildung 3.1. Die durchschnittlichen Kommunikationskosten zwischen den LANs sind in Millisekunden angegeben. Speziell für die beiden in diesem Kapitel vorgestellten Migrationsstrategien wurde

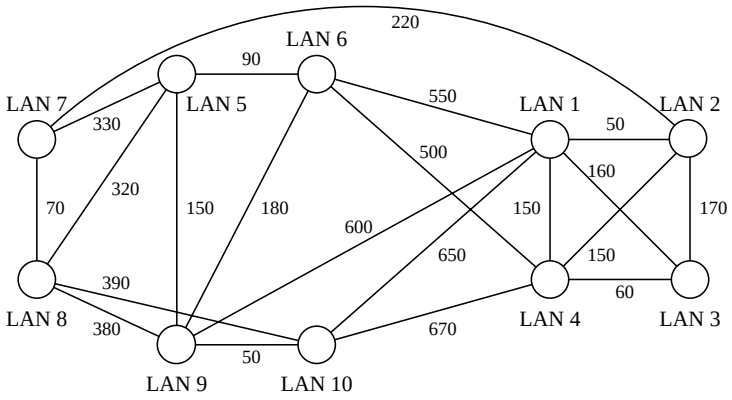


Abbildung 3.1: Beispiel einer Netzwerkstruktur eines weltweit verteilten Systems

eine Komponente (`AutOSystemStructure`) für AutO entwickelt, die Informationen über die Struktur des Systems verwaltet. Um diese Informationen den Objekten zugänglich zu machen, besitzt jeder AutO-Prozeß eine eigene Instanz dieser AutOSystemStructure. Mit Hilfe dieser Daten kann ein Objekt sehr einfach die Kommunikationskosten berechnen, die durch Zugriffe auf das Objekt entstanden sind. Genauso ist es einem Objekt möglich, die Kosten zu berechnen, die angefallen wären, wenn das Objekt zur Zeit der Zugriffe in einem beliebigen anderen LAN plaziert gewesen wäre[1]. Somit ist es einem Objekt möglich, das LAN zu bestimmen, das in der Vergangenheit als Aufenthaltsort optimal bezüglich der Kommunikationskosten gewesen wäre. Innerhalb dieses LANs wird dann der Rechner gewählt, von dem die meisten Zugriffe auf das Objekt stammen, da dadurch die anfallenden Kommunikationskosten unter Umständen nochmals geringfügig gesenkt werden können. Sind bei dieser Auswahl mehrere Rechner gleichwertig, oder hat keiner der Rechner auf das Objekt zugegriffen, wird zufällig einer der in Frage kommenden Rechner ausgewählt.

3.2 Vorstellung der Strategien

Beide hier vorgestellten Migrationsstrategien fällen ihre Migrationsentscheidungen basierend auf dem Zugriffsverhalten in der Vergangenheit und den Strukturinformationen über das Netzwerk, die ihnen vom AutO-System zur Verfügung gestellt werden. Die Finding-The-Best-LAN Strategie[2] berechnet aus den zur Verfügung gestellten Daten den Rechner, der in der Vergangenheit optimal gewesen wäre, indem sie explizit für jedes LAN die Kosten berechnet, die entstanden wären, wenn sich das Objekt in diesem LAN aufgehal-

[1] Die Kosten innerhalb eines LANs werden, wie bereits erwähnt, vernachlässigt. Damit sind alle Rechner eines LANs bei dieser Berechnung gleichwertig.

[2] Im folgenden wird die Finding-The-Best-LAN Strategie oft als Best-LAN Strategie bezeichnet.

ten hätte. Die Finding-A-Good-LAN Strategie[3] verwendet eine Heuristik, auf die noch genauer eingegangen wird, um ein geeignetes Migrationsziel zu ermitteln.

3.2.1 Finding-The-Best-LAN Strategie

Die Finding-The-Best-LAN Strategie betrachtet eine bestimmte (konfigurierbare) Anzahl an Nachrichten, die ein Objekt empfangen hat. Um das LAN zu finden, das die optimale Plazierung für dieses Objekt in der jüngsten Vergangenheit gewesen wäre, berechnet die Strategie für jedes LAN die Kommunikationskosten, die angefallen wären, wenn das Objekt in diesem LAN stationiert gewesen wäre. Nach Berechnung der Kosten für alle LANs werden die minimalen Kosten K_{best_lan} mit den Kosten im aktuellen LAN K_{akt_lan} verglichen. Liegt die durch die Migration zu erwartende Kosteneinsparung über einem Schwellwert t_{global}, wird die Migration zu dem besten LAN eingeleitet:

$$K_{akt\_lan} - K_{best\_lan} \geq t_{global}$$

Dabei migriert das Objekt auf den Rechner, von dem die meisten Zugriffe auf das Objekt stammen bzw. einen zufällig gewählten Rechner des LANs, falls kein solcher Rechner existiert.

Ist das aktuelle LAN schon optimal bezüglich der Kommunikationskosten, berechnet das Objekt $K_{best\_rechner}$. Das sind die Kosten des Rechners innerhalb des aktuellen LANs, von dem die meisten Zugriffe stammen. Liegt hierbei die Kostenersparnis über einem gewissen Schwellwert t_{lokal}, wird innerhalb des aktuellen LANs migriert:

$$K_{akt\_rechner} - K_{best\_rechner} \geq t_{lokal}$$

Die Werte t_{global} und t_{lokal} repräsentieren dabei die Kostenersparnis, die nötig ist, um eine Migration als sinnvoll einzustufen. Aus naheliegenden Gründen werden beide Werte in Abhängigkeit der Anzahl der betrachteten Nachrichten und einer gewünschten mittleren Ersparnis pro Nachricht c_{global} bzw. c_{lokal} berechnet:

$$t_{global} = c_{global} \times \text{Anzahl\_der\_betrachteten\_Nachrichten}$$

$$t_{lokal} = c_{lokal} \times \text{Anzahl\_der\_betrachteten\_Nachrichten}$$

Die Verwendung von zwei verschiedenen Konstanten c ist notwendig, da eine Migration innerhalb eines LANs kein so hohes Einsparungspotential bietet wie eine Migration zwischen zwei LANs. Dafür ist sie billiger, da die Nachrichten, die den Migrationsablauf steuern, innerhalb eines LANs verschickt werden.

Die Migrationskosten an sich werden bei der Berechnung des optimalen Rechners nicht betrachtet, da davon ausgegangen wird, daß das Objekt nach der Migration einige Zeit auf dem gewählten Rechner verweilen wird. Die im Vergleich zu den zu erwartenden Kosteneinsparungen relativ geringen und selten anfallenden Migrationskosten sind damit vernachlässigbar.

[3] Im folgenden wird die Finding-A-Good-LAN Strategie oft als Good-LAN Strategie bezeichnet.

Der Pseudo-Code für die beschriebene Strategie sieht folgendermaßen aus, wenn das Objekt O (plaziert auf dem Rechner $akt\_rechner$ im LAN akt_lan) migrieren will:

RechnerID berechneMigrationsZielBestLan()
  K_{akt_lan} := berechneKommunikationsKosten(akt_lan);
  $best\_lan := akt\_lan$;
  $K_{best\_lan} := K_{akt\_lan}$;
  for each LAN $L \neq akt\_lan$
    $kosten$:= berechneKommunikationsKosten(L);
    if ($kosten < K_{best\_lan}$) **then**
      $K_{best\_lan} := kosten$;
      $best\_lan := L$;
    endif
  endfor
  if ($K_{akt\_lan} - K_{best\_lan} \geq c_{global} \times anz\_betr\_nachrichten$) **then**
    return RechnerID des Rechners aus $best\_lan$ mit den meisten Zugriffen;
  endif
  if ($akt\_lan == best\_lan$) **then**
    $best\_rechner$:= Rechner aus $best\_lan$ mit den meisten Zugriffen;
    $K_{best\_rechner}$:= berechneKommunikationsKosten($best\_rechner$);
    $K_{akt\_rechner}$:= berechneKommunikationsKosten($akt\_rechner$);
    if ($K_{akt\_rechner} - K_{best\_rechner} \geq c_{lokal} \times anz\_betr\_nachrichten$) **then**
      return RechnerID von best_rechner;
    endif
  endif
  return RechnerID von akt_rechner;

3.2.2 Finding-A-Good-LAN Strategie

Die eben beschriebene Strategie findet immer den Rechner mit den geringsten Kommunikationskosten im System[4]. Problematisch ist der Rechenaufwand, der durch die Auswertung der Strategie verursacht wird, da für jedes LAN im System die Kommunikationskosten berechnet werden müssen. Auch bei einer sehr effizienten Implementierung der obigen Strategie wird es in einem großen System zu Performaceproblemen kommen, wenn viele Objekte gleichzeitig versuchen, ihre Strategie auszuwerten. Aus diesem Grund wurde eine Heuristik mit dem Ziel entwickelt, ähnlich gute Kommunikationskosteneinsparungen bei wesentlich geringerem Rechenaufwand zu erzielen.

Betrachtet man die Netzstuktur eines WANs genau, kristallisiert sich eine hierarchische Struktur heraus, die mit der Netzwerkentfernung[5] der LANs zusammenhängt. Diese Struk-

[4] Dies gilt nur unter der Annahme, daß das Zugriffsverhalten auf das Objekt gleich bleibt. Verändert sich dieses nicht zu stark, ist der gewählte Rechner immer noch eine gute Plazierung für das Objekt.

[5] Netzwerkentfernung bedeutet hier die mittlere Übertragungszeit einer Nachricht von einem Netzwerk in ein anderes.

tur korreliert normalerweise mit der geographischen Lage der Rechner, dies ist aber nicht zwingend. So kann ein Rechner in München zu einem Rechner in Passau sehr hohe Kommunikationskosten aufweisen, während er zu einem Rechner in New York über eine sehr schnelle Verbindung verfügt. Der Normalfall ist jedoch, daß die Kommunikationskosten mit zunehmendem geographischen Abstand steigen. LANs, die bezüglich der Kommunikationskosten nicht weit voneinander entfernt sind, können zu Clustern (Regionen) zusammengefasst werden. Diese wiederum können, basierend auf der durchschnittlichen Übertragungszeit, zu größeren Clustern (Kontinenten) zusammengefasst werden, wie Abbildung 3.2 verdeutlicht.

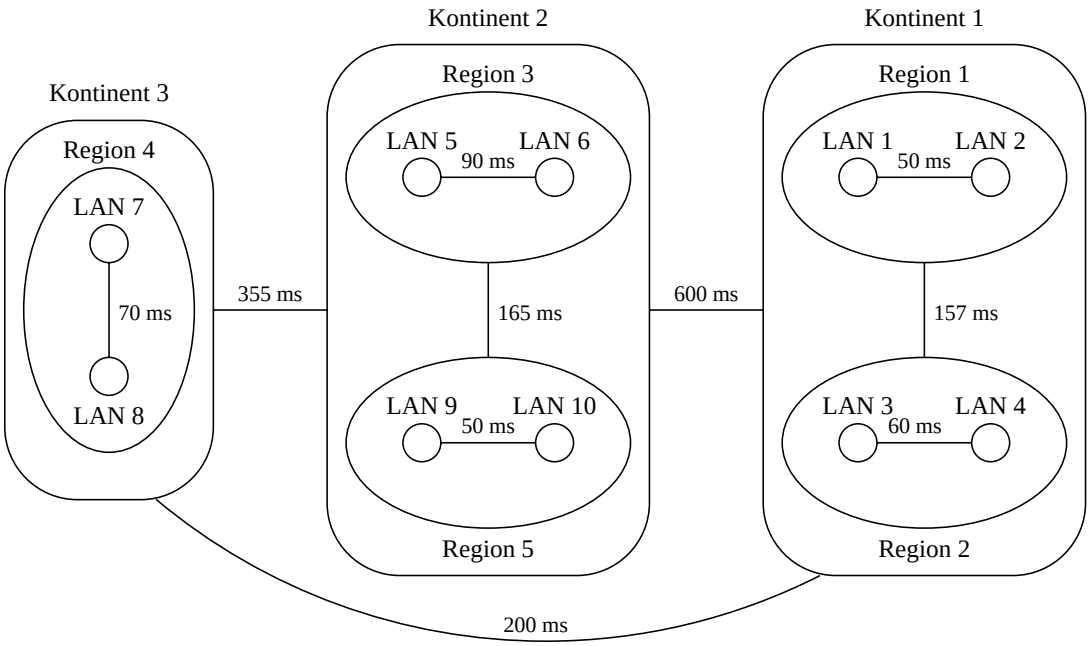


Abbildung 3.2: Beispiel einer hierarchischen Netzwerkstruktur

Die angegebenen Übertragungszeiten der Cluster geben dabei den Mittelwert der Übertragungszeiten der Sub-Cluster an. In diesem Beispiel enthalten Regionen LANs, die weniger als 100 ms mittlere Übertragungszeit haben; Kontinente fassen Regionen zusammen, die im Durchschnitt weniger als 200 ms zur Kommunikation benötigen.

Basierend auf dieser hierarchischen Struktur kann eine Migrationsstrategie entwickelt werden. Im ersten Schritt der Entscheidung, ob migriert werden soll oder nicht, werden nur Kommunikationskosten auf Kontinentebene betrachtet. Die Strategie berechnet dabei die Kosten, die für die betrachteten Nachrichten entstanden wären, wenn das Objekt in einem anderen Kontinent beheimatet gewesen wäre. Zu beachten ist dabei, daß nur interkontinentale Nachrichten in die Berechnung einfließen und daß diese Berechnung auf den durchschnittlichen Kommunikationskosten zwischen den Kontinenten beruht.

Derjenige Kontinent mit den geringsten Kommunikationskosten wird auf Regionen-Ebene betrachtet, es werden also die Kosten berechnet, die angefallen wären, wenn sich das Objekt in einer der Regionen des Kontinents aufgehalten hätte. Da die interkontinentalen

Kosten für alle Regionen eines Kontinents im Mittel gleich groß sind, finden in diesem Berechnungsschritt nur die Nachrichten Beachtung, die innerhalb des gewählten Kontinents verschickt worden sind.

Wie schon bei den Kontinenten, wird nun die Region mit den geringsten Kommunikationskosten ausgewählt und auf LAN-Ebene betrachtet. Auch hier wird wieder das beste LAN ausgewählt. Innerhalb des besten LANs gewinnt der Rechner, von dem die meisten Zugriffe auf das betrachtete Objekt stammen. Existiert kein solcher Rechner, wird zufällig ein Rechner des LANs ausgewählt. Die Entscheidung, ob migriert wird, wird auf dieselbe Art und Weise getroffen, wie schon bei der Best-LAN Strategie.

Der Pseudo-Code für die beschriebene Strategie sieht damit folgendermaßen aus, wenn das Objekt O (plaziert auf dem Rechner $akt\_rechner$ im LAN akt_lan) migrieren will:

RechnerID berechneMigrationsZielGoodLan()
  K_{akt_lan} := berechneKommunikationsKosten(akt_lan);
  $best\_kontinent$:= berechneBestenKontinent();
  $best\_region$:= berechneBesteRegion($best\_kontinent$);
  $best\_lan$:= berechneBestesLAN($best\_region$);
  if ($K_{akt\_lan} - K_{best\_lan} \geq c_{global} \times anz\_betr\_nachrichten$) **then**
    return RechnerID des Rechners aus $best\_lan$ mit den meisten Zugriffen;
  endif
  if ($akt\_lan == best\_lan$) **then**
    $best\_rechner$:= Rechner aus $bast\_lan$ mit den meisten Zugriffen;
    $K_{best\_rechner}$:= berechneKommunikationsKosten($best\_rechner$);
    $K_{akt\_rechner}$:= berechneKommunikationsKosten($akt\_rechner$);
    if ($K_{akt\_rechner} - K_{best\_rechner} \geq c_{lokal} \times anz\_betr\_nachrichten$) **then**
      return RechnerID von best_rechner;
    endif
  endif
  return RechnerID von akt_rechner;

Die drei verwendeten Methoden $berechneBestenKontinent$, $berechneBesteRegion$ und $berechneBestesLAN$ berechnen den besten Kontinent bzw. die beste Region bzw. das beste LAN wie oben geschildert, und zusätzlich die anfallenden Gesamtkommunikationskosten, um den zu erwartenden Gewinn einer Migration abschätzen zu können.

3.3 Vergleich der Strategien

Dieser Abschnitt befasst sich mit dem Vergleich der beiden Strategien. Wie schon erwähnt, findet die Best-LAN Strategie stets die Plazierung, die für die betrachteten Nachrichten optimal gewesen wäre. Die Good-LAN Strategie berechnet eine Plazierung mit Hilfe einer Heuristik, die suboptimale Ergebnisse liefern kann, dafür aber weniger Rechenzeit benötigt. Die Good-LAN Strategie kann deshalb eine andere Plazierung liefern als Best-LAN,

da sie mit den durchschnittlichen Kommunikationskosten zwischen Clustern rechnet und nicht mit den Kosten zwischen LANs direkt. Hat nicht jeder Sub-Cluster ungefähr dieselbe Anzahl an Nachrichten versendet, stimmen die berechneten Kosten nicht mehr mit den tatsächlichen Kosten überein.

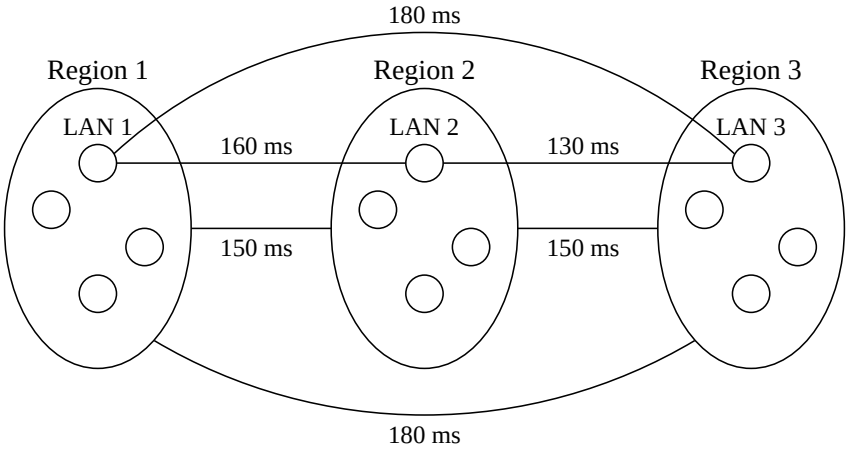


Abbildung 3.3: Beispiel für verschiedene Migrationsentscheidungen der beiden Strategien

Ein Beispiel, bei dem die beiden Strategien verschiedene Ergebnisse liefern, zeigt Abbildung 3.3. Um die Netzstruktur übersichtlich zu halten, besteht sie lediglich aus drei Regionen. Das betrachtete Objekt O befindet sich auf einem Rechner im LAN 2 und hat 200 Nachrichten von LAN 1, 20 Nachrichten von LAN 2 und 190 Nachrichten von LAN 3 erhalten. Dadurch ergeben sich für die Good-LAN Strategie folgende Werte:

$$K_{Region1} = 100 \times 150ms + 190 \times 180ms = 49200ms$$
$$K_{Region2} = 200 \times 150ms + 190 \times 150ms = 58500ms$$
$$K_{Region3} = 100 \times 150ms + 200 \times 180ms = 51000ms$$

Nach Berechnung dieser Kosten würde O feststellen, daß am wenigsten Kosten in der Vergangenheit entstanden wären, wenn es auf einem LAN in Region 1 plaziert gewesen wäre. Je nach Migrationsschwellwert wird O also zu LAN 1 migrieren oder auf dem aktuellen Rechner bleiben.

Die Best-LAN Strategie würde folgende Kosten ermitteln:

$$K_{LAN1} = 100 \times 160ms + 190 \times 180ms = 50200ms$$
$$K_{LAN2} = 200 \times 160ms + 190 \times 130ms = 56700ms$$
$$K_{LAN3} = 100 \times 130ms + 200 \times 180ms = 49000ms$$

Mit der Best-LAN Strategie würde das Objekt O also im Falle einer positiven Migrationsentscheidung zu LAN 3 migrieren. Da die Best-LAN Strategie exakt die Kosten für jedes LAN berechnet, ist die Entscheidung der Best-LAN Strategie zwar die bessere, benötigt aber, wie später noch gezeigt wird, wesentlich mehr Rechenleistung.

3.4 Leistungsmessungen in einem Simulationssystem

Um aussagekräftige Messungen durchführen zu können, wurden die Migrationsstrategien in C++ reimplementiert. Für die Benchmarks wurde für diese Strategien ein laufendes System simuliert, indem eine Netzwerkstruktur und zufällige Nachrichten generiert wurden. Basierend auf diesen Daten mußten beide Migrationsstrategien entscheiden, ob migriert werden soll oder nicht. Im Falle einer Migration wurde der optimale Rechner im Ziel-LAN bestimmt. War eine Migration in ein anderes LAN nicht sinnvoll, mußte der Rechner des lokalen LANs bestimmt werden, von dem die meisten Zugriffe stammten, und entschieden werden, ob eine Migration zu diesem ratsam ist. Die Laufzeitexperimente wurden auf einer *Sun UltraSPARC 1 Creator 170E* unter Solaris 2.6 durchgeführt.

3.4.1 Rechenaufwand der Strategien

Um den Rechenaufwand der Strategien zu bestimmen, wurden zufällige Netzwerkstrukturen simuliert, die einigen Randbedingungen genügten. Die Anzahl der Rechner innerhalb dieser Struktur variierte zwischen 100 und 5000. Dabei wurden je fünf Rechner zu einem LAN zusammengefaßt. Für die Good-LAN Strategie wurde außer der Einteilung in LANs noch eine Hierarchie benötigt. Dazu wurde die Anzahl der Kontinente auf 5 festgelegt. Jeder Kontinent mußte mindestens zwei Regionen beinhalten. Die Anzahl der Regionen lag zufällig zwischen 10 und der Hälfte der Anzahl der LANs. Die Kommunikationskosten zwischen den LANs wurden zufällig generiert, genügten dabei allerdings den in Abschnitt 3.2.2 beschriebenen Nebenbedingungen. Für die Berechnung der Migrationsentscheidung wurden jeweils 1000 zufällig generierte Nachrichten, deren Herkunft gleichmäßig auf alle Rechner im System verteilt war, verwendet. Die Ergebnisse des Experimentes zeigt Abbildung 3.4. Der Graph veranschaulicht, daß bei einer kleinen Anzahl an Rechnern der Unterschied an Rechenzeit zur Berechnung des Migrationsziels nicht gravierend war. Je mehr die Anzahl der Rechner im System anstieg, um so größer wurden die Unterschiede in den CPU-Kosten der Migrationsstrategien. Im größten hier simulierten Fall von 5000 Rechnern, also 1000 LANs, war die Good-LAN Strategie fast 90 mal schneller als die Best-LAN Strategie. Dieser Unterschied ist in einem realen System relevant, da die Migrationsstrategien sehr oft ausgewertet werden. Außerdem ist der Rechner, der die Migrationsziele berechnet, schon durch die Aktivitäten seiner autonomen Objekte belastet.

3.4.2 Qualität der Migrationsentscheidungen

Die separate Implementierung der Migrationsstrategien in C++ wurde auch dazu verwendet, die Qualität der Entscheidungen der Migrationsstrategien zu vergleichen. Dabei wurden die Kommunikationskosten der Objekte mit den Kosten verglichen, die aufgetreten wären, wenn sich das Objekt schon auf dem Rechner aufgehalten hätte, den die Strategie vorschlug. Wie im vorhergehenden Benchmark wurden jeweils 1000 zufällig generierte Nachrichten analysiert. Beide Migrationsstrategien berechneten, basierend auf diesen Nachrichten, ihre Migrationsentscheidung. Zuerst wurde untersucht, was die bei-

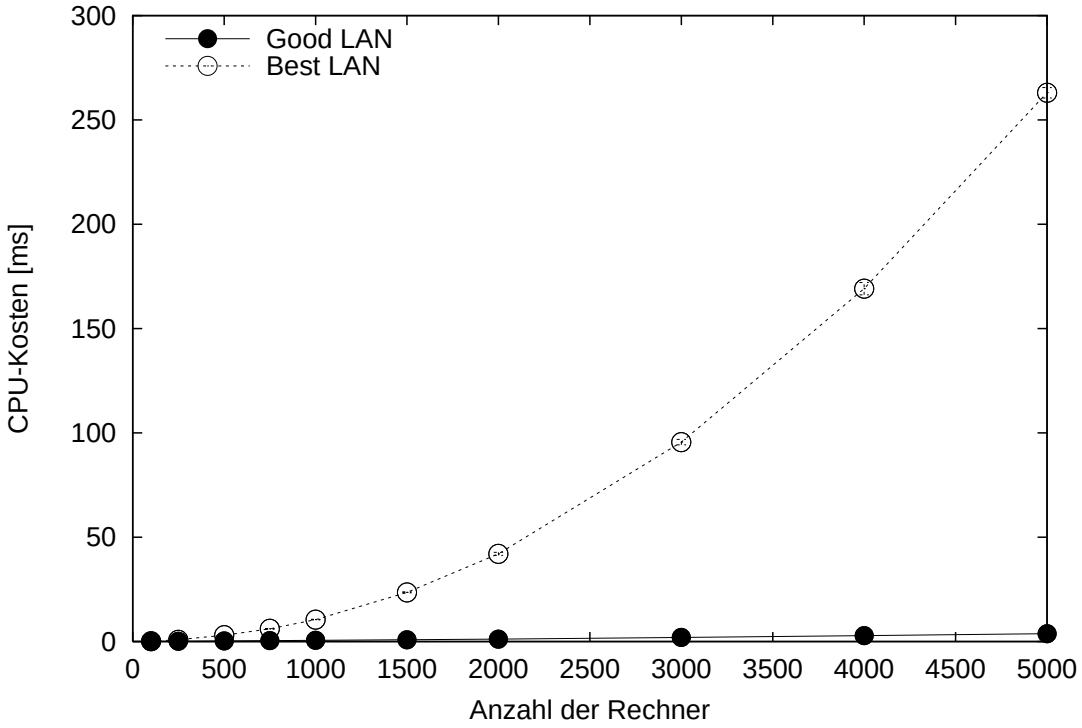


Abbildung 3.4: CPU-Kosten für die Migrationsentscheidung

den Migrationsstrategien leisten können, wenn der Ursprung der 1000 Nachrichten gleichmäßig auf alle Rechner im System verteilt wird. Dabei stellte sich überraschenderweise heraus, daß trotz der geringen Lokalität der Nachrichten eine ca. 15%ige Verringerung der Kommunikationskosten erreicht werden konnte (siehe Abbildung 3.5). Diese Reduzierung der Kommunikationskosten läßt sich dadurch erklären, daß die Objekte auf die Rechner im System migrierten, die im Durchschnitt die schnellsten Netzverbindungen zu anderen Rechnern hatten. Die Good-LAN Strategie schnitt dabei um etwa 2% schlechter ab als die Best-LAN Strategie, da ihr nur die Mittelwerte der Kommunikationskosten zwischen den Clustern bei der Entscheidungsfindung zur Verfügung standen.

Da dieser für die Strategien denkbar ungünstige Fall einer gleichmäßigen Verteilung der Nachrichten in der Realität selten auftritt, wurde ein zweites Experiment mit einer realistischeren Konfiguration durchgeführt. Dabei wurden die 1000 Nachrichten so erzeugt, daß 80% der Nachrichten von einem Kontinent stammten. Die dabei erwarteten Einsparungen an Kommunikationskosten durch die Migration lagen deutlich höher als in dem vorherigen Experiment. Diese Vermutung konnte durch Messungen belegt werden, wie Abbildung 3.6 zeigt. Die Kommunikationskosten sanken hier bei Verwendung der Strategien um über 50%, wobei kleinere Netzwerke mehr von der Migration profitierten als große, da die Anzahl der Regionen pro Kontinent kleiner war. Damit war es wahrscheinlicher, daß Kontinent-lokale Nachrichten auch Region-lokal waren, als in großen Netzwerken. In Anbetracht der nur geringfügig besseren Ergebnisse der Best-LAN Strategie und der deutlich höheren CPU-Kosten, wäre die Good-LAN Strategie in diesem Szenario die bessere Wahl.

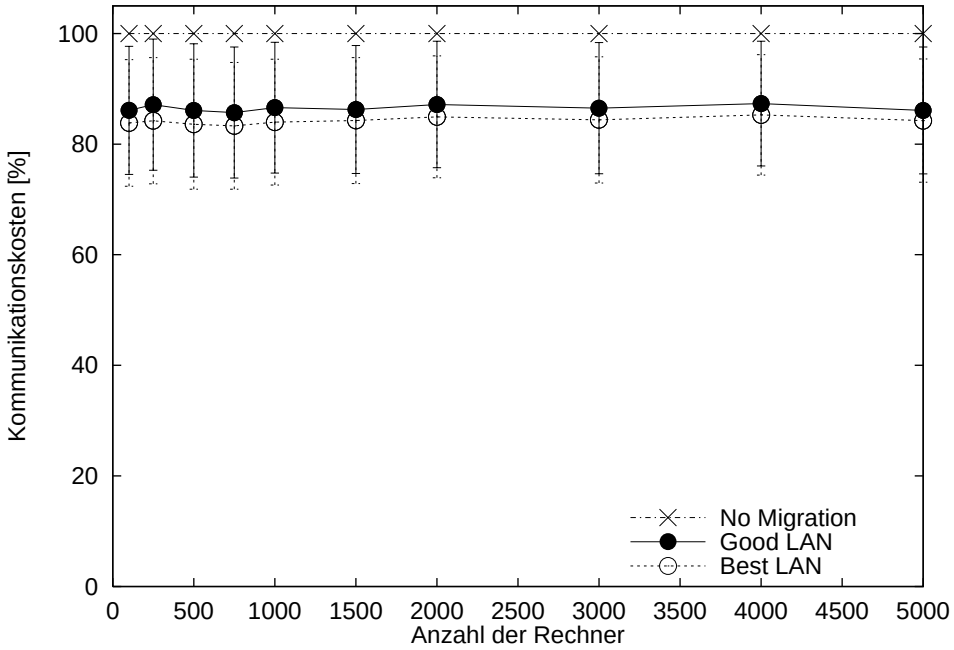


Abbildung 3.5: Kommunikationskosten (Nachrichten gleichverteilt)

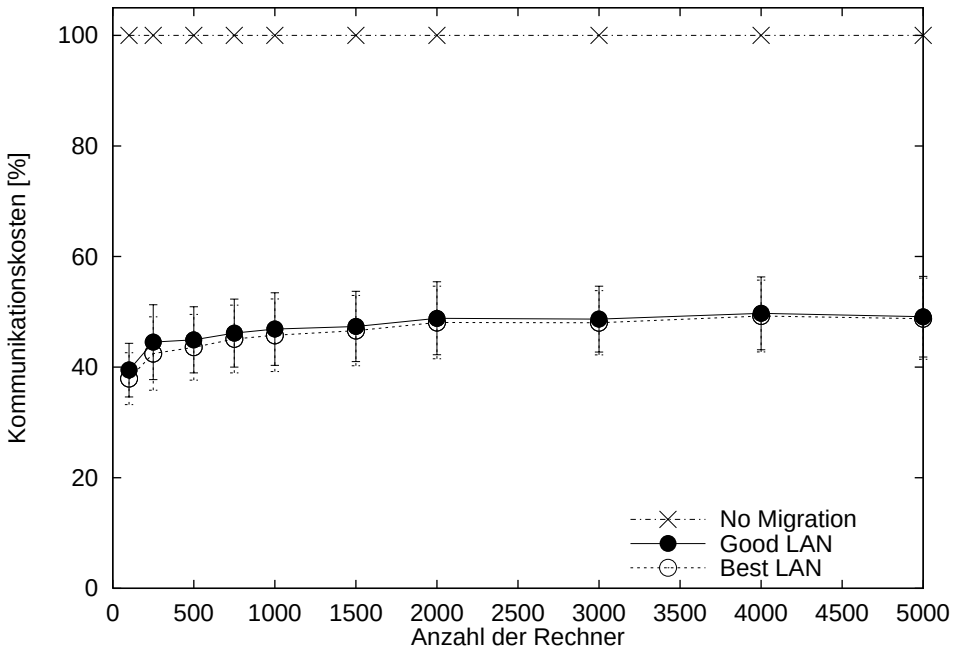


Abbildung 3.6: Kommunikationskosten (80% von einem Kontinent)

3.5 Leistungsmessungen in AutO

Um Aussagen über die Vorteile der Migration in einem realen System treffen zu können, waren Experimente in AutO unter Verwendung des Meßsystems nötig, das von Markus Islinger [Isl97] entwickelt worden ist. Im Gegensatz zu den synthetischen Benchmarks aus Abschnitt 3.4 erlaubten es diese Tests, das System über einen längeren Zeitraum hinweg zu betrachten und somit Aussagen über die Auswirkungen der Migration auf das System im zeitlichen Verlauf zu treffen. Die Migrationsbenchmarks liefen nicht in einer realen AutO-Installation, sondern in einer speziellen Testumgebung, die Netzwerklatenzzeiten simulieren kann. Dieses Vorgehen war nötig, da lediglich zehn Rechner innerhalb eines LANs für die Messungen zur Verfügung standen. Um dennoch realistische Verhältnisse zu schaffen, wurde dem System eine virtuelle Netzwerkstruktur (siehe Abbildung 3.7 und Tabelle 3.1) vorgegeben. Die Netzwerkkosten wurden dann durch das Verzögern von Nachrichten erzeugt, und so das Verhalten eines WANs simuliert.

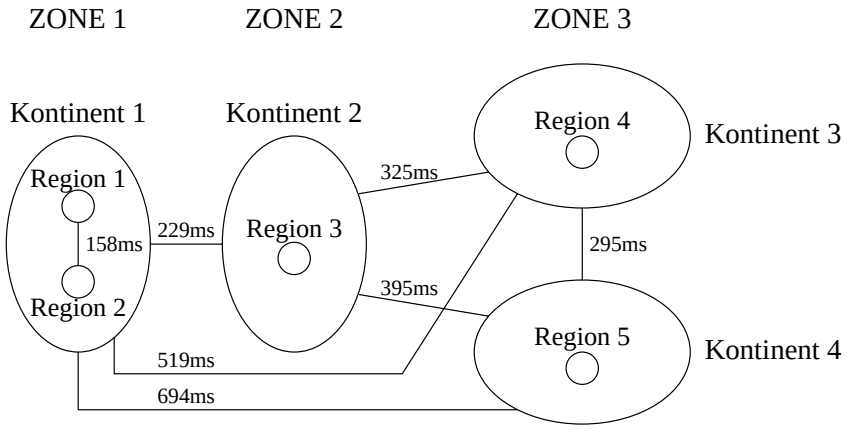


Abbildung 3.7: Virtuelle Netzstruktur der Experimente

Die Experimente liefen in einem virtuellen WAN, das vier Kontinente umfaßte. Ein Kontinent bestand aus zwei Regionen, die anderen drei beinhalteten jeweils nur eine Region. Die Regionen wiederum beherbergten jeweils zwei LANs. Die Best-LAN Strategie benötigte die LAN-zu-LAN Kommunikationskosten für jedes Paar von LANs. Diese Tabelle wird hier nicht angegeben, die Kosten bewegten sich jedoch in dem Rahmen, den Tabelle 3.1 angibt. Die Good-LAN Strategie benötigte die in Abbildung 3.7 gezeigte Hierarchie von Clustern, die im Vorfeld der Experimente berechnet worden ist und dem System zur Verfügung gestellt wurde.

Anzahl Kontinente	4	Kosten zwischen Kontinenten	[200, 700]
Anzahl Regionen	5	Kosten zwischen Regionen	[100, 200]
Anzahl LANs	10	Kosten zwischen LANs	[20, 100]

Tabelle 3.1: Konfiguration der Experimente

Um den Einfluß der Migration auf das System zu testen, wurde eine Anwendung für AutO entwickelt, die ein stark vereinfachtes Flugbuchungssystem modelliert. Da in derartigen Anwendungen die Kosten einer Anfrage zum größten Teil aus Netzwerkkosten bestehen, sind sie für den Test der Migrationskomponente von AutO gut geeignet. In AutO wurde diese Anwendung durch die Implementierung von zwei verschiedenen Objekttypen realisiert: *Fluglinien* und *Flüge*. Jede Fluglinie bietet zehn verschiedene Flüge an und kann als „bevorzugt" gekennzeichnet werden. Im System existieren 20 Fluglinien, wobei jedem Kontinent vier bevorzugte Fluglinien zugeordnet sind. Die Kunden werden durch verschiedene Arten von Transaktionen im System repräsentiert: *TreuerKundeTA*, *ZufallsKundeTA* und *EinfacherKundeTA*. Die Transaktionen TreuerKundeTA und ZufallsKundeTA greifen auf je drei Fluglinien zu. Jede dieser Fluglinien liefert eine Liste mit bis zu fünf Flugangeboten zurück. Die Transaktionen buchen daraufhin ein Drittel dieser Flüge. Der Unterschied zwischen den Transaktionen besteht darin, daß ZufallsKundeTA die drei Fluglinien zufällig aus allen vorhandenen Fluglinien aussucht, während TreuerKundeTA zwei Fluglinien aus den bevorzugten Fluglinien des Kontinents auswählt, auf dem die Transaktion läuft, und nur eine zufällig bestimmt. Der letzte Transaktionstyp EinfacherKundeTA wendet sich nur an eine einzige zufällig gewählte Fluglinie und bucht die Hälfte der angebotenen Flüge. Einen Überblick über die Konfiguration der Experimente in AutO bietet Tabelle 3.2. In einem realen System schwankt die Anzahl der Zugriffe der Transaktionen auf

Anzahl Fluglinien	20
Anzahl Flüge	200 (10 pro Fluglinie)
Maximale Transaktionslast	17 Transaktionen pro Rechner
TreuerKundeTA	
Objektzugriffe (Fluglinien + Flüge) pro Transaktion	[4-8]
Zugriffsverhalten auf Fluglinien	2 bevorzugte Fluglinien 1 zufällige Fluglinie
Prozentualer Anteil am Transaktionsaufkommen	60%
ZufallsKundeTA	
Objektzugriffe (Fluglinien + Flüge) pro Transaktion	[4-8]
Zugriffsverhalten auf Fluglinien	3 zufällige Fluglinien
Prozentualer Anteil am Transaktionsaufkommen	30%
EinfacherKundeTA	
Objektzugriffe (Fluglinien + Flüge) pro Transaktion	[2-3]
Zugriffsverhalten auf Fluglinien	1 zufällige Fluglinie
Prozentualer Anteil am Transaktionsaufkommen	10%

Tabelle 3.2: Konfiguration der Experimente in AutO

Objekte stärker als in diesem Experiment. Da die Migrationsentscheidung aber aufgrund der anfallenden Kommunikationskosten getroffen wird, unabhängig von welchen Transaktionen die Nachrichten abgeschickt wurden, hat dies keinen signifikanten Einfluß auf das Ergebnis.

Auf jedem Rechner liefen maximal 17 Transaktionen gleichzeitig. Transaktionen, die abge-

brochen wurden, wurden automatisch erneut gestartet. Diejenigen, die ihre Arbeit erledigt hatten, wurden durch neue ersetzt. Der Typ der neuen Transaktion wurde gemäß der in Tabelle 3.2 angegebenen prozentualen Anteile der verschiedenen Transaktionstypen zufällig gewählt.

Für alle Rechner im System stand nur ein einziger Nameserver zur Verfügung. Anfragen an diesen wurden nicht verzögert. Das System verhielt sich dadurch so, als ob in jedem LAN ein eigener Nameserver laufen würde, wobei bei jeder Migration alle Server gleichzeitig aktualisiert werden.

Ziel dieses Experimentes war die Reduzierung der Antwortzeiten der Transaktionen durch die Verringerung der anfallenden Kommunikationskosten. Da die Messungen in einem realen System ohne exklusiven Zugriff auf die Rechner durchgeführt wurden, hätte die Messung der Antwortzeiten keine aussagekräftigen Ergebnisse geliefert. Deshalb wurde bei den Messungen das Hauptaugenmerk auf die Kommunikationskosten im System gelegt.

Um die anfallenden Kommunikationskosten im System zu ermitteln, wurde für jede Transaktion die durch sie direkt entstehenden Kosten, basierend auf den künstlichen Netzwerklatenzzeiten, ermittelt. Zusätzlich zu diesen Kosten fielen allerdings noch migrationsbedingte Kosten an. Die Kosten für das Versenden eines migrierenden Objektes wurden allen Transaktionen zugeschlagen, die auf das migrierende Objekt warten mußten. Außerdem wurden die Kosten für das Nachsenden von Nachrichten den Transaktionen, von denen diese Nachrichten stammten, zugeschlagen. In AutO werden normalerweise mehrere Nachrichten gebündelt einem migrierten Objekt nachgesendet, dadurch sind die berechneten Kosten im Durchschnitt höher, als die wirklich anfallenden. Dies benachteiligte die Migrationsstrategien etwas gegenüber einem System ohne Migration, war aber nötig, um den „worst-case“, also das separate Nachsenden jeder einzelnen Nachricht, abzudecken.

In AutO wurden Experimente mit den beiden vorgestellten Migrationsstrategien durchgeführt. Für beide Strategien wurden die durchschnittlichen Kommunikationskosten der Transaktionen und der Anteil an Nachrichten, die innerhalb eines LANs bzw. einer Region versendet worden sind, berechnet. Außerdem wurden die Kosten berechnet, die in einem System ohne Migration angefallen wären und die Anteile an LAN- bzw. Region-lokalen Nachrichten, die in diesem Fall aufgetreten wären.

3.5.1 Szenario 1: Zufällig Verteilte Objekte, Konstante Last

Für dieses Szenario wurden die Objekte zufällig auf die Rechner im System verteilt. Nach der Generierung der Objekte wurde auf allen Rechnern vom Anfang bis zum Ende der Messung die volle Transaktionslast gefahren. In einem WAN-System ist dieses Verhalten nicht sehr realistisch, da sich normalerweise lokale Aktivitätsmaxima herausbilden würden. Eine gleichmäßige Systemlast wie in diesem Versuch erschwert eine Migrationsentscheidung, da Nachrichten von allen Rechnern eintreffen. Dieses Szenario zeigt also das Verhalten der Strategien in einem sehr ungünstigen Umfeld.

Bedingt durch die geringe Anzahl an Rechnern wurde die Migrationsstrategie nach jeweils 40 Nachrichten ausgewertet, und für die Migrationsentscheidung wurden immer diese 40

letzten Nachrichten betrachtet. Das Ergebnis der Messungen zeigt Abbildung 3.8.

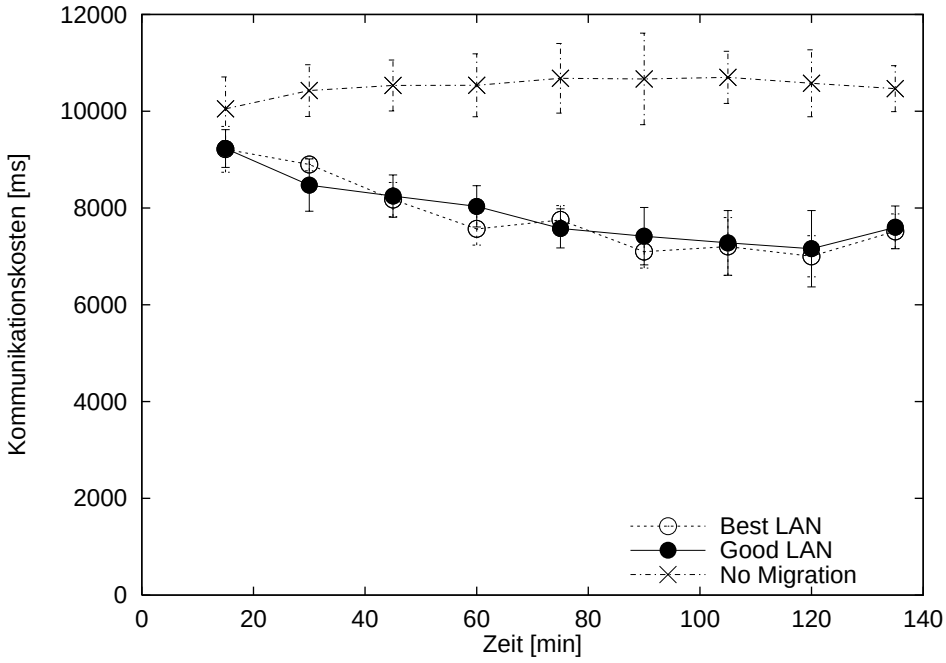


Abbildung 3.8: Durchschnittliche Kommunikationskosten pro Transaktion

Da die Last auf allen Rechnern stets gleich hoch war, waren die Kommunikationskosten, die im Fall ohne Migration auftraten, nahezu konstant. In einem System mit Migration fielen die Kosten nach einiger Zeit langsam, weil die Objekte auf Rechner migrierten, die eine gute Netzanbindung an möglichst viele andere Rechner hatten. Der relativ langsame Abfall läßt sich dadurch erklären, daß die Zugriffe auf die Objekte durch Rechner im gesamten System stattfanden und so nur geringe Lokalität in den Zugriffen vorhanden war. Obwohl dieses Szenario denkbar ungünstig für die Migrationsstrategien war, war es ihnen doch möglich, die Kommunikationskosten um bis zu 34% zu senken. Zwischen den beiden Strategien war in diesem Fall kein signifikanter Unterschied festzustellen. Dies lag vor allem daran, daß es in diesem Szenario keinen wirklich optimalen Rechner, dafür aber mehrere gute gab. Die Abbildungen 3.9 und 3.10 zeigen, wie sich der Anteil LAN- bzw. Region-lokaler Nachrichten im Laufe der Zeit änderte. Beide Strategien schafften es, sowohl die Anzahl der LAN-lokalen als auch die Anzahl der Region-lokalen Nachrichten zu verdoppeln. Dieser Effekt läßt sich darauf zurückführen, daß die Rechner im System nicht alle gleich schnell waren. Von schnelleren Rechnern gingen mehr Nachrichten aus als von langsamen. Deshalb lohnte es sich, auf einen Rechner zu migrieren, der gute Verbindungen zu schnellen Rechnern hatte. Auch hier zeigte sich kein signifikanter Unterschied zwischen den Strategien. In einem realen System wird deshalb die Good-LAN Strategie bevorzugt werden, da sie bei vergleichbar guten Ergebnissen geringere CPU-Kosten verursacht.

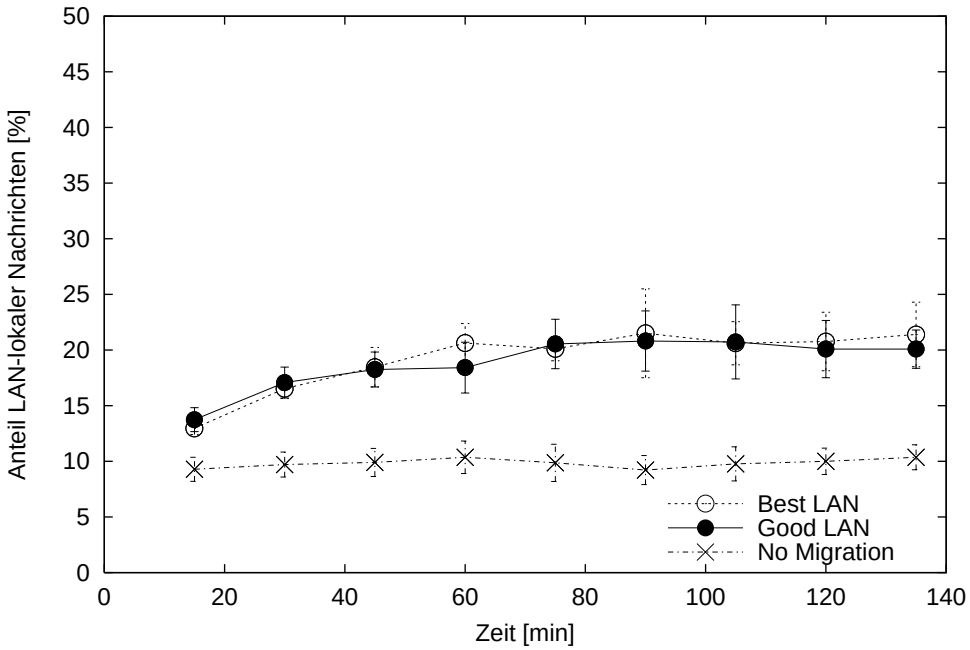


Abbildung 3.9: Anteil LAN-lokaler Nachrichten

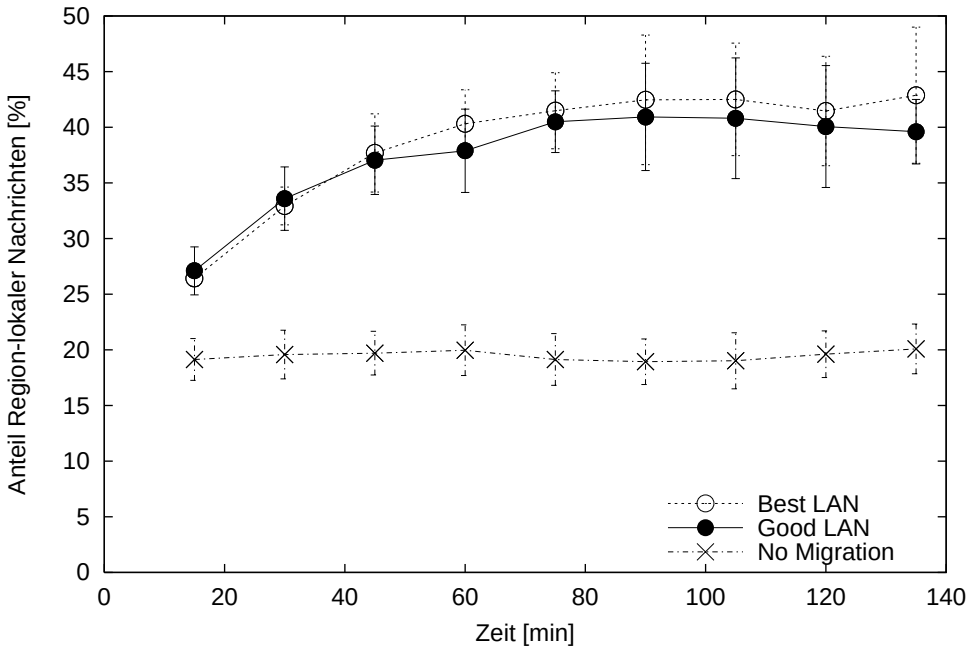


Abbildung 3.10: Anteil Region-lokaler Nachrichten

3.5.2 Szenario 2: Zufällig Verteilte Objekte, Lastverschiebung

In diesem Szenario wurde untersucht, wie sich die Migrationsstrategien in einem realistischeren Fall verhalten. Dazu wurde eine Lastverschiebung in das Szenario integriert, die den natürlichen Tag/Nacht Zyklus nachahmt. Je nach Zone, in der die Rechner lagen (siehe Abbildung 3.7), hatten sie verschiedene Last- und Ruhezeiten, die in Abbildung 3.11 dargestellt werden. Die Aktivität in einer „Zeitzone“ startete dabei mit 50% der ma-

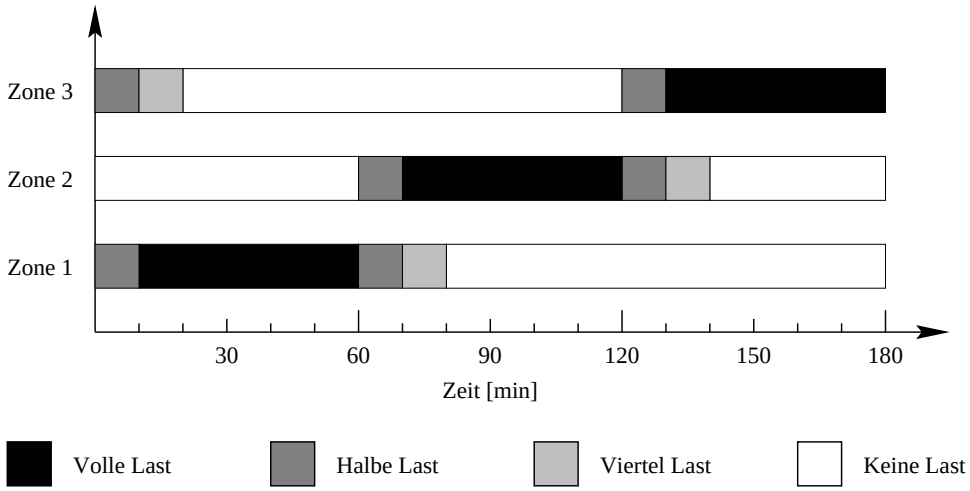


Abbildung 3.11: Lastverschiebung im System

ximalen Last und stieg nach 10 Minuten auf volle Last an. Diese wurde für 50 Minuten beibehalten und danach für je 10 Minuten auf die Hälfte und dann auf ein Viertel der maximalen Last reduziert. Dann ruhten die Rechner einer Zone für 100 Minuten. Ein voller Zyklus beanspruchte 180 Minuten. Auch bei diesen Messungen wurden die Migrationsstrategien nach je 40 empfangenen Nachrichten ausgewertet. Der Aufbau des Experiments war derselbe, wie in Abbildung 3.7 dargestellt.

Die durchschnittlichen Kommunikationskosten pro Transaktion sind in Abbildung 3.12 dargestellt. Die gestrichelten vertikalen Linien markieren die Zeitpunkte, zu denen eine Lastverschiebung einsetzte. In diesem Fall waren die Kommunikationskosten in einem System ohne Migrationskosten natürlich nicht gleichbleibend, sondern schwankten durch die Lastverschiebung. Dieses Verhalten wurde durch die verschiedenen Netzwerkkosten zwischen den Kontinenten und innerhalb der Kontinente erzeugt, da die Kosten in einem Kontinent mit guten Netzverbindungen geringer ausfallen, als in einem Kontinent mit schlechten Kommunikationsanbindungen.

Im Falle eines Systems mit Migration waren die anfallenden Kommunikationskosten deutlich geringer. Auch an diesen Kurven sind die Lastverschiebungen im System deutlich zu erkennen, da bei jeder Verschiebung der Last in eine andere Zone die Kommunikationskosten anstiegen, da die Objekte noch in dem vorher aktiven Kontinent lagen. Nach kurzer Zeit sanken die Kommunikationskosten wieder, da die Objekte auf die veränderte Situation im Netz reagierten und in das neue Lastzentrum migrierten. Dabei sparten die

Migrationsstrategien bis zu 73% Prozent der ohne Migration anfallenden Kosten ein. In einer realen Anwendung treten solche Lastwechsel seltener auf als in diesem Experiment. Es ist also zu erwarten, daß dort die Einsparungen durch Migration noch größer sind, da sich die Kosten einer Migration in einem System mit weniger Lastwechseln schneller amortisiert haben, und das System viel länger von einer Migration profitieren kann.

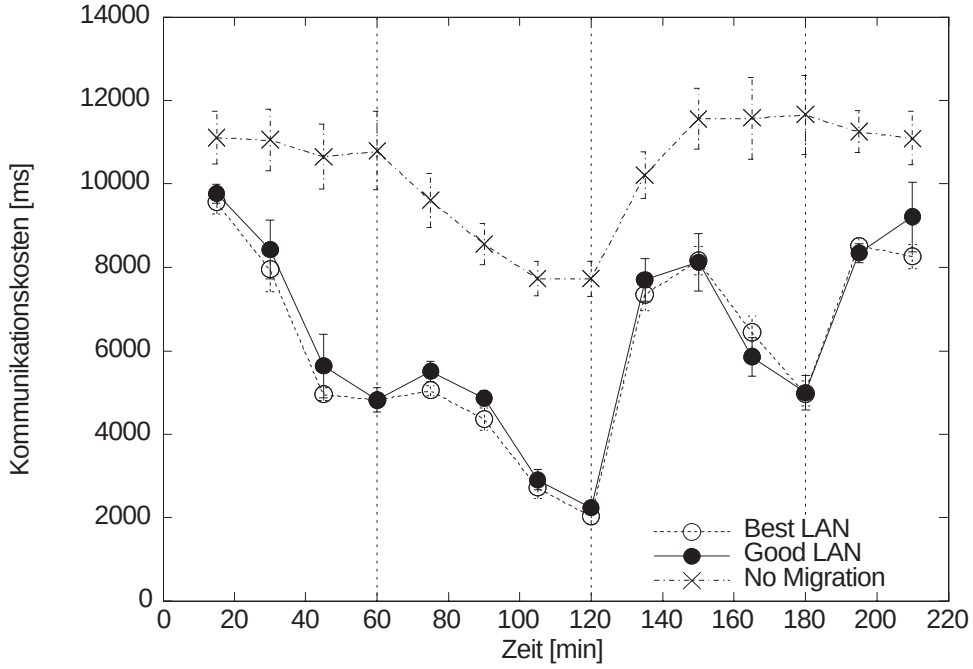


Abbildung 3.12: Durchschnittliche Kommunikationskosten pro Transaktion

Die von den Migrationsstrategien erwirtschafteten Einsparungen waren nicht immer gleich groß, sondern hingen von den Netzverbindungen des aktiven Kontinents ab. In der Zeit, in der die Rechner der Zone 2 aktiv waren, konnten die Strategien einen besonders großen Teil der Kommunikationskosten einsparen, da Kontinent 2 die besten Netzwerkverbindungen zu den anderen Kontinenten hatte. Die Migration eines Objektes auf Kontinent 2 hatte also nicht nur den Vorteil, daß der Anteil an Region- bzw. LAN-lokalen anstieg, sondern bot den Objekten auch geringere Kommunikationskosten mit den anderen Kontinenten. In der Zeit, in der Kontinent 1 oder 3 aktiv waren, migrierten zumindest die Fluglinien auf die aktiven Rechner, da viele Zugriffe von diesen Knoten stammten. Dadurch entstanden allerdings höhere Kommunikationskosten zu den Flügen, die unter Umständen auf anderen Kontinenten lagen.

Ein weiterer Grund für die je nach Zonenaktivität unterschiedlich hohen Einsparungen war die Netzwerkstruktur, wie auch in den Abbildungen 3.13 und 3.14 gut zu erkennen ist.

War Zone 1 aktiv, verteilten sich die Objekte auf zwei Regionen, also vier LANs, was dazu führte, daß öfter die Kommunikationskosten zwischen LANs und zwischen den beiden Regionen anfielen. Natürlich ist die Verteilung der Objekte auf verschiedene Rechner in einem laufenden System erwünscht, da sonst einzelne Rechner zu stark belastet werden.

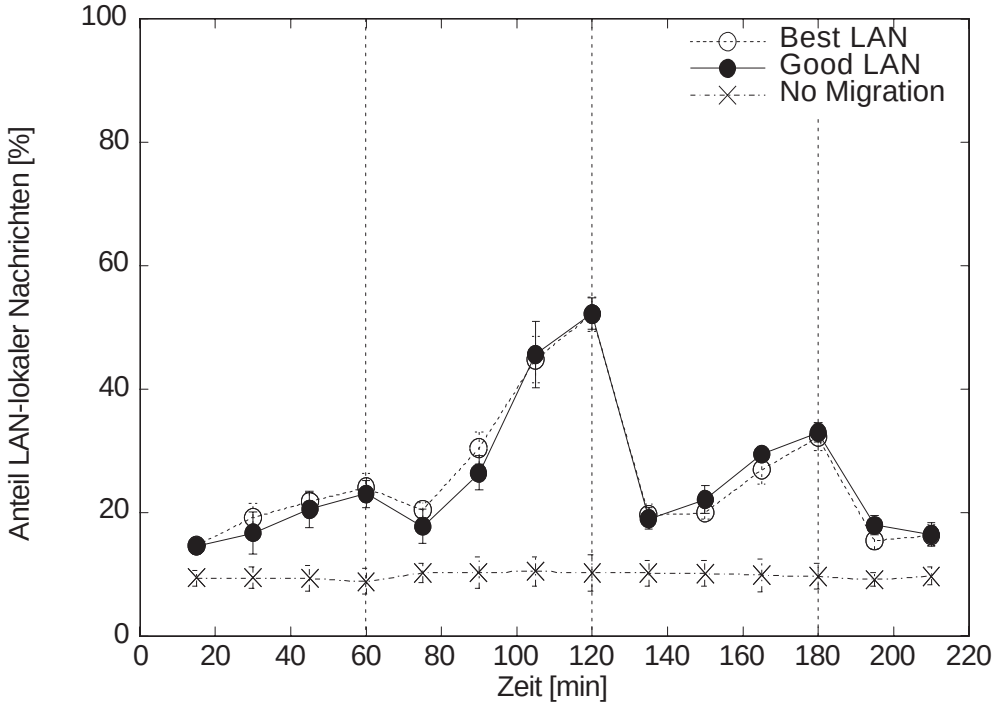


Abbildung 3.13: Anteil LAN-lokaler Nachrichten

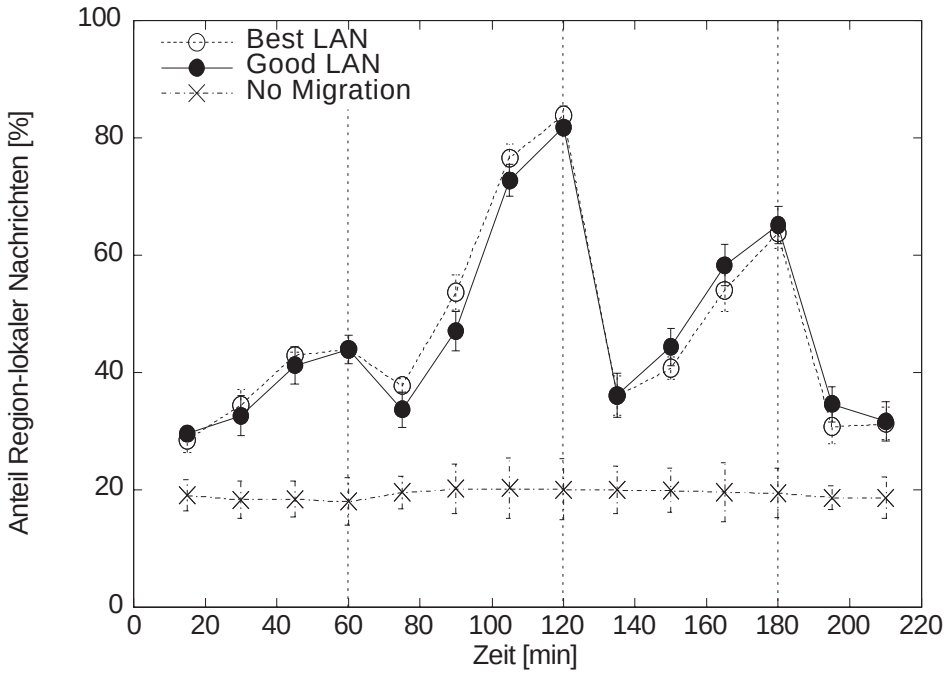


Abbildung 3.14: Anteil Region-lokaler Nachrichten

In der Zeitzone 2 dagegen befand sich nur eine Region und damit auch nur zwei LANs, auf die sich die migrierenden Objekte konzentrierten. Dadurch erhöhte sich der Anteil von LAN- und Region-lokalen Nachrichten gegenüber den anderen Zonen enorm. Aber obwohl bis zu 80% der Nachrichten Region-lokal waren, heißt das nicht, daß sich 80% der Objekte auf den Rechnern der zweiten Zone aufhielten. Die Beobachtung der Experimente ergab, daß vor allem die bevorzugten Fluglinien der aktiven Kontinente und die von ihnen angebotenen Flüge auf die aktiven Rechner migrierten. Auch einige der anderen Fluglinien migrierten auf die augenblicklich arbeitenden Knoten, da von diesen viele Zugriffe stammten. Die von ihnen angebotenen Flüge migrierten aber im allgemeinen nicht, da eine Migration unrentabel war.

In der dritten Zeitzone lagen zwei Kontinente. Das führte dazu, daß sich die Objekte, die auf Rechner innerhalb dieser Zone migrierten, auf zwei Kontinente verteilten. Aus diesem Grund entstanden höhere Kommunikationkosten als während der Aktivität innerhalb Zone 2. Im Vergleich zu Zone 1 waren allerdings die Anteile an LAN- bzw Region-lokalen Nachrichten höher, da sich die migrierenden Objekte innerhalb der Kontinente nicht auf mehrere Regionen verteilen konnten (siehe Abbildung 3.7).

Auch in diesem Experiment zeigten sich keine deutlichen Qualitätsunterschiede in den Migrationsentscheidungen der beiden Strategien. Als Fazit läßt sich feststellen, daß auch in diesem Szenario die Good-LAN Strategie aufgrund ihres besseren Kosten/Nutzen-Verhältnisses eingesetzt werden sollte.

Kapitel 4

Anbindung eines verteilten Nameservice an das AutO-System

Dieses Kapitel behandelt die Erweiterung des AutO-Systems um einen verteilten Nameservice, der von André Eickler im Rahmen seiner Dissertation [Eic98] entwickelt wurde. Bisher stand AutO nur ein zentraler Nameservice zur Verfügung, der durch ein Caching-Verfahren innerhalb der Knotenmanager unterstützt wurde. In einem weit verteilten System wie AutO stellt jede häufig genutzte zentrale Komponente früher oder später einen Flaschenhals dar und verschlechtert die Skalierbarkeit des Systems. Ein weiterer Grund, einen verteilten Nameservice zu verwenden, ist die Migration, die AutO anbietet. Migriert ein Objekt in ein LAN, ist es für die Rechner in der Umgebung des neuen Aufenthaltsortes des Objektes trotzdem nötig, die – möglicherweise hohen – Kommunikationskosten für eine Anfrage beim Nameservice in Kauf zu nehmen, um den Aufenthaltsort des Objektes zu erfahren. Dadurch geht ein Teil der durch die Migration erwirtschafteten Einsparungen an Kommunikationskosten verloren. Der verteilte Nameservice hingegen ist darauf ausgelegt, möglichst billig Objekte in der Nachbarschaft der anfragenden Rechner zu finden, und reduziert dadurch den Aufwand, z. B. Objekte im eigenen LAN zu finden.

4.1 Die Nameservice-Architektur des AutO-Systems

Das AutO-System ist lediglich über die Knotenmanager der AutO-Rechner an den Nameservice angebunden. Um verschiedene Nameservices mit AutO verwenden zu können, existiert ein Interface, daß alle hierzu nötigen Methoden spezifiziert. Dieses kann für den gewünschten Nameservice in geeigneter Form implementiert werden. Der Name einer solchen Treiberklasse kann in der Konfigurationsdatei von AutO angegeben werden, um den zu verwendenden Treiber festzulegen. Durch diese Architektur ist eine sehr flexible Anbindung verschiedener Nameservices an AutO möglich. Die Nameservice-Ansteuerung durch AutO wird in Abbildung 4.1 nochmals verdeutlicht. Der Cache wird vom Knotenmanager verwaltet, wodurch er unabhängig vom verwendeten Nameservice arbeitet. Findet ein Knotenmanager ein Objekt nicht im Cache, wird der Nameservice danach gefragt. Mit der Antwort des Nameservice wird dann der Cache aktualisiert.

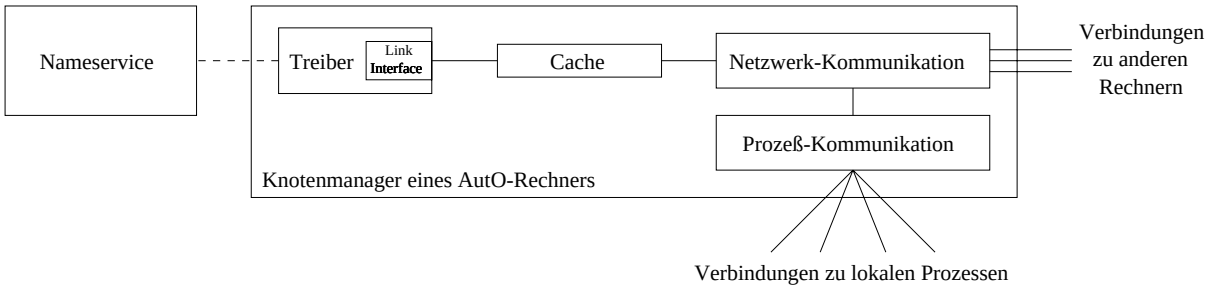


Abbildung 4.1: Anbindung eines Nameservice an AutO

Eine weitere Komponente, die direkten Zugriff auf den Nameservice hat, ist der Informationservice (siehe [Gri97]). Dieser dient unter anderem dazu, Objekten global gültige Namen zuzuweisen oder Beschreibungen von Objekttypen zu hinterlegen. Dabei bietet der Informationservice die Möglichkeit, das dem anfragenden Rechner nächstliegende registrierte Objekt eines bestimmten Typs zurückzuliefern. Um dies leisten zu können, sendet er dem Nameservice die Objektidentifikatoren (OIDs) aller in Frage kommenden Objekte, damit dieser entscheiden kann, welches Objekt das Nächstgelegene ist. Dabei existieren zwei Verbindungsarten, die je nach Art der dem Informationservice zugrundeliegenden Datenbank verwendet werden. Bei Verwendung von AutOShore kommuniziert dieses über ein proprietäres Protokoll mit dem Nameservice, bei einer JDBC-Datenbankanbindung erfolgt die Kommunikation mit dem Nameservice über das Interface, das auch die Knotenmanager verwenden. Ein allgemein einsetzbarer Nameservice muß also auch das Protokoll von AutOShore unterstützen.

4.2 Der zentrale Nameservice von AutO

Der bisher von AutO verwendete Nameservice ist ein einfacher, speziell für das AutO-Projekt entwickelter zentraler Nameserver, der seine Daten in einer Hash-Tabelle verwaltet. Diese Hash-Tabelle wird jeweils nach einer vorkonfigurierten Zeit persistent unter Verwendung der Serialisierung von Java-Objekten auf Platte geschrieben. Dies ist eine recht einfache Implementierung des Link-Interfaces, das AutO zur Ansteuerung des Nameservice verwendet, und bietet – vor allem in einem größeren System – nicht die nötige Leistung und Sicherheit. Die größten Probleme in einem realen System dürften dabei die Last auf dem Rechner, auf dem der Nameserver läuft, und die hohen Kommunikationskosten durch Anfragen an diesen sein. Die Struktur eines AutO-Systems mit einem zentralen Nameservice zeigt Abbildung 4.2.

Die Abbildung zeigt drei LANs, die einen gemeinsamen Nameserver verwenden. Rechner von LAN 1 und LAN 3 müssen für Anfragen an den Nameservice deshalb höhere Kommunikationskosten in Kauf nehmen. In einem größeren Szenario mit mehreren Regionen und Kontinenten würden für die meisten Rechner im System sehr hohe Kosten für Anfragen beim Nameserver entstehen, wodurch sich dieser Nameservice in großen Systemen nicht effektiv einsetzen läßt.

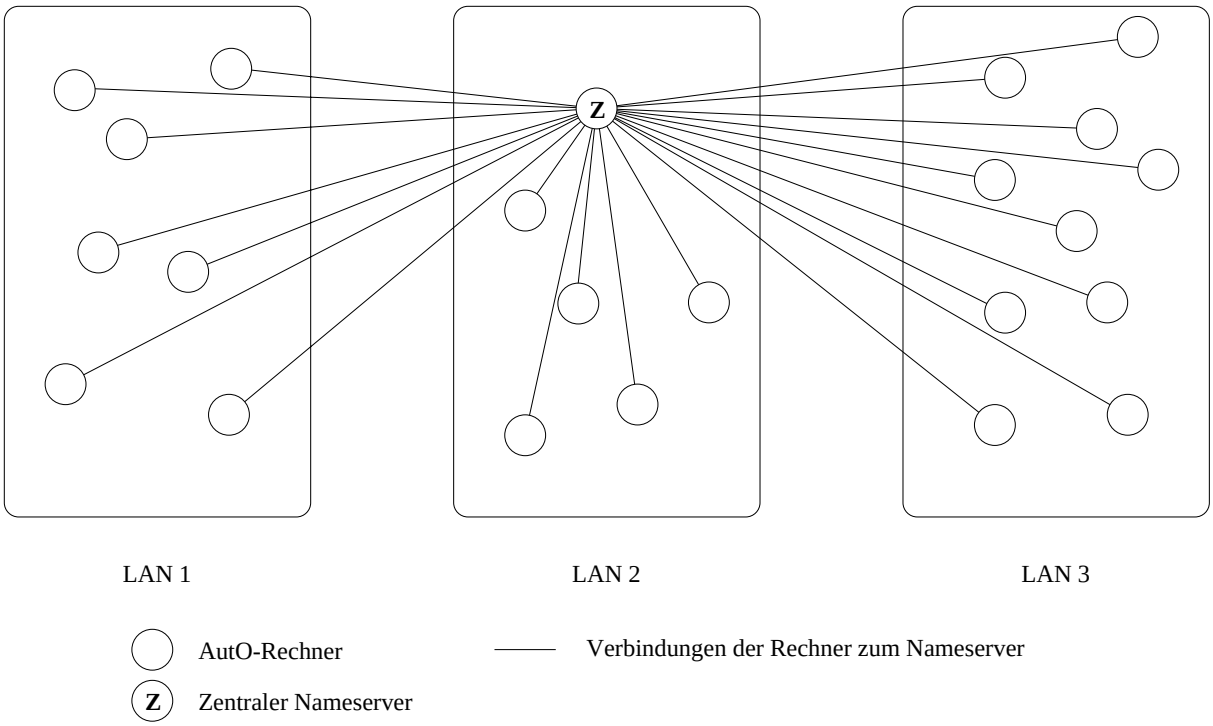


Abbildung 4.2: Beispielstruktur eines AutO-Systems mit einem zentralen Nameserver

4.3 Der verteilte Nameservice

Um den Nachteilen eines zentralen Nameservice entgegenzuwirken, wurde eine Anbindung für den verteilten Nameservice von André Eickler entwickelt. Dieser wird im folgenden beschrieben, um einen Eindruck von der Leistungsfähigkeit dieses Service zu erhalten. Für tiefgehendere Informationen wird auf [Eic98] verwiesen.

4.3.1 Aufbau des verteilten Nameservice

Der verteilte Nameservice arbeitet mit hierarchisch verknüpften Nameservern, die einen verteilten B^{link}-Baum repräsentieren[1]. Die Hierarchie der Nameserver orientiert sich dabei an der zugrundeliegenden Netzwerktopologie, wodurch typischerweise die Kommunikation eines Nameservers mit seinen direkt über- bzw. untergeordneten Servern sehr günstig ist, während die Kommunikation mit Nameservern in anderen Zweigen der Baumstruktur über mehrere Netzwerke hinweg erfolgt und dementsprechend teuer ist. Dabei speichern nur die Server, die die Blätter der Baumstruktur repräsentieren, Informationen über die Aufenthaltsorte der Objekte ab. Nameserver in den darüberliegenden Ebenen enthalten nur die Indexstrukturen, die nötig sind, um die benötigten Daten auf den darunterliegenden Servern zu finden. Jeder Blattserver ist für einen konfigurierbaren Teil der Objektidenti-

[1] Ein B^{link}-Baum ist ein B^{+}-Baum, bei dem alle Knoten einer Ebene durch Rechtsverweise verkettet sind.

fikatoren des Systems zuständig.

Für das Szenario aus Abbildung 4.2 würde sich zum Beispiel eine zweistufige Nameserver-Hierarchie anbieten, die über einen Region-Nameserver verfügt. Dieser speichert die nötigen Index-Informationen, um Anfragen an die richtigen LAN-Nameserver delegieren zu können. Die Rechner eines LANs teilen sich jeweils einen gemeinsamen Nameserver, der auf einem beliebigen Rechner innerhalb des LANs läuft. Anfragen an den Nameservice werden dabei stets an den LAN-lokalen Nameserver gestellt. Kennt dieser den Aufenthaltsort des gesuchten Objektes nicht, delegiert er die Aufgabe an den übergeordneten Server (in diesem Beispiel also an den Region-Nameserver), der in diesem Beispiel auf jeden Fall in der Lage ist, alle existierenden Objekte zu lokalisieren. Dadurch würde eine Nameserver-Hierarchie entstehen, wie in Abbildung 4.3 gezeigt.

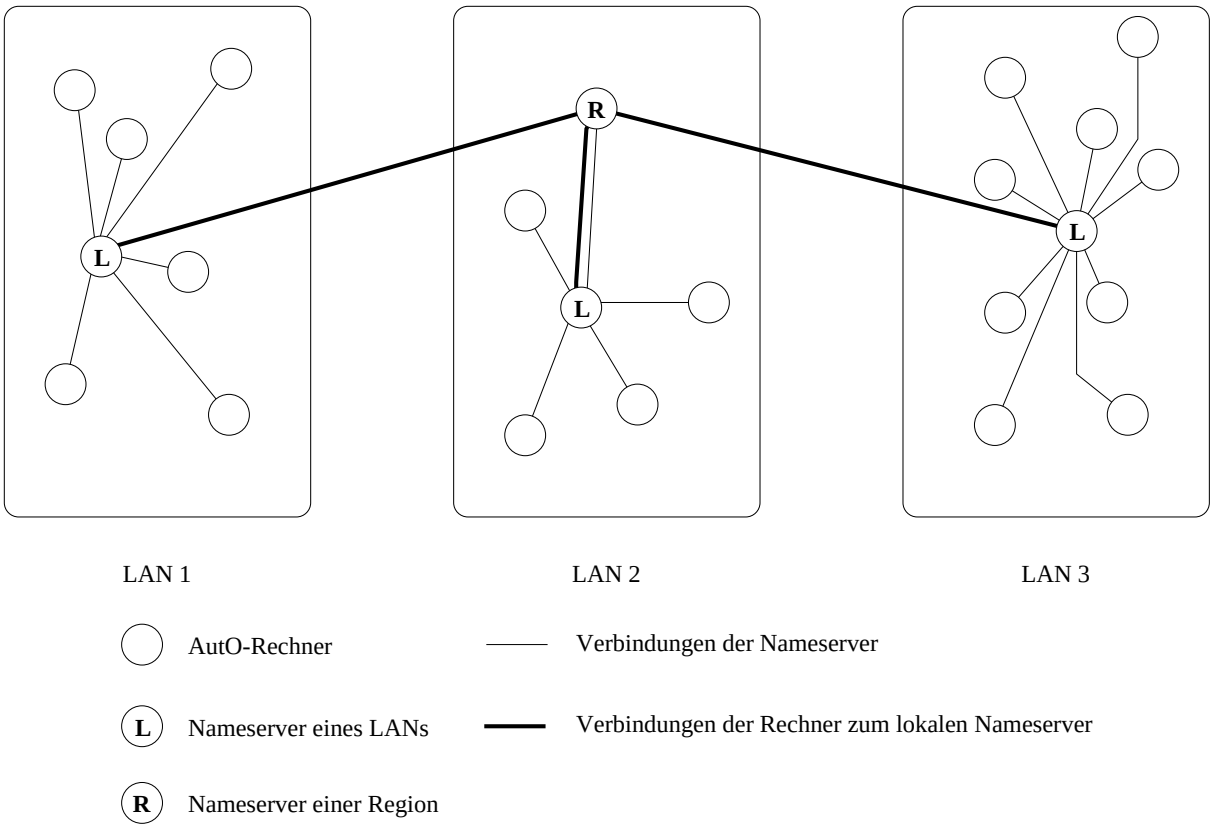


Abbildung 4.3: Beispielstruktur eines AutO-Systems mit verteiltem Nameservice

Zu beachten ist, daß Anfragen stets nur an die Blattserver gerichtet werden sollten, um Informationen über lokale Objekte möglichst billig zu erhalten. Es können auch Anfragen z. B. direkt an den Wurzelserver gestellt werden, der allerdings im allgemeinen die Anfrage nicht direkt (wenn er die Antwort nicht zufällig im Cache gespeichert hat) beantworten kann und deshalb die Anfrage weitergeben muß.

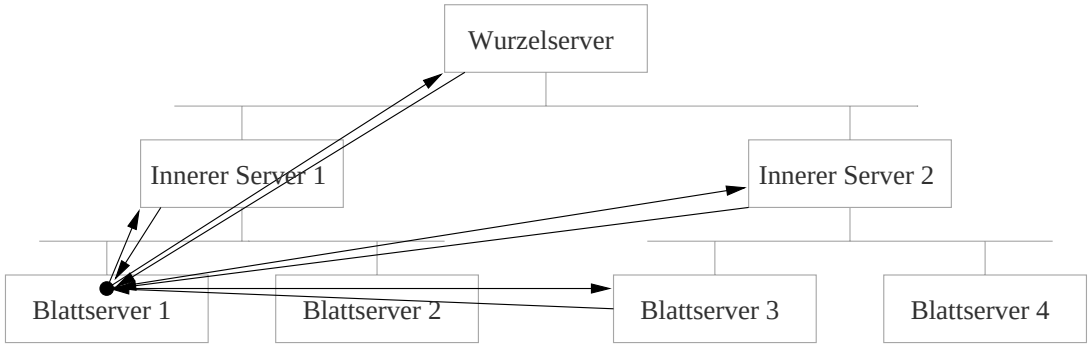
Die Verwendung von B^{link}-Bäumen ermöglicht eine sehr einfache Replikation und Pufferung von Informationen der Index-Knoten der Hierarchie. In einem größeren System ist es zum Beispiel naheliegend, die Wurzel des verteilten Nameservice massiv zu replizieren,

da die Wurzel sehr oft gebraucht, dabei aber nur sehr selten verändert wird. Auch für die Informationen der zweiten Ebene zahlen sich Replikationen in vielen Fällen aus. Die replizierten Daten werden durch asynchrone Propagierung von Änderungen aktualisiert. Der hier vorgestellte verteilte Nameservice unterstützt außerdem die Replikation von Blattknoten und bietet Mechanismen, potentiell entstehende Inkonsistenzen festzustellen und zu korrigieren. Durch die Fähigkeit, Daten von anderen Servern zu puffern, wird der Wuzelknoten des Baumes in fast allen Servern lokal vorhanden sein, was dazu führt, daß die oberen Ebenen der Hierarchie, die durchaus zu Flaschenhälsen werden können, entlastet werden. Replikation ist aber dennoch notwendig, um den Nameserver bei migrierten Objekten zu unterstützen. Betrachtet man ein Objekt O aus Passau, das nach New York migriert, weil es in Amerika oft benötigt wird, verdeutlicht das eine Schwäche der Pufferung von Daten: sie kann keinen Vorteil aus geographischer Nähe ziehen. Benötigt ein Rechner aus Los Angeles die Position des Objektes O, beantragt er diese Information normalerweise bei einem anderen Nameserver als dem, der für New York zuständig ist. Diesem liegen noch keine Daten über den neuen Aufenthaltsort von O vor, deshalb muß er in Passau nachfragen, wo das Objekt jetzt liegt. Hätte man sowohl den Blattknoten von Passau in New York repliziert als auch alle Indexknoten, die Vorfahren von dem Passauer Blattknoten und nicht gleichzeitig Vorfahren von dem New Yorker Blattknoten sind, wäre eine Suche des Objektes O innerhalb Amerikas möglich gewesen.

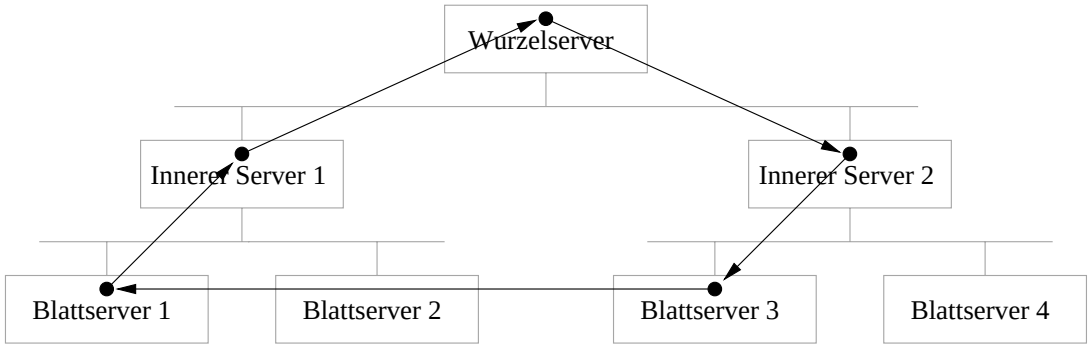
4.3.2 Die Suche im verteilten Nameservice

Der verteilte Nameservice bietet verschiedene Möglichkeiten an, Objekte in der Nameserver-Hierarchie zu lokalisieren. Prinzipiell verlangt das Auffinden eines beliebigen Objektes die Traversierung des Indexes. Dabei muß nicht, wie z. B. bei B^+-Bäumen, bei der Wurzel begonnen werden, es kann bei einem beliebigen Nameserver, also auch bei den Blattservern, begonnen werden. Dabei werden Suchanfragen solange an übergeordnete Server weiterdelegiert, bis ein Server erreicht ist, in dessen Zuständigkeitsbereich das gesuchte Objekt fällt. Im schlimmsten Fall ist das der Server, der die Wurzel des Baumes verwaltet, in anderen Fällen können das auch „innere“ Rechner oder Blattrechner der Nameserverhierarchie sein. Von diesem Rechner aus wird die Anfrage dann wieder im Baum nach unten delegiert, um das Anfrageergebnis zu erhalten. Der Vollständigkeit halber sei erwähnt, daß der Nameservice auch eine vollständige Traversierung anbietet, d. h. die Suche startet immer beim Wurzelknoten des Baumes. Da die oben erläuterte abgekürzte Traversierung aber in den meisten Anwendungsfällen eine deutliche Verbesserung der Antwortzeit des Nameservice bietet, wird im folgenden davon ausgegangen, daß eine abgekürzte Traversierung des Serverhierarchie stattfindet.

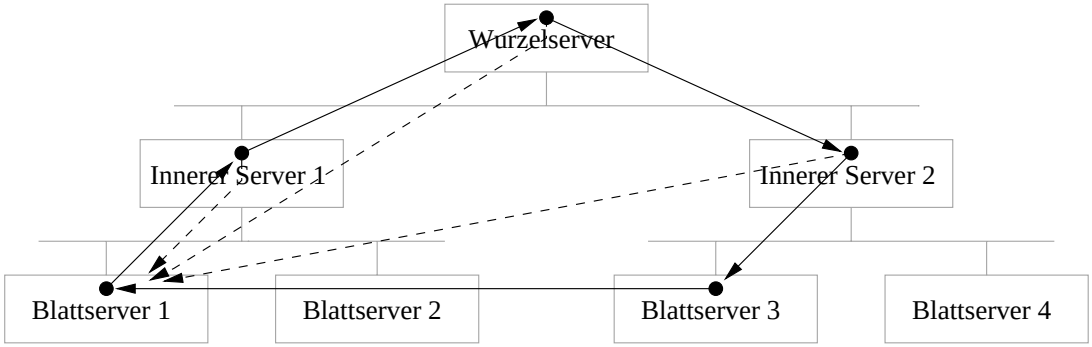
In der vorliegenden Implementierung stehen dabei drei verschiedene Arten einer abgekürzten Traversierung zur Verfügung: die iterative Suche, die rekursive Suche und die verbesserte rekursive Suche, wie in Abbildung 4.4 veranschaulicht. Abbildung 4.4a) zeigt ein Beispiel einer iterativen Suche. Dabei wird eine Anfrage an den Blattserver 1 über ein Objekt gestellt, das auf Blattserver 3 liegt. Da Blattserver 1 die Anfrage nicht selbst beantworten kann, delegiert er die Aufgabe zuerst an den inneren Server 1 und danach, da dieser auch keine Informationen über das gesuchte Objekt hat, an den Wurzelserver.



a) Iterative Suche



b) Rekursive Suche



c) Verbesserte rekursive Suche

Abbildung 4.4: Verschiedene Möglichkeiten der Traversierung der Serverhierarchie

Dieser Server liefert den relevanten Teil vom Wurzelindex des Baumes an den Initiator der Suche zurück, der daraufhin feststellen kann, daß er seine Anfrage an den inneren Server 2 delegieren muß. Aufgrund der Informationen, die dieser zurückliefert, kann Blattserver 3 befragt werden, der schließlich die gesuchte Information zurückschickt. Dabei behält immer der initiierende Server die Kontrolle über die Suche im Baum.

Bei der rekursiven Suche, die in Abbildung 4.4b) dargestellt wird, wird die Kontrolle über die Suche im Baum an andere Server übergeben. Wieder hat Blattserver 1 keine Informationen über das gesuchte Objekt und delegiert deshalb die Aufgabe an den inneren Server 1. Auch dieser hat keine verwertbaren Daten, meldet das aber nicht zurück, sondern beauftragt den Wurzelserver mit der Suche. Mit Hilfe des Wurzelindexes der verteilten Struktur ist es diesem natürlich möglich, festzustellen, daß er sich an den inneren Server 2 wenden muß, der wiederum weiß, daß Blattserver 3 die gesuchte Information besitzt. Dieser sendet dann das Ergebnis der Suche zurück an Blattserver 1, der die Suche gestartet hat. Der Vorteil der rekursiven Suche ist, daß weniger und billigere Nachrichten während der Suche gebraucht werden. Der Nachteil ist, daß im Gegensatz zur iterativen Suche, der Blattserver 1 keinen Vorteil aus der Pufferung von Informationen ziehen kann. Im iterativen Fall kann der Server nämlich die Wurzel, innere Knoten und den Blattknoten im Puffer halten, während er im rekursiven Fall nicht einmal die besuchten Rechner kennt. Aus diesem Grund gibt es noch eine verbesserte rekursive Suche, die die Pufferung von Daten unterstützt.

Diese Suchart wird in Abbildung 4.4c) verdeutlicht: Genauso wie bei der rekursiven Suche durchläuft die Suchanfrage den Baum. Allerdings senden die besuchten Server asynchron relevante Daten an den initiierenden Server zurück. Da dies rein informative Nachrichten sind, kann man auch ein unsicheres und damit ressourcenschonendes Protokoll zur Datenübertragung verwenden. Diese Möglichkeit vereint also die Vorteile der iterativen und der rekursiven Suche und wird deshalb auch bevorzugt eingesetzt.

4.3.3 Die Anbindung des Nameservice an AutO

Der Nameservice von André Eickler verwendet Integerwerte mit 64 Bit Länge als Objektidentifikatoren. Diese Objektidentifikatoren bestehen aus einem 32-Bit Präfix und einer 32-Bit Seriennummer (siehe Abbildung 4.5). Jeder Nameserver kennt die Präfixe, für die er verantwortlich ist. Die Objektidentifikatoren in AutO bestehen aus einer Internetadresse und einer Seriennummer, die 64 Bit lang ist. Die Internetadresse läßt sich in einem 32-Bit Integer kodieren und würde sich damit als Präfix im verteilten Nameservice eignen. Da aber die Blätter in einem B^{link}-Baum sortiert sind, würde der Einsatz der IP-Adressen der Rechner als Präfix die Möglichkeiten in der Bildung einer Nameserver-Hierarchie stark einschränken. Aus diesem Grund wird der 32-Bit Integer, in dem die IP-Adresse eines Rechners kodiert ist, in einer konfigurierbaren Art und Weise auf einen beliebigen anderen 32-Bit Integer abgebildet. Dabei ist natürlich darauf zu achten, daß die Abbildung zwischen den IP-Adressen und den vom Nameservice verwendeten Präfixen bijektiv ist.

Für die Seriennummern der Identifikatoren bleiben dabei allerdings nur 32-Bit übrig, wodurch die Anzahl der erzeugbaren Objekte in AutO verringert wird. Da 2^{32} Objekte

im Normalfall ausreichend sein dürften, stellt dies keine große Einschränkung dar.

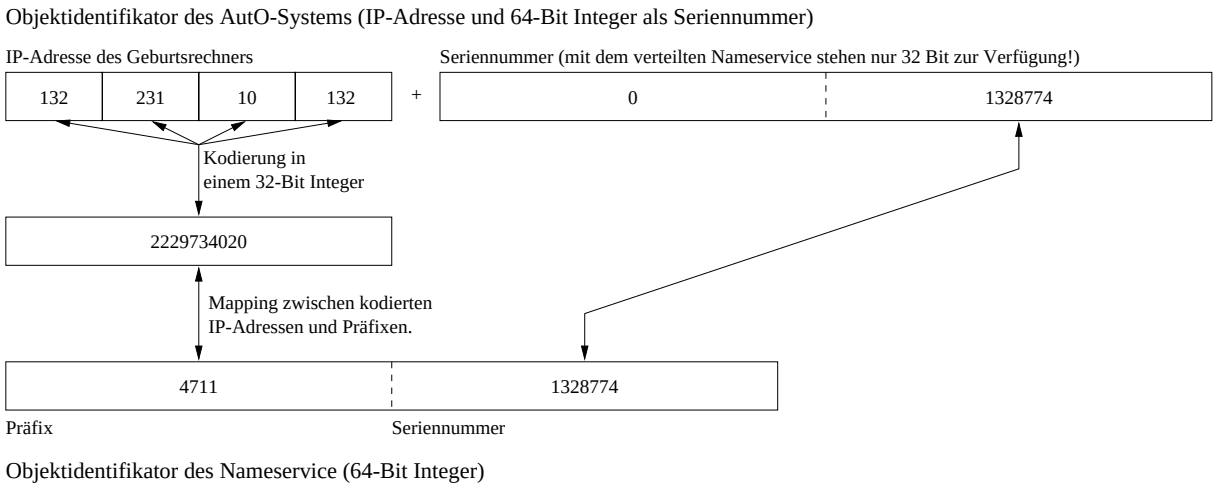


Abbildung 4.5: Mapping der OIDs zwischen AutO und dem verteilten Nameservice

Da der verteilte Nameservice Replikation und Pufferung von Daten unterstützt, ist es möglich, daß die Antwort auf eine Anfrage falsch ist, das heißt, einen falschen (früheren) Aufenthaltsort eines Objektes enthält. Aus diesem Grund kann bei Anfragen an den verteilten Nameservice angegeben werden, ob die Antwort verbindlich sein soll oder nicht. Bei einer verbindlichen Antwort wird direkt bei dem Server angefragt, der für das gesuchte Objekt verantwortlich ist. Muß die Antwort nicht unbedingt richtig sein, werden existierende Replikate und Puffer verwendet, um die Antwort zu finden, was dazu führen kann, daß veraltete Daten verwendet werden. Da der bisherige zentrale Nameservice keinen derartigen Mechanismus kannte, muß AutO so erweitert werden, daß es damit umgehen kann. Dazu wird dem Treiber-Interface für den Nameservice ein Flag hinzugefügt, das angibt, ob der Nameservice, den der Treiber ansteuert, einen derartigen Mechanismus unterstützt. Ist das nicht der Fall, funktionieren Anfragen an den Nameservice wie bisher. Bietet der Nameservice allerdings die beschriebene Möglichkeit, werden Anfragen an den Nameservice im schnelleren unverbindlichen Modus gestellt. Wird nach einer derartigen Anfrage festgestellt, daß die Information veraltet ist, weil z. B. ein Rechner eine Nachricht für ein Objekt erhält, das er gar nicht kennt, der Nameservice allerdings diese Information bestätigt, wird eine Anfrage im verbindlichen Modus gestellt. Die Antwort auf diese Anfrage beinhaltet auf jeden Fall die korrekte Information über den Aufenthaltsort des gesuchten Objektes.

Die Anbindung des Nameservice teilt sich nun in drei Aufgaben auf: Spezifikation des Kommunikationsprotokolls zwischen AutO und dem Nameservice, Implementierung eines Treibers auf der Seite von AutO und Implementierung eines multithreaded Servers auf der Basis der vorhandenen Nameservice-Bibliothek.

Das Kommunikationsprotokoll

Um eine sichere Verbindung zwischen dem AutO-System und den Servern des verteilten Nameservice zu garantieren, wird als zugrundeliegendes Netzwerkprotokoll TCP/IP verwendet. Jede Nachricht von AutO an einen Nameserver beginnt mit einem Identifikationsbyte, das angibt, um welchen Nachrichtentyp es sich handelt: Update, Lookup, Remove oder GetClosest[2]. Der Typ der Nachricht spezifiziert, welche Parameter mit der Nachricht versendet werden und wie diese zu interpretieren sind. Eine Update-Nachricht beinhaltet zum Beispiel den Objektidentifikator des Objektes und den neuen Aufenthaltsort. Die Datentypen werden zwischen AutO und dem Nameservice auf eine festgelegte Weise konvertiert. IP-Adressen werden z. B. als 32-Bit Integer an den Nameservice gesendet. Die Konvertierung aller Daten erfolgt auf der Seite von AutO, um den Server auf der Seite des Nameservice so effizient wie möglich zu halten. Die Umsetzung zwischen kodierten IP-Adressen und den vom Nameservice verwendeten Präfixen wird allerdings von den einzelnen Nameservern vorgenommen, um diesen Vorgang für AutO vollständig transparent ablaufen zu lassen und die bisher mögliche Verbindung zwischen Nameservice und Informationservice auch weiterhin zu unterstützen.

Der Nameservice-Treiber

Der Treiber auf der Seite von AutO übernimmt die Kommunikation mit dem Nameserver, der dem Knotenmanager zugewiesen ist. Dabei kann für jeden AutO-Rechner einzeln konfiguriert werden, zu welchem Nameserver er Kontakt aufnehmen soll. Da der Treiber nur sehr grundlegende Funktionen bei der Ansteuerung des Nameservers übernimmt, ist der Implementierungsaufwand relativ gering. Für jeden Anfragetyp (Update, Lookup, Remove, GetClosest) wird eine Methode implementiert, die eine Nachricht an den Nameserver in dem durch das Protokoll festgelegten Format schickt, und dabei alle eventuell notwendigen Datenkonvertierungen vornimmt. Wird bei einer Anfrage eine Antwort erwartet, wird diese in ein Format konvertiert, mit dem AutO arbeiten kann.

Der multithreaded Nameserver

Der letzte Teil der Anbindung besteht aus der Implementierung eines Servers, der Anfragen vom AutO-System entgegennimmt, an die Nameservice-Bibliothek weiterreicht und das Ergebnis der Anfragen über die Netzwerkverbindung zurücksendet. Der Server muß dabei mit mehreren Threads arbeiten, um Anfragen von verschiedenen Rechnern gleichzeitig bearbeiten zu können. Dazu wird für jede Verbindung, die zu einem Nameserver aufgebaut wird, ein neuer Thread erzeugt, der sämtliche Aufträge, die über diese neue Verbindung eintreffen, abarbeitet. Dabei gibt es zwei verschiedene Verbindungstypen: Verbindungen, die das Nameservice-Protokoll des AutO-Systems verwenden, und Verbindungen von AutOShore. Je nach Verbindungstyp wird das entsprechende Protokoll verwendet, um den Nachrichtentyp und die Parameter festzustellen. Diese werden an

[2]GetClosest liefert als Antwort ein Objekt eines gegebenen Typs, das dem anfragenden Rechner am nächsten liegt.

die passende Funktion der Nameservice-Bibliothek übergeben. Besitzen diese Methoden Rückgabewerte, werden diese entsprechend dem Protokoll in Nachrichten verpackt und an den Knotenmanager zurückgesendet.

Der zweite Verbindungstyp existiert nur, wenn AutOShore als Informationservice verwendet wird. AutOShore verwendet ein proprietäres Protokoll, um mit Hife des Nameservice aus einer Menge von Objekten das zu finden, das einem bestimmten Rechner am nächsten liegt. Um auch mit dem verteilten Nameservice diese Möglichkeit bieten zu können, wurde das proprietäre Protokoll von AutOShore im Nameserver implementiert. Dadurch ist die transparente Verwendung des verteilten Nameservice sowohl vom AutO-System selbst als auch vom Informationservice möglich.

4.4 Einsatz des verteilten Nameservice

Um den verteilten Nameservice zu verwenden, muß sowohl das AutO-System als auch der Nameservice entsprechend konfiguriert werden. Auf der Seite von AutO muß allen Knotenmanagern die Adresse eines Nameservers mitgeteilt werden, zu dem sie eine Verbindung aufbauen sollen.

Der verteilte Nameservice benötigt einen hierarchischen Aufbau seiner Server. Wie schon in Abschnitt 4.3 beschrieben, bietet es sich dabei an, sich an der Hierarchie zu orientieren, die die Good-Lan Strategie (siehe Abschnitt 3.2) verwendet. Ein sinnvoller Aufbau ist dabei, in jedem LAN einen eigenen Nameserver einzusetzen, der die Anfragen aller Rechner aus diesem LAN bearbeitet. Dabei kann der Rechner, auf dem der Nameserver läuft, durchaus auch für AutO genutzt werden. Er sollte so ausgewählt werden, daß er möglichst gute Kommunikationsverbindungen zu den anderen Rechnern im LAN hat. In jeder Region wird dann ein Rechner gesucht, der als übergeordneter Nameserver dient, um auf die Informationen aller direkt untergeordneten Nameserver in der Region zugreifen zu können. Auch dieser Rechner sollte natürlich möglichst geringe Kosten für die Kommunikation mit den LAN-Nameservern haben. Nach dem gleichen Prinzip wird ein Rechner pro Kontinent gesucht, der den Index für alle Region-Server verwaltet. Schließlich und endlich muß ein Rechner bestimmt werden, der die Verwaltung der Wurzel des verteilten Indexes übernimmt. Da die Kosten für interkontinentale Kommunikation normalerweise sehr hoch sind, ist es sinnvoll, die Informationen des Wurzelknotens auf allen Kontinent-Nameservern zu replizieren, die nicht im selben Kontinent liegen wie der Wurzelserver selbst. Damit wird verhindert, daß die Rechner benachteiligt werden, die in einem anderen Kontinent liegen als der Wurzel-Nameserver. Hat man eine geeignete Hierarchie gefunden, muß noch eine Konfigurationsdatei für den Nameservice geschrieben werden, die angibt, wie die bijektive Abbildung zwischen den IP-Adressen der Rechner und den vom Nameservice verwendeten Präfixen aussieht. Dazu werden beispielsweise die Rechner in der Reihenfolge durchnummeriert, in der sie als Blätter in der Hierarchie erscheinen. Eine schematische Darstellung einer solchen Konfiguration zeigt Abbildung 4.6.

Falls die Kosten für die Kommunikation der Nameserver auch innerhalb eines Kontinents zu hoch sind, kann man sowohl die Daten der Wurzel als auch die der Kontinent-Nameserver auf die Region-Nameserver replizieren. Da beide relativ selten geändert wer-

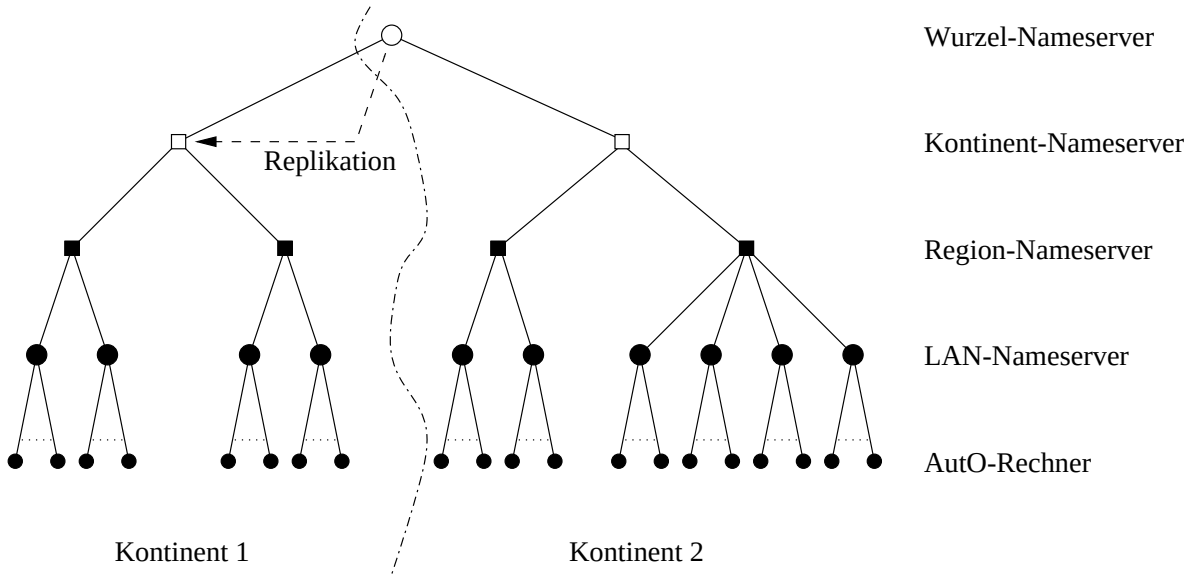


Abbildung 4.6: Mögliche Hierarchie der Nameserver

den, fällt normalerweise die Ersparnis an Kommunikationskosten höher aus als die Kosten für die Konsistenzhaltung der Replikate. Ob sich Replikate der Daten tieferliegender Nameserver auszahlen, hängt sehr stark von der konkreten Anwendung ab. In einem Szenario wie in Abschnitt 3.5.2 beschrieben, in dem die Objekte regelmäßig wegen der Lastverschiebung migrieren, dürfte sich eine Replikation der tieferen Ebenen negativ auswirken. Die Kommunikationskosten, die im Nameservice bei jeder Migration anfallen, um alle Replikate konsistent zu halten, würden über dem zu erwartenden Gewinn liegen. Außerdem ist eine derartige Replikation in den meisten Fällen nicht nötig, da die Informationen über die Aufenthaltsorte der am häufigsten benötigten Objekte sowohl von AutO als auch von den Nameservern gepuffert werden und dadurch den effizienten Zugriff auf diese Informationen ermöglichen.

Kapitel 5

Sicherheit in der Migration

Dieses Kapitel beschäftigt sich mit der Erweiterung der bestehenden Sicherheitskomponente, die von Markus Keidl [Kei99] geplant und implementiert worden ist. Diese Komponente bietet Autorisierung und sichere Kommunikation. Da die Migrationskomponente in AutO spezielle Anforderungen an die Sicherheitskomponente stellt, sind zusätzliche Maßnahmen nötig, um keine Sicherheitslücken entstehen zu lassen. Zum einen ist es nötig, die Auswahl des Zielrechners einer Migration gezielt beeinflussen zu können, zum anderen müssen die Aktionen autonomer Objekte überwacht werden, um die Rechner, auf denen das AutO-System läuft, vor Angriffen zu schützen. Mit dem erweiterten Sicherheitsmodell kann AutO auch mit Migration in einem sicherheitsrelevanten Umfeld eingesetzt werden.

Für das Verständnis dieses Kapitels ist es nötig, das bisherige Sicherheitskonzepet, das in [Kei99] beschrieben wird, zu kennen. Grundlegendes Wissen über die zur Realisierung dieses Sicherheitskonzeptes verwendeten Verfahren, wie z. B. Verschlüsselung und digitale Signaturen, wird in dieser Arbeit vorausgesetzt. Informationen dazu finden sich zum Beispiel in [Kei99, DH76, Rul93]. In Java werden kryptographische Methoden durch die *Java Cryptography Extension* (JCE) zur Verfügung gestellt [Sun98b, Sun97a, Sun97b].

Wie bereits angedeutet, gliedert sich die Erweiterung des Sicherheitskonzepts in zwei Teile, die in den nachfolgenden Abschnitten ausführlich behandelt werden:

Objektschutz: Die Daten der autonomen Objekte sollen vor unbefugtem Zugriff geschützt werden. Das bestehende Sicherheitssystem unterbindet den unautorisierten Zugriff durch die Benutzer von AutO und verhindert durch Verschlüsselung, daß die Daten eines migrierenden Objektes gelesen oder manipuliert werden können. Zusätzlich soll jetzt verhindert werden, daß Objekte auf unsichere Rechner migrieren, da die Gefahr besteht, daß dort die Daten eines Objektes direkt aus der Datenbank oder dem Speicher gelesen werden können.

Rechnerschutz: Außer dem Schutz der Objekte ist auch der Schutz der Rechner nötig, auf denen sie residieren. Dazu überwacht AutO die autonomen Objekte während ihrer Ausführung und unterbindet bei Bedarf verbotene Aktionen. Um diese Aufgabe möglichst gut zu lösen, bietet es sich an, auf die in Java 1.2 integrierten Sicherheitsmechanismen zurückzugreifen und diese für die Zwecke von AutO einzusetzen.

5.1 Sicherheit in Java

Java bietet, wie z. B. auch Telescript [TV96], ein leistungsfähiges Sicherheitsmodell, das es erlaubt, den Zugriff auf System-Ressourcen zu kontrollieren [Oak98, Pos98, SDBT]. Gedacht ist dieses Modell speziell für Probleme, die mobiler Code (in diesem Falle Applets) verursacht, der aus dem Internet geladen wird und deshalb nicht uneingeschränkten Zugriff auf den Host-Rechner, also auf den Rechner, auf dem er ausgeführt wird, haben soll. In Java 1.2 wurde dieses Modell gegenüber der Version 1.1 stark erweitert und kann nun auch ohne großen Aufwand für normale Anwendungen verwendet werden[1]. Dieses Sicherheitsmodell beruht auf der Verwendung geschützter Bereiche („Sandkasten“ oder „Sandbox“) [Kop98], in denen die Anwendungen ausgeführt werden. Objekte dürfen nur innerhalb der Grenzen, die ihnen durch den Sandkasten vorgegeben sind, agieren. Unerlaubte Ausbruchversuche aus dem geschützten Bereich bestraft die Laufzeitumgebung mit einer *SecurityException*. Um Java die Möglichkeit zu geben, alle Ausbruchversuche zu erkennen, ist das Sandkastenmodell durch fünf Schichten realisiert (siehe Abbildung 5.1), auf die nun kurz eingegangen werden soll [Oak98]:

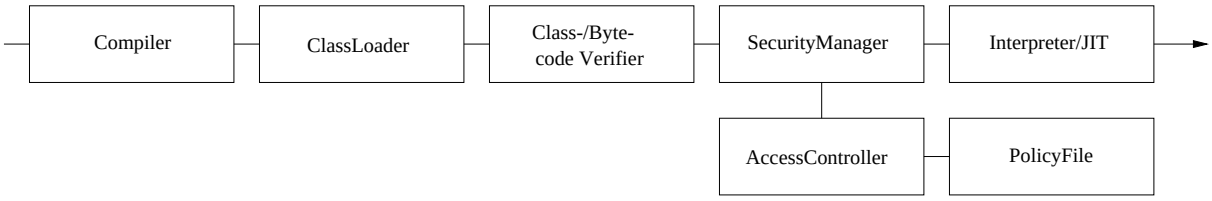


Abbildung 5.1: Die fünf Schichten des Sandkastenmodells.

Compiler: Der *Compiler* bildet die erste Schicht der Sandbox und ist für die grundsätzliche Einhaltung der Sprachsemantik von Java zuständig. Um diese zu gewährleisten, kümmert er sich um folgende Punkte:

- Die Zugriffsrechte der Klassen, Methoden und Variablen (*private, protected, public, package*) müssen strikt eingehalten werden.
- Programme dürfen nicht auf beliebige Speicherstellen zugreifen. Dies ist relativ einfach sicherzustellen, da Java keine expliziten Zeiger unterstützt.
- Methoden, Variablen und Klassen, die als *final* gekennzeichnet wurden, dürfen nicht mehr geändert werden. Bei Variablen heißt das, daß sich deren Wert nach der Initialisierung nicht mehr ändern darf. Methoden, die als *final* gekennzeichnet wurden, dürfen in abgeleiteten Klassen nicht überschrieben werden, von *final*-Klassen dürfen keine weiteren Klassen abgeleitet werden.
- Variablen dürfen nicht verwendet werden, bevor sie initialisiert worden sind. Wäre das nicht der Fall, könnte eine Klasse viele uninitialisierte Variablen

[1] Im Prinzip kann man das meiste, das Java 1.2 an Neuerungen im Bezug auf Sicherheit bietet, auch in Java 1.1 programmieren. Java 1.2 bietet aber viele neue Klassen, die die Implementierung eigener Sicherheitsstrategien sicherer und einfacher machen.

anlegen und diese benutzen, um Daten, die vorher einmal in den Speicher geschrieben worden sind, auszukundschaften.

- Es dürfen nur Typkonvertierungen (Casts) innerhalb der Vererbungshierarchie ausgeführt werden. Das heißt, einem Objekt kann nur dann ein anderer Typ zugeweisen werden, wenn der Zieltyp mit dem tatsächlichen Objekttyp übereinstimmt oder eine Oberklasse des tatsächlichen Objekttyps ist. Diese Prüfung kann vom Compiler nur eingeschränkt vorgenommen werden, die Laufzeitumgebung prüft dynamisch die Korrektheit von Casts, die statisch nicht entschieden werden konnte.

Da es möglich ist, z. B. auch durch einen modifizierten C++ Compiler Java-Bytecode zu erzeugen, ist es nötig, Prüfungen, die der Compiler vornimmt, in einer höheren Schicht des Sicherheitsmodells zu wiederholen. Die Prüfungen des Compilers sind in erster Linie als Unterstützung für Programmierer gedacht, die Fehler in ihren Programmen dadurch schon nach dem Compilieren und nicht erst bei der Ausführung des Programms bemerken können.

ClassLoader: Der *ClassLoader* kümmert sich um das Laden von Klassen. In Java 1.2 werden standardmäßig nur die Systemklassen vom internen ClassLoader der JVM geladen, alle anderen Klassen werden durch eine Instanz der Klasse `URLClassLoader` oder einer benutzerdefinierten Klasse geladen. Dabei definiert jeder ClassLoader seinen eigenen Namensraum. Dies wird z. B. in Browsern verwendet, die Applets ausführen. Jedes Applet wird durch eine eigene Instanz eines ClassLoaders geladen, wodurch sichergestellt ist, daß die Applets stets mit den richtigen Klassen arbeiten und keinen Zugriff auf die Klassen anderer Applets oder die internen Klassen des Browsers haben.

Die ClassLoader in Java 1.2 können hierarchisch verknüpft werden. Ein untergeordneter ClassLoader beauftragt stets den in der Hierarchie darüberliegenden Klassenlader, die Klasse zu laden. Nur in dem Fall, daß dies scheitert, lädt er die Klasse selbst. Eine weitere neue Eigenschaft der ClassLoader in Java 1.2 ist, daß sie die Informationen, woher die Klasse stammt und von wem sie signiert wurde, an die geladene Klasse weitergeben. Somit stehen diese Informationen dem *SecurityManager* zur Verfügung.

Class-/Bytecode Verifier: Beim Laden einer Klasse werden standardmäßig alle Klassen (außer den Systemklassen) durch den *Verifier* auf die Einhaltung der Sprachsemantik von Java überprüft. Dies ist nötig, weil der Code von einem nicht vertrauenswürdigen Java-Compiler stammen könnte oder unter Umständen nachträglich modifiziert wurde. Im einzelnen prüft der Verifier bei einer Klasse folgende Punkte:

- Die Klasse muß im korrekten Format vorliegen. Dies beinhaltet die korrekte Länge der Datei, die richtigen „magic numbers“ an den richtigen Stellen, u.s.w.
- Die Einhaltung des *final*-Modifiers bei Variablen, Methoden und Klassen muß gewährleistet sein.
- Jede Klasse (außer `java.lang.Object`) muß von genau einer Oberklasse abgeleitet sein.

- Es dürfen keine unerlaubten Datenkonversionen von Sorten (z. B. `int` oder `long`) vorkommen (z. B. `int` → `java.lang.Object`).
- Die Korrektheit der Typkonvertierungen von Objekten wird im selben Umfang geprüft wie vom Compiler.
- Es dürfen keine Über- oder Unterläufe des Operandenstacks eines Threads auftreten (Jeder Thread in Java hat zwei Stacks: den Datenstack und den Operandenstack!).

SecurityManager: Der *SecurityManager* implementiert die Sicherheitsstrategie einer Java-Anwendung, legt also die Grenzen des Sandkastens fest, in dem diese laufen soll. Dazu bietet er mehrere *checkXXX*-Methoden, die die Zugriffe auf verschiedene Ressourcen regeln. Diese Methoden werden von den Systemklassen von Java aufgerufen, bevor sie auf das Netzwerk, das Dateisystem, den Drucker oder ähnliche Komponenten zugreifen. Dadurch hat der SecurityManager die Möglichkeit, vor der Ausführung einer dieser Operationen sein Veto einzulegen und eine *SecurityException* zu werfen. Natürlich überstimmt die Genehmigung einer Aktion durch den SecurityManager nicht die Entscheidung des Betriebssystems; man kann also z. B. auf UNIX-Rechnern nicht Dateien lesen, für die der Benutzer, der die Java-Anwendung gestartet hat, keine Leserechte besitzt, nur weil der SecurityManager den Lesezugriff prinzipiell erlauben würde.

In Java 1.2 ist die in früheren Versionen abstrakte Klasse `SecurityManager` standardmäßig implementiert. Dieser konkrete SecurityManager vereinfacht die Definition einer eigenen Sicherheitsstrategie enorm und fällt seine Entscheidungen unter Verwendung des *AccessControllers*. Dieser verwendet seinerseits ein *Policy*-Objekt, das die eigentliche Sicherheitsstrategie implementiert. Im Lieferumfang von Java ist eine Klasse `PolicyFile` vorhanden, die eine Konfigurationsdatei einliest und darauf basierend Klassen verschiedene Rechte zuweist. Dabei kann auch darauf Rücksicht genommen werden, ob die Klassen signiert vorliegen oder nicht. Standardmäßig ist dabei Klassen aus dem *Classpath* jeglicher Zugriff auf Systemressourcen gestattet. Klassen, die über einen `URLClassLoader` geladen werden, haben standardmäßig nur die Erlaubnis, einige wenige Systemproperties (z. B. die Version der JVM) zu lesen.

Interpreter/JIT: Die Laufzeitumgebung von Java übernimmt die letzten nötigen Überprüfungen des Bytecodes. Es ist, wie bereits erwähnt, während des Übersetzens einer Klasse nicht möglich, alle Casts statisch zu prüfen; in diesen Fällen übernimmt die Laufzeitumgebung dynamisch die Überprüfung der vorgenommenen Casts und wirft eine *ClassCastException* in dem Fall, daß illegale Casts ausgeführt werden sollen. Außerdem prüft die Laufzeitumgebung bei Zugriffen auf Arrays, ob sie innerhalb der Arraygrenzen liegen, da ansonsten beliebig auf Speicherstellen im Hauptspeicher zugegriffen werden könnte.

Der JIT-Compiler (Just-in-time-Compiler) beschleunigt die Ausführung des Codes um bis zum Faktor 50, ohne dabei allerdings die Semantik der Java-Sprache zu ändern. Deshalb ist es irrelevant, ob eine Anwendung durch den Java-Interpreter oder einen JIT-Compiler ausgeführt wird, die Sicherheit des Sandkastens ist auf jeden Fall gewährleistet.

5.2 Objektsicherheit

In diesem Abschnitt werden verschiedene Ansätze vorgestellt, die die Sicherheit der autonomen Objekte garantieren. Dieser Aspekt der Sicherheit mußte im AutO-System ohne Migration nicht betrachtet werden, da alle Objekte auf dem Rechner blieben, auf dem sie erzeugt wurden. In einem System mit Migration ist es nötig, auf die Sicherheit der Objekte zu achten, da sonst Angriffe auf die Daten eines Objektes, unter Umgehung der bisherigen Sicherheitsmechanismen wie z. B. Autorisierung, möglich wären. Migriert ein Objekt *TopSecretFirmaA* z. B. auf einen Rechner der Firma B, könnte der dort zuständige Systemadministrator die Daten des Objektes direkt aus der Datenbank oder dem Hauptspeicher lesen. Wäre keine Migration möglich, könnte das nur der Systemadministrator der Firma A, der aus vielfältigen Gründen eine Vertrauensperson sein muß.

In den folgenden Abschnitten werden zuerst die Möglichkeiten vorgestellt, die das ursprüngliche Sicherheitsmodell geboten hat, und dann die verschiedenen Alternativen, deren Verwendung in AutO in Erwägung gezogen wurde.

5.2.1 Das bisherige Sicherheitskonzept

Mit dem ursprünglichen Sicherheitskonzept, das nicht für migrierende Objekte spezifiziert wurde, kann man lediglich feststellen, ob ein Rechner bzw. der darauf laufende Knotenmanager autorisiert ist, mit dem AutO-System zu kommunizieren. Dies ist damit die einzige verwertbare Information, die nur eine sehr rudimentäre Klassifikation der Rechner im Netz ermöglicht. Die resultierende Einteilung der Rechner kann damit nur Objektschutz auf unterstem Niveau bieten:

Autorisierte Rechner: Diese Rechner besitzen Node-Manager, deren öffentliche Schlüssel durch den Signaturserver des AutO-Systems signiert worden sind [Kei99]. Stehen keine weiteren Informationen zur Verfügung, muß die Migration auf diese Klasse von Rechnern zugelassen werden.

Nicht autorisierte Rechner: Auf diesen Rechnern laufen Knotenmanager, die keinen signierten öffentlichen Schlüssel besitzen. Um zumindest ein Mindestmaß an Sicherheit zu gewährleisten, dürfen autonome Objekte nicht auf diese Rechner migrieren. Dies wird schon dadurch erreicht, daß die Kommunikation derartiger Rechner mit dem AutO-System unterbunden wird.

Rechner ohne Knotenmanager: Rechner ohne aktiven Knotenmanager kommunizieren nicht mit dem AutO-System und stehen deshalb auch nicht als Migrationsziel zur Verfügung. Gehen gefälschte Nachrichten von einem derartigen Rechner ein, kümmert sich das bereits bestehende Sicherheitssystem darum, daß sie nicht beachtet werden. Diese Rechner-Klasse wird es auch in den nachfolgenden Sicherheitskonzepten geben, ohne daß speziell auf sie eingegangen wird.

Eine weitere Möglichkeit, regulierend auf die Migrationskomponente einzuwirken, ist die Verwendung einer speziellen Migrationsstrategie, die die Migration von autonomen Ob-

jekten unterbindet. Dadurch ist es zumindest möglich, einzelne Objekte auf einem dafür vorgesehenen und entsprechend abgesicherten Rechner zu halten und somit Angriffe auf diese Objekte zu verhindern. Auf diese Art und Weise erhält man zwei Klassen von Objekten:

Objekte mit niedrigen Sicherheitsanforderungen: Diese Objekte besitzen eine normale Migrationsstrategie und können somit auf alle autorisierten Rechner im System migrieren.

Objekte mit hohen Sicherheitsanforderungen: Diesen Objekten ist die „*Immobile-Strategy*“ zugeordnet, die verhindert, daß sie selbständig auf andere Rechner migrieren[2].

Mit der beschriebenen groben Klassifikation von Rechnern und Objekten steht nur eine sehr beschränkte Kontrolle der Migration zur Verfügung. Um ein Objekt zu schützen, muß es stationär auf einem festgelegten Rechner positioniert werden. Damit wird dem Objekt – und auch teilweise dem Gesamtsystem – die Möglichkeit genommen, sich selbst zu optimieren. Gibt es in einem System mehrere dieser unbeweglichen Objekte, was in jeder realen Installation wohl der Fall wäre, zieht das einige Probleme nach sich:

- Der Aufwand für einen AutO-Administrator, der für die manuelle Optimierung des Systems zuständig ist, wird schnell unzumutbare Ausmaße annehmen. Er muß die stationären Objekte „per Hand“ auf sichere Rechner migrieren, die für das aktuelle Zugriffsverhalten im System angemessen erscheinen. Diese Vorgehensweise widerspricht dabei allerdings zwei grundlegenden Konzepten in AutO, nämlich der Autonomie der Objekte und der Selbstoptimierung des Systems.
- Die stationären Objekte können für andere Objekte zu „Magneten“ werden, wenn diese oft auf stationäre Objekte zugreifen. Dieser Effekt ist allerdings sehr stark von der Anwendung des Systems abhängig. Im Normalfall ist die Zahl der Zugriffe von Objekten auf andere gering.

Wie hier gezeigt, bietet AutO bisher nur ungenügende Möglichkeiten der Einflußnahme auf die Migrationskomponente, wodurch eine flexible Steuerung der Objektmigration nicht möglich ist. In den folgenden Abschnitten werden deshalb verschiedene Ansätze vorgestellt, die es ermöglichen, flexibel und effizient auf die Migrationskomponente einzuwirken, und dadurch die Autonomie der Objekte weniger stark zu beeinträchtigen als durch ein Migrationsverbot. Dabei arbeiten alle drei im folgenden vorgestellten Methoden nach demselben Prinzip, lediglich die Klassifikation der Objekte und Rechner wird nach verschiedenen Gesichtspunkten vorgenommen. Alle drei Möglichkeiten werden daraufhin untersucht, ob sie bei vertretbarem Verwaltungsaufwand flexibel genug sind, in AutO eingesetzt zu werden.

[2] Man muß allerdings beachten, daß ein solches Objekt trotz allem migriert, wenn es eine Nachricht mit einer Migrationsaufforderung von einer dazu autorisierten Person erhält. Außerdem muß man sicherstellen, daß es nicht unautorisiert möglich ist, dem Objekt eine andere Migrationsstrategie zuzuweisen. Dies kann durch eine entsprechende Konfiguration der Autorisierungskomponente in AutO erreicht werden.

5.2.2 Sicherheitsstufen für Objekte und Rechner

Dieses Verfahren basiert auf der Klassifikation von Rechnern und Objekten unter Verwendung sogenannter Sicherheitsstufen. Diese sind folgendermaßen definiert:

Definition 1 (Sicherheitsstufe) Eine Sicherheitsstufe S wird festgelegt durch eine Zahl $x \in \{0, \ldots, 10\}$. Die Sicherheitsstufe 0 steht dabei für „*unklassifiziert*“ und ist damit der Standardwert für alle Rechner und Objekte, denen nicht ausdrücklich eine andere Sicherheitsstufe zugewiesen wurde. Die Sicherheitsstufe 10 markiert das oberste Sicherheitsniveau und steht damit für „*top secret*“ bzw. für „*höchst vertrauenswürdiger Rechner*“.

Diese Einteilung der Rechner und Objekte in verschiedene Sicherheitsstufen erlaubt nun eine feinere Beeinflussung der Migration, als es bisher möglich war: Will ein Objekt mit der Sicherheitsstufe S_O auf einen Rechner der Sicherheitsstufe S_R migrieren, ist das nur dann erlaubt, wenn die Sicherheit, die der gewünschte Rechner R bietet, größer ist, als die vom Objekt geforderte Sicherheitsstufe. Das ist genau dann der Fall, wenn $S_O \leq S_R$ ist.

Der Programmierer autonomer Objekte hat damit die Möglichkeit (und auch die Aufgabe), den Objekten Sicherheitsstufen zuzuordnen. Diese initiale Sicherheitsstufe kann dann im Bedarfsfall vom AutO-Sicherheitsadministrator oder einer anderen dazu autorisierten Person geändert werden. Ein bei weitem schwierigeres Problem ist die Klassifikation der Rechner im System. Diese muß von einem AutO-Administrator bei der Autorisierung des zugehörigen Knotenmanagers vorgenommen werden, da die Sicherheitsstufe von mehreren Faktoren abhängt:

- Verwendetes Betriebssystem (verschiedene Betriebssysteme bieten unterschiedlich gute Sicherheitsmechanismen).
- Vertrauen in die Person, die den Knotenmanager und das gesamte AutO-System startet und administriert.
- Standort des Rechners.
- Ausfallsicherheit des Rechners.
- Die von AutO verwendete Datenbank.

Wurde einem Rechner nach sorgfältiger Prüfung eine Sicherheitsstufe zugeordnet, muß diese Information mit dem geheimen Schlüssel eines systemweit gültigen Schlüsselpaares signiert werden, da es nur so für ein migrationswilliges Objekt möglich ist, die Gültigkeit der Sicherheitsstufe des Zielrechners zu überprüfen.[3]

Bei einer derartigen Erweiterung der Sicherheitskomponente sieht der allgemeine Verlauf einer Migration folgendermaßen aus:

[3] Auf die Signierung der Sicherheitsklassifikation eines Rechners wird im Abschnitt 5.6 genauer eingegangen.

- Objekt O wertet seine Migrationsstrategie aus und stellt fest, daß es auf den Rechner R migrieren will.
- O schickt eine Nachricht an R, um diesem den Migrationswunsch mitzuteilen. Dabei sendet O seine eigene Sicherheitseinstufung mit.
- Falls R das Objekt akzeptiert, sendet dieser als Antwort seine signierte Sicherheitsstufe S_R zurück. Andernfalls lehnt er eine Migration ab (z. B. wegen einer zu geringen Klassifikation von R oder wegen zu hoher Last) und beendet diese damit.
- Das Objekt O überprüft, ob S_R korrekt ist, d. h. es überprüft die digitale Signatur der Sicherheitsstufe.
- Falls $S_O \leq S_R$ ist, migriert O auf den Rechner R, andernfalls bricht O die Migration ab.

Im Vergleich zu der bisherigen groben Klassifikation ergibt sich mit diesem Verfahren eine feinere Migrationssteuerung, die innerhalb einer kleinen Firma sicher ausreichend wäre, um sensible Daten vor unberechtigtem Zugriff zu schützen.

Problematisch ist die Verwendung von Sicherheitsstufen dann, wenn man sie in einem größeren System einsetzen will. Dort hat man verschiedene Objekte, die gleichermaßen schützenswert sind, z. B. das Personalverzeichnis und die Produktionsgeheimnisse einer Firma. Beide sind gleichermaßen schützenswert, wenn auch aus verschiedenen Gründen: das Personalverzeichnis aus Gründen des gesetzlich vorgeschriebenen Datenschutzes, die Produktionsverfahren, weil die Firma an deren Geheimhaltung enormes wirtschaftliches Interesse hat. Geht man nun davon aus, daß die Computer der verschiedenen Abteilungen einer Firma von verschiedenen Administratoren verwaltet werden, sollte es möglich sein, Objekte innerhalb „ihrer“ Abteilung zu halten. Ist diese Möglichkeit nicht gegeben, können z. B. Produktionsgeheimnisse auf Rechner der Verwaltungsabteilung migrieren und vom dort zuständigen Administrator aus der Datenbank gelesen werden. Mit der Anzahl der Personen, die die Möglichkeit haben, auf die Daten unautorisiert zuzugreifen, steigt auch das Risiko, daß geheime Daten kompromittiert werden.

Um diese Problematik zu entschärfen, kann man das Sicherheitsstufenkonzept um die Verwendung von Kategorien erweitern, wie im nächsten Abschnitt ausführlich erläutert.

5.2.3 Verwendung von Kategorien

Um eine noch bessere Migrationskontrolle zu erhalten, sollte es zusätzlich zu obigem Modell möglich sein, den Objekten und Rechnern Kategorien zuzuweisen. Angelehnt ist dieses Konzept der Kategorisierung an das aus der Autorisierung bekannte *Mandatory-Access-Controll Modell* [CFMS95]. Kategorien können dabei – je nach Anwendung – beliebige Zeichenketten sein. Im Falle einer Firma wären mögliche Kategorien z. B. „Produktion“, „Personal“, „Entwicklung“ und „Administration“. Um diese Erweiterung des Sicherheitskonzeptes vornehmen zu können, muß als erstes der Begriff der Sicherheitsklasse eingeführt werden:

Definition 2 (Sicherheitsklasse) Eine Sicherheitsklasse ist ein Tupel (S, K), wobei S eine Sicherheitsstufe und K eine Menge von Kategorien repräsentiert.
Auf diesen Sicherheitsklassen kann man nun eine partielle Ordnung definieren: Seien $SK_1 = (S_1, K_1)$ und $SK_2 = (S_2, K_2)$ Sicherheitsklassen. Dann gilt:

$$SK_1 \leq SK_2 \Leftrightarrow S_1 \leq S_2 \wedge K_1 \subseteq K_2.$$

Diese Sicherheitsklassen können nun sowohl Rechnern als auch Objekten zugewiesen werden. Auch hier gilt, wie schon im obigen Fall, daß eine Migration nur dann erlaubt ist, wenn die Sicherheitsklasse S_O eines migrationswilligen Objektes O kleiner oder gleich der Sicherheitsstufe S_R des Zielrechners R ist, also der Rechner eine genügend hohe Sicherheitsstufe bietet und mindestens für die Kategorien zugelassen ist, die das Objekt fordert. Natürlich müssen auch hier die Sicherheitsklassen, die den Rechnern zugewiesen werden, signiert werden, um Manipulationen zu verhindern. Damit entspricht das zu verwendende Migrationsprotokoll dem des letzten Abschnittes.

Diese Art der Objekt- und Rechnerklassifikation erlaubt nun eine noch genauere Eingrenzung der Rechner, auf die ein Objekt migrieren darf. Der Verwaltungsaufwand steigt dabei nur in einem vertretbaren Maß. Ein Beispiel für für den Einsatz von Sicherheitsklassen zeigt Abbildung 5.2. Hier werden den Servern, die im allgemeinen nicht öffentlich zugänglich sind und mit einem sicheren Betriebssystem (z. B. Unix, Windows NT) arbeiten, sehr hohe Sicherheitsstufen zugewiesen. Rechner, die einfach zugänglich sind und unter Umständen mit einem weniger sicheren Betriebssystem arbeiten, erhalten entsprechend niedrigere Sicherheitsstufen. Die Kategorien wurden dazu verwendet, die Rechner und Objekte den verschiedenen Abteilungen einer Firma zuzuordnen.

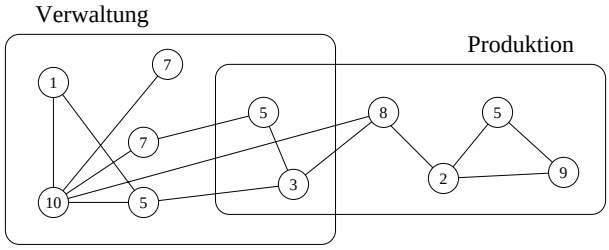


Abbildung 5.2: Beispiel für die Verwendung von Sicherheitsklassen

Auch dieses Sicherheitskonzept stößt in manchen (sehr komplexen) Anwendungsfällen an seine Grenzen; man stelle sich zwei Firmen A und B vor, die beide ihre Verwaltungsaufgaben an eine externe Firma X übertragen haben. Damit X seiner Aufgabe nachkommen kann, benötigt es einerseits von Firma A verschiedene Objekte mit der Sicherheitsklasse $(6, \{\text{Verwaltung\_Firma\_A}\})$, und andererseits benötigt es von Firma B Objekte der Sicherheitsklasse $(3, \{\text{Verwaltung\_Firma\_B}\})$. Um für beide Firmen arbeiten zu können, müßte X seine Rechner als $(6, \{\text{Verwaltung\_Firma\_A, Verwaltung\_Firma\_B}\})$ einstufen lassen und könnte sich damit unter anderem auch unautorisierten Zugriff auf Objekte der Klasse $(6, \{\text{Verwaltung\_Firma\_B}\})$ verschaffen. Diese Tatsache wäre wohl in den

meisten Fällen für Firma B nicht tolerierbar, da dadurch X Zugriff auf Daten erlangt, die in keiner Verbindung zu den Verwaltungsaufgaben stehen, die X übertragen wurden.

Das vorliegende Modell kann so erweitet werden, daß auch der eben beschriebene, wenn auch zugegebenermaßen etwas hypothetische Fall, korrekt behandelt werden kann. Diese Erweiterung wird im folgenden Abschnitt kurz vorgestellt und bewertet.

5.2.4 Verallgemeinerung des Kategorienmodells

Verallgemeinert man das Kategorienmodell aus dem letzten Kapitel, so erhält man folgende erweiterte Sicherheitsklassen:

Definition 3 (Erweiterte Sicherheitsklasse) Eine erweiterte Sicherheitsklasse ESK ist eine Menge von (Sicherheitsstufe, Kategorie)-Tupeln:

$$ESK = \{(S_1, K_1), (S_2, K_2), \ldots, (S_n, K_n)\},$$

wobei S_i jeweils die zur Kategorie K_i gehörende Sicherheitsstufe ist.

Der Vergleich erweiterter Sicherheitsklassen $ESK = \{(S_1, K_1), (S_2, K_2), \ldots, (S_n, K_n)\}$ und $ESK' = \{(S'_1, K'_1), (S'_2, K'_2), \ldots, (S'_m, K'_m)\}$ hat dann folgende Form: $ESK \leq ESK'$, wenn folgende Bedingungen gelten:

1. $\{K_1, K_2, \ldots, K_n\} \subseteq \{K'_1, K'_2, \ldots, K'_m\}$
2. $\forall k \in \{K_1, K_2, \ldots, K_n\} : S(k) \leq S'(k)$, wobei $S(k)$ und $S'(k)$ jeweils die zur Kategorie k gehörige Sicherheitsstufe in ESK bzw. ESK' angeben.

Mit diesem sehr allgemeinen Modell ist nun auch das Szenario aus dem vorhergehenden Abschnitt handhabbar, denn die Rechner der Firma X können die erweiterte Sicherheitsklasse {(6,Verwaltung_Firma_A),(3,Verwaltung_Firma_B)} erhalten. Damit hätte X bei keiner der beiden Firmen Zugriff auf Daten mit höherer Sicherheitsklasse, als unbedingt notwendig.

Nun stellt sich die Frage, ob sich der zusätzliche Aufwand lohnt, den das verallgemeinerte Kategorienmodell verursacht. Der Vorteil der Verallgemeinerung liegt klar in der größeren Flexibilität bei sehr komplexen Anwendungsszenarien, der Nachteil ist in der Unübersichtlichkeit und dem erhöhten Administrationsaufwand zu sehen.

Betrachtet man das geschilderte Szenario genauer, wird man sehr schnell feststellen, daß es nicht sehr realistisch ist. Damit die Firmen A, B und X in der geschilderten Art und Weise zusammenarbeiten können, müssen sie ein gemeinsames AutO-System verwenden. Diese Tatsache dürfte in einer realen Anwendung eine derartige Zusammenarbeit verhindern, da die einzelnen Firmen die Kontrolle über ihr Sicherheitssystem aufgeben müssen, da nur ein Sicherheitsadministrator vorgesehen ist (siehe 5.6).

Abschließend kann man sagen, daß das verallgemeinerte Kategorienmodell seine Vorteile nur in sehr komplexen Anwendungsfällen ausspielen könnte. In allen anderen Fällen fällt die aufwendigere Administration und der zusätzliche Aufwand wohl stärker ins Gewicht.

Am sinnvollsten erscheint also die Integration des normalen Kategorienmodells in AutO, das nur wenig Verwaltungsmehraufwand im Vergleich zum Sicherheitsstufenmodell hat, dafür aber flexibler einsetzbar ist.

5.3 Änderungen der Klassifikationen

Bei der Erweiterung von AutO um die Verwendung von Sicherheitsklassen müssen auch zwei Guards in die autonomen Objekte integriert werden, die es erlauben, die Sicherheitsklasse eines Objektes abzufragen oder zu ändern. Durch die Autorisierung in AutO wird verhindert, daß diese Guards von unautorisierten Personen verwendet werden. Die Sicherheitsklasse eines Rechners wird in einer Datei abgelegt und ist – wie bereits erwähnt – digital signiert, um sie vor Manipulationen zu schützen. Bei Bedarf kann die Datei, die die bisherige Sicherheitseinstufung des Rechners enthält, einfach mit einer neuen signierten Sicherheitsklasse überschrieben werden. Beim nächsten Zugriff auf diese Klassifikation, die vom Knotenmanager verwaltet wird, liest dieser die geänderte Datei ein und verwendet sie ab diesem Zeitpunkt.

5.3.1 Änderung der Klassifikation von autonomen Objekten

Ändert sich die Sicherheitsklasse eines Objektes, kann es passieren, daß die Sicherheitsklasse, die der aktuelle Rechner bietet, für das Objekt nicht mehr ausreichend ist. In diesem Fall muß sich die Person, die die Änderung veranlaßt hat, auch darum kümmern, daß das Objekt auf einen anderen Rechner migriert. Das System geht davon aus, daß ein Rechner, auf dem ein Objekt liegt, nicht plötzlich zu unsicher für dieses wird. Eine automatische Migration ist in diesem Fall nicht möglich, da im allgemeinen keine Informationen über die Sicherheitsklassen der anderen Rechner im System vorliegen. Nur der Good-LAN und der Best-LAN Strategie stehen bisher derartige Informationen über die Netzstruktur und die Sicherheitsklassen der Rechner zur Verfügung. Wären diese Informationen immer vorhanden, könnte man in dem Fall, daß einem Objekt der aktuelle Rechner zu unsicher wird, einen auswählen, der eine geeignete Sicherheitsklasse aufweist, und das Objekt zu diesem Rechner migrieren. In dem Fall, daß nirgends im System ein genügend sicherer Rechner existiert oder dem Objekt die *ImmobileStrategie* zugewiesen ist, würde allerdings auch dieser Ansatz scheitern.

5.3.2 Änderung der Klassifikation von Rechnern

Noch schlimmer wirkt sich die Änderung der Sicherheitseinstufung eines Rechners aus. Ist die neue Sicherheitsstufe kleiner als die alte, oder sind beide unvergleichbar, müßten alle Objekte, die ein Rechner beherbergt, daraufhin überprüft werden, ob ihnen der Rechner noch eine ausreichende Sicherheitsklasse bietet. Dazu müßten auch alle Objekte, die nicht im Hauptspeicher sind, sondern in der Datenbank liegen, geladen und überprüft werden. Da die Kosten, die ein solcher Schritt verursacht, nicht abzuschätzen sind, wird eine solche

Überprüfung nicht vorgenommen, sondern davon ausgegangen, daß der Rechner trotz geänderter Sicherheitsklasse weiterhin sicher genug für alle Objekte ist, die auf diesem Rechner positioniert sind. Da keine neuen Objekte auf den Rechner migrieren können, für die er keine ausreichende Sicherheit bieten kann, wird die Anzahl der nicht ordnungsgemäß positionierten Objekte mit jeder Migration eines solchen Objektes geringer.

5.4 Anpassung der Migrationsstrategien

Wird das erweiterte Sicherheitskonzept mit Kategorien verwendet, bietet es sich an, auch die Migrationsstrategien daran anzupassen. Damit wird erreicht, daß Migrationen nicht deshalb fehlschlagen, weil der anvisierte Zielrechner keine geeignete Sicherheitklasse aufweist. Dies läßt sich verhindern, indem der Zielrechner nur in der Menge der Rechner mit geeigneter Sicherheitsklasse ausgewählt wird. Für diese Anpassung eignen sich die bereits beschriebenen Strategien Good-LAN und Best-LAN besonders, weil sie ohnehin ein fundamentales Wissen über die Netzstuktur des AutO-Systems haben. Um dieses Wissen verfügbar zu machen, besitzt jeder Prozeß eine Instanz der Klasse `AutOSystemStructure`. Erweitert man diese derart, daß sie auch die Sicherheitsklassen der einzelnen Rechner speichert, können diese Informationen genutzt werden, um effizient eine gültige Migrationsentscheidung zu treffen. Dazu werden für alle LANs, Regionen und Kontinente die jeweils höchsten verfügbaren Sicherheitsklassen[4] der darin enthaltenen Rechner ermittelt. Damit ist es der Good-LAN Strategie ohne großen Aufwand möglich, die zulässigen Kontinente, Regionen und LANs zu bestimmen, wie im nächsten Abschnitt ausführlicher beschrieben.

5.4.1 Erweiterung der Good-LAN-Strategie

Bisher betrachtet ein migrationswilliges Objekt alle Kontinente, sucht sich den besten aus, betrachtet dann alle Regionen des gewählten Kontinentes und bestimmt davon die Beste. In dieser sucht sie schließlich das optimale LAN. Da den Rechnern und Objekten nun Sicherheitsklassen zugewiesen sind, kann man diese Berechnung effizienter gestalten, indem man schon bei der Auswahl der Kontinente, Regionen und LANs auf deren Sicherheitsklassen achtet und nur für diejenigen die Kosten berechnet, die eine geeignete Klassifikation bieten.

Der Pseudocode der Migrationsstrategie sieht somit folgendermaßen aus (siehe auch Abschnitt 3.2):

RechnerID berechneMigrationsZielGoodLan()
  K_{akt_lan} := berechneKommunikationsKosten(akt_lan);
  $kontinente$:= Menge der Kontinente mit geeigneter Sicherheitsklasse;
  $best\_kontinent$:= berechneBestenKontinent($kontinente$);
  $regionen$:= Menge der Regionen in $best\_kontinent$ mit geeigneter Sicherheitsklasse;
  $best\_region$:= berechneBesteRegion($regionen$);

[4] Dabei kann es sich um eine Menge handeln, da es unvergleichbare Sicherheitsklassen gibt.

$lans$:= Menge der LANs in $best\_region$ mit geeigneter Sicherheitsklasse;
$best\_lan$:= berechneBestesLAN($lans$);
if ($K_{akt\_lan} - K_{best\_lan} \geq c_{global} \times anz\_betr\_nachrichten$) **then**
  R := Alle Rechner aus $best\_lan$ mit geeigneter Sicherheitsklasse;
  return RechnerID des Rechners aus R mit den meisten Zugriffen;
endif
if ($akt\_lan == best\_lan$) **then**
  R := Alle Rechner aus akt_lan mit geeigneter Sicherheitsklasse;
  $best\_rechner$:= Rechner aus R mit den meisten Zugriffen;
  $K_{best\_rechner}$:= berechneKommunikationsKosten($best\_rechner$);
  $K_{akt\_rechner}$:= berechneKommunikationsKosten($akt\_rechner$);
  if ($K_{akt\_rechner} - K_{best\_rechner} \geq c_{lokal} \times anz\_betr\_nachrichten$) **then**
    return RechnerID von best_rechner;
  endif
endif
return RechnerID von akt_rechner;

Die Auswertung dieser Strategie liefert nun immer einen Rechner, der mindestens die geforderte Sicherheitsklasse bietet und verhindert dadurch, daß Migrationen wegen einer zu niedrigen Sicherheitseinstufung fehlschlagen. Trotzdem muß der ausgewählte Zielrechner seine aktuelle Sicherheitsklasse an den Rechner des migrationswilligen Objektes schicken, da sich diese mittlerweile geändert haben kann. Eine Änderung einer Klassifikation kann nicht ohne Verzögerungszeiten an alle Rechner im System weitergegeben werden, deshalb können kurzzeitige Inkonsistenzen zwischen den gespeicherten Informationen und den realen Verhältnissen im System auftreten.

Die Good-LAN Strategie eignet sich also sehr gut, Migrationsentscheidungen in einem System zu treffen, das mit Sicherheitsklassen arbeitet. Auch die zweite Migrationsstrategie, die schon vorgestellt worden ist, läßt sich ausgezeichnet anpassen, wie der nächste Abschnitt zeigt.

5.4.2 Erweiterung der Best-LAN-Strategie

Auch die Best-LAN-Strategie läßt sich sehr gut an die Verwendung von Sicherheitsklassen anpassen. Dazu ist es lediglich nötig, bei der Berechnung des optimalen LANs diejenigen zu ignorieren, deren Sicherheitsklassen für das migrierende Objekt nicht ausreichen. Außerdem muß bei der endgültigen Auswahl des Zielrechners auf eine geeignete Sicherheitsklasse geachtet werden. Der Pseudo-Code der entsprechend modifizierten Strategie sieht damit folgendermaßen aus (siehe auch Abschnitt 3.2):

RechnerID berechneMigrationsZielBestLan()
  K_{akt_lan} := berechneKommunikationsKosten(akt_lan);
  $best\_lan$:= K_{akt_lan};
  K_{best_lan} := K_{akt_lan};

```
for each LAN L ≠ akt_lan
   if (Sicherheitsklasse(L) ≥ Sicherheitsklasse(O)) then
      kosten := berechneKommunikationsKosten(L);
      if (kosten < K_best_lan) then
         K_best_lan := kosten;
         best_lan := L;
      endif
   endif
endfor
if (K_akt_lan − K_best_lan ≥ c_global × anz_betr_nachrichten) then
   R := Alle Rechner aus best_lan mit geeigneter Sicherheitsklasse;
   return RechnerID des Rechners aus R mit den meisten Zugriffen;
endif
if (akt_lan == best_lan) then
   R := Alle Rechner aus akt_lan mit geeigneter Sicherheitsklasse;
   best_rechner := Rechner aus R mit den meisten Zugriffen;
   K_best_rechner := berechneKommunikationsKosten(best_rechner);
   K_akt_rechner := berechneKommunikationsKosten(akt_rechner);
   if  (K_akt_rechner − K_best_rechner ≥ c_lokal × anz_betr_nachrichten) then
      return RechnerID von best_rechner;
   endif
endif
return RechnerID von akt_rechner;
```

5.5 Rechnersicherheit

Die bisherigen Bemühungen zielten darauf ab, die Objekte vor unbefugtem Zugriff zu schützen. Nun stellt sich noch die Aufgabe, die Rechner und das AutO-System selbst vor Attacken autonomer Objekte und Transaktionen zu schützen. Erweitert man das Sicherheitskonzept in geeigneter Weise, kann man verhindern, daß autonome Objekte die virtuelle Maschine beenden, Dateien lesen, speichern und löschen, Kommunikationsverbindungen aufbauen oder andere Aktionen auslösen, die sicherheitsspezifische Probleme nach sich ziehen können. Viele dieser Aktionen müssen schon aus der Sicht von AutO unterbunden werden, weil durch sie unter Umständen die ACID-Eigenschaften des Systems verletzt werden. Um derartige unerlaubte Handlungen zu verhindern, ist es nötig, einen spezialisierten SecurityManager zu implementieren und eine Sicherheitsstrategie zu spezifizieren, die illegale Methodenaufrufe autonomer Objekte abfängt.

Außerdem ist es benutzerdefinierten autonomen Objekten und Transaktionen möglich, direkt auf Objekte des AutO-Systems zuzugreifen und diese dadurch zu manipulieren. Um dies zu verhindern, wird ein ClassLoader entwickelt, der die systeminternen Klassen von AutO vor benutzerdefinierten Objekten verbirgt.

5.5.1 Der SecurityManager

Das Sicherheitskonzept von Java, das bei Problemen dieser Art eingesetzt werden kann, ist mehrstufig aufgebaut (siehe Abschnitt 5.1). Beim Laden einer Klasse wird der Code auf Korrektheit geprüft. Dabei ist auch der ClassLoader aktiv, der den Code in die JVM lädt und erste anwendungsspezifische Sicherheitsprüfungen vornehmen kann. Darunter fällt z. B. die Prüfung, in welche *Packages*[5] Klassen geladen werden dürfen. Die letzte vom Programmierer beeinflußbare Sicherheitsinstanz bildet die Klasse `SecurityManager`. Diese gestattet es, folgende Aktionen im Bedarfsfall zu unterbinden:

- Aufbau und Verwendung von Socket-Verbindungen
- Änderung von Threadparametern und Threadgruppen
- Verwendung der AWT-Queue und Generierung von Top-Level Fenstern
- Verändern des ClassLoaders von Java
- Lesen, Schreiben und Löschen von Dateien
- Ausführen von Subprozessen (via „exec“)
- Beenden der virtuellen Maschine
- Laden dynamischer Laufzeitbibliotheken
- Generierung von Druckjobs
- Zugriff auf die Systemeigenschaften des Rechners
- Verwendung der Zwischenablage des Betriebssystems

Alle diese Aktionen dürfen normalerweise von autonomen Objekten nicht ausgeführt werden, da sie entweder ACID verletzen (z. B. Druckjobs), das System gefährden (z. B. Beenden der virtuellen Maschine) oder mit der Migration nicht verträglich sind (z. B. Generierung von Fenstern).

In Java 1.2 trifft der SecurityManager seine Entscheidungen, ob eine Aktion ausgeführt werden darf oder nicht, unter Verwendung der Klasse `java.security.AccessController`. Diese fällt ihr Urteil basierend auf einer konfgurierbaren Sicherheitspolitik. Dabei können Rechte an Klassen in Abhängigkeit des Herkunftsortes[6] und/oder der Signatur einer Klasse vergeben werden. Bei der Rechtevergabe können die AutO-Systemklassen als vertrauenswürdig eingestuft werden und unterliegen deshalb keinerlei Restriktionen. Den Klassen aus dem Verzeichnis, in dem benutzerdefinierte Klassen gespeichert sind, werden normalerweise keine speziellen Rechte zugewiesen. Dadurch werden alle eben erwähnten Aktionen, die durch den SecurityManager kontrolliert werden, verhindert.

[5] Java bietet die Möglichkeit, Klassen in Packages zu strukturieren. Dabei können Klassen innerhalb eines Packages erweiterte Zugriffsrechte auf Variablen, Methoden und Klassen dieses Packages erhalten.

[6] angegeben als URL

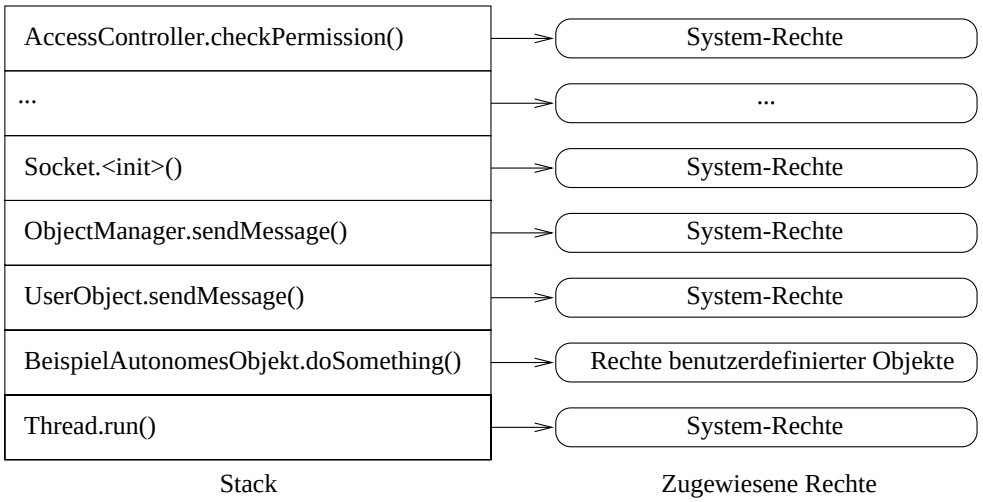


Abbildung 5.3: Beispielstack eines autonomen Objekts

Um einen reibungslosen Betrieb von AutO garantieren zu können, muß die Arbeitsweise des AccessControllers genauer betrachtet werden. Dieser erlaubt normalerweise nur dann eine Aktion, wenn alle Klassen, die auf dem Stack[7] liegen, das Recht haben, diese auszuführen. Einen Beispielstack zeigt Abbildung 5.3. Aus diesem Grund muß man Aktionen im AutO-System, die von benutzerdefinierten Objekten angestoßen werden können, aber für diese nicht erlaubt sind, speziell behandeln. Der AccessController kontrolliert alle Klassen auf dem Stack von oben nach unten, endet aber bei der ersten Klasse, die ihre gerade auf dem Stack liegende Methode als *privilegierten Code* markiert hat. Nötig ist das z. B. bei Netzwerk-Verbindungen, die AutO nur im Bedarfsfall herstellt. Greift ein benutzerdefiniertes Objekt auf die Methode einer Systemkomponente (im Beispiel ist das die Objektverwaltung `ObjectManager`) von AutO zu, die einen dieser „On-Demand-Sockets“ verwendet, muß von dieser u. U. zuerst eine Verbindung aufgebaut werden. Da das benutzerdefinierte Objekt als Aufrufer der Methode natürlich auf dem Stack liegt, würde der AccessController dies verhindern. Wird das Öffnen des Sockets innerhalb der AutO-Systemkomponente als privilegierter Code gekennzeichnet, wird diese Aktion vom AccessController gestattet. Diese Möglichkeit, Privilegien des AutO-Systems für spezielle Aktionen an benutzerdefinierte Objekte „auszuleihen“, ermöglicht erst den reibungslosen Betrieb von AutO.

Um die Sicherheit der AutO-Rechner zu gewährleisten, sollte die – zugegebenermaßen sehr restriktive – Sicherheitspolitik, den benutzerdefinierten Objekten keinerlei Rechte zu geben, verwendet werden. Sollte es nötig sein, eine der oben aufgezählten Aktionen für eine benutzerdefinierte Klasse zu erlauben, kann dies in die Konfigurationsdatei (AutO.policy) eingetragen werden (was natürlich nur dem AutO-Sicherheitsadministrator erlaubt sein darf). In diesem Fall ist es nötig, darauf zu achten, daß der Betrieb von AutO trotz Migration noch reibungslos funktioniert. Dies kann beispielsweise garantiert werden, wenn

[7] Der Stack spiegelt Methodenaufrufe eines Threads in ihrer Reihenfolge wieder. Dabei liegen stets nur Methodenaufrufe auf dem Stack, die noch nicht vollständig abgearbeitet worden sind.

man sich an einem der drei folgenden Punkte orientiert:

- Die Objekte dieser privilegierten Klasse sind stationär. Dadurch steht ihnen ihr privilegiertes Recht immer zur Verfügung.
- Alle Rechner des AutO-Systems verwenden dieselbe Sicherheitsstrategie.
- Die Objekte können mit der Situation, daß sie auf anderen Rechnern nicht privilegierte Rechte haben, korrekt umgehen und bleiben dabei voll arbeitsfähig.

5.5.2 Der ClassLoader

Der SecurityManager schützt einen Rechner, der autonome Objekte und Transaktionen beheimatet, davor, daß diese unkontrollierten Zugriff auf die Ressourcen des Rechners haben. Zusätzlich ist es aber nötig, das AutO-System selbst vor Manipulationen oder anderen Angriffen durch autonome Objekte zu schützen. Um dies zu erreichen, wird ein ClassLoader implementiert, der Zugriffe benutzerdefinierter Objekte auf Klassen des AutO-Systems überwacht. Dies ist möglich, da jeder ClassLoader in Java seinen eigenen Namensraum definiert, und damit die Klassen, die dem AutO-System zur Verfügung stehen, und die, die benutzerdefinierte Objekte verwenden können, vollkommen unabhängig sind. Eine weitere Aufgabe dieses ClassLoaders (im folgenden *AutOClassLoader* genannt) ist es, benutzerdefinierte Klassen, die nicht lokal vorhanden sind, durch den Informationservice zu lokalisieren und in das lokale Dateisystem zu kopieren. Dies kann nötig sein, da bei der Migration eines Objektes nur die Klasse des Objektes selbst, deren Basisklassen und die Klassen der Parameter der Methoden des Objektes auf den Zielrechner kopiert werden. Weitere Klassen, die eventuell von diesem Objekt benötigt werden, stellt der AutOClassLoader dem System vollkommen transparent zur Verfügung.

In jedem AutO-Prozeß gibt es, wie bereits erläutert, verschiedene Verwaltungs- und Dienstleistungsobjekte, wie z. B. den `ObjectManager` oder den `CommunicationAgent`. Verhindert man den unkontrollierten Zugriff auf diese Objekte bzw. deren Klassen nicht, können sich benutzerdefinierte Objekte eigene Instanzen dieser Klassen erzeugen oder die Daten existierender Instanzen manipulieren. Ein weiteres Problem des ungehinderten Zugriffs auf die Methoden dieser Dienstleistungsobjekte ist, daß sie mit falschen Parametern aufgerufen werden können und es damit möglich ist, z. B. eine Nachricht mit einer gefälschten Absenderadresse zu versenden. Um dies zu verhindern, stehen sowohl für Transaktionen als auch für autonome Objekte Basisklassen (`Transaction` bzw. `UserObject`) zur Verfügung, die alle notwendigen Zugriffe auf Systemkomponenten von AutO implementieren. Benutzerdefinierten Objekten und Transaktionen ist es nur erlaubt, über die Methoden dieser Basisklassen mit dem AutO-System zu kommunizieren. Der AutOClassLoader überprüft die Einhaltung dieser Forderung, indem er dazu verwendet wird, die benutzerdefinierten Objekte zu laden. Jede weitere Klasse, die von einem solchen Objekt geladen wird – entweder explizit durch den Aufruf einer Methode des ClassLoaders oder implizit durch Java veranlaßt (z. B. Basisklassen, Klassen von Membervariablen oder von Parametern) – lädt Java ebenfalls über diesen AutOClassLoader, der dadurch die Möglichkeit hat, das Laden von Klassen zu verweigern, auf die nicht zugegriffen werden darf.

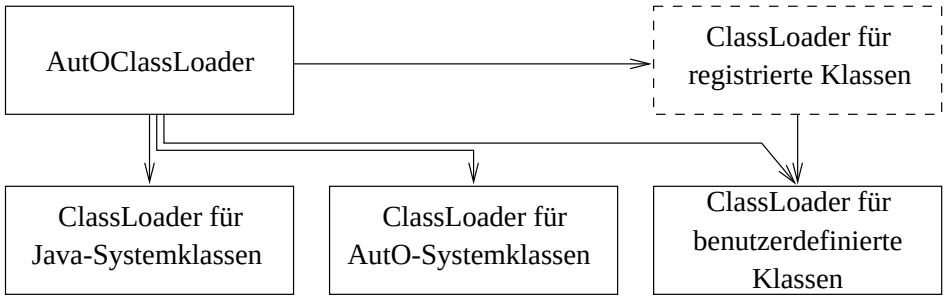


Abbildung 5.4: Arbeitsweise des AutOClassLoaders

Um diese Aufgabe bewältigen zu können, besteht der AutOClassLoader aus mehreren verschiedenen ClassLoadern, an die das Laden der Klasse nacheinander delegiert wird, bis die angeforderte Klasse entweder gefunden wurde, oder bis feststeht, daß die Klasse nicht verwendet werden darf oder nicht existiert. Der Aufbau des AutOClassLoaders wird in Abbildung 5.4 verdeutlicht.

Beim Laden einer Klasse durch eine Transaktion oder ein benutzerdefiniertes Objekt wird dabei nach folgendem Schema vorgegangen:

- Zuerst wird versucht, die Klasse innerhalb der Java-Systemklassen zu lokalisieren. Da diese Klassen bei allen eventuell gefährlichen Aktionen den SecurityManager befragen, stellen sie kein Sicherheitsrisiko dar und dürfen von benutzerdefinierten Objekten geladen werden. Der AutOClassLoader verwendet zum Laden der Klasse einen URLClassLoader, der die geforderte Klasse in den Verzeichnissen sucht, die die Java-Systemklassen beinhalten.

- Handelt es sich nicht um eine Systemklasse von Java, wird überprüft, ob die Klasse in einer Liste enthalten ist, die alle diejenigen Klassen des AutO-Systems beinhaltet, die von autonomen Objekten geladen werden dürfen. Die wichtigsten Klassen sind dabei `UserObject` und `Transaction`, die die Basisklassen aller autonomen Objekte bzw. Transaktionen in AutO bilden. Diese spezifizieren alle Methoden, die nötig sind, um mit AutO zu kommunizieren. Gehört die angeforderte Klasse zu dieser Art von AutO-Klassen, wird sie von dem ClassLoader geladen, der die Klassen des AutO-Systems geladen hat.

- Ist die gesuchte Klasse auch keine der zugänglichen AutO-Klassen, wird als nächstes überprüft, ob das Package, in das die Klasse geladen werden soll, bereits von anderen Klassen im CLASSPATH belegt ist. Sollte das der Fall sein, wird das Laden der Klasse verweigert. Diese Sicherheitsmaßnahme ist nötig, um zu verhindern, daß ein benutzerdefiniertes Objekt z.B. in eines der `java.*` oder der Packages von AutO eingeschleust wird und sich dadurch privilegierte Zugriffsrechte auf andere Klassen in diesem Package erschleicht.

- Anschließend wird versucht, die Klasse in dem Verzeichnis zu lokalisieren, in dem die benutzerdefinierten Objekte abgelegt sind. Wird die Klasse dort gefunden, wird

sie vom AutOClassLoader selbst geladen.

Ist sie dort nicht zu finden, wird bei dem zentralen Informationserver nachgefragt, ob die gesuchte Klasse bei ihm registriert ist. Ist das der Fall, wird die Adresse eines Rechners geliefert, der die Klasse besitzt. Von diesem wird sie angefordert, in dem Verzeichnis für benutzerdefinierte Klassen installiert und dann vom AutO-ClassLoader geladen.

- Schlug auch dieser letzte Versuch fehl, die Klasse zu laden, ist sie weder auf dem lokalen Rechner vorhanden noch beim Informationserver registriert. Das Laden der Klasse scheitert damit.

Durch das eben beschriebene Vorgehen beim Laden einer Klasse erreicht man, daß keine AutO-internen Klassen unautorisiert von autonomen Objekten verwendet werden können. Versucht ein autonomes Objekt nämlich, z. B. den ObjectManager zu manipulieren, müßte zuerst dessen Klasse geladen werden. Diese gehört nicht zu den oben erklärten „erlaubten Klassen", damit scheitert dieser Zugriff. Die Klassen `Transaction` und `UserObject` dienen als Brücke zwischen dem Namensraum des AutO-Systems und dem der benutzerdefinierten Objekte. Da der AutOClassLoader den normalen ClassLoader mit dem Laden der UserObject-Klasse beauftragt hat, wurde die Klasse nur einmal geladen, ist aber in beiden Namensräumen bekannt. Ruft beispielsweise ein autonomes Objekt nun die *sendMessage*-Methode von UserObject auf, kann dieses auf den ObjectManager zugreifen, weil dieser dem normalen ClassLoader (der ja auch UserObject geladen hat) bekannt ist. Versucht das autonome Objekt selbst, auf den ObjectManager ohne den Umweg über die Basisklasse zuzugreifen, wird es aus den im letzten Absatz angeführten Gründen scheitern. Auch der Zugriff auf weitere Basisklassen von Transaction bzw. UserObject ist für benutzerdefinierte Objekte nicht möglich, wie in Abbildung 5.5 verdeutlicht.

Die Transaktion selbst wird durch den AutOClassLoader geladen. Die JVM von Java versucht daraufhin die Basisklassen der eben geladenen Klasse nachzuladen und wendet sich mit dem Auftrag, die Klasse `Transaction` zu laden, an den AutOClassLoader. Dieser stellt fest, daß dies eine für benutzerdefinierte Transaktionen erlaubte Klasse ist und delegiert den Ladevorgang an den normalen ClassLoader. Alle weiteren Basisklassen, die in der realen Implementation der Transaktion in AutO existieren, werden dadurch auch durch den normalen ClassLoader geladen und sind damit dem AutOClassLoader unbekannt. Dadurch erreicht man, daß benutzerdefinierte Objekte wirklich nur die ihnen zur Verfügung gestellten Methoden in ihren Basisklassen verwenden können, um mit AutO zu kommunizieren.

5.5.3 Signierte Klassen

Die bisher beschriebenen Erweiterungen des Systems stellen eine deutliche Verbesserung dar, können aber nicht alle Angriffe auf die Rechner verhindern. Weiterhin problematisch für das System sind sogenannte *denial-of-service*-Attacken, die auch aus anderen Bereichen der Informatik bekannt sind und die mittlerweile unter anderem durch Sicherheitslücken in den TCP/IP-Stacks verschiedener Betriebssysteme traurige Berühmtheit erlangt haben.

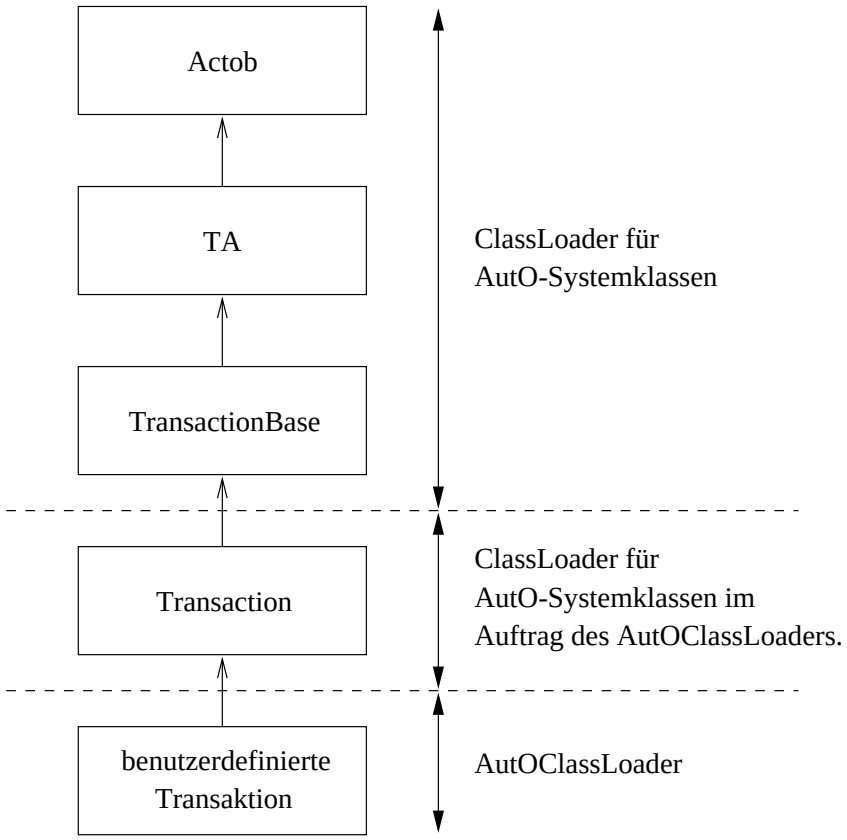


Abbildung 5.5: Aufgabenverteilung beim Laden einer Transaktion

Diese Attacken haben dabei nicht das Ziel, Daten unbefugt zu lesen oder zu modifizieren, sondern sie versuchen, das System in einen nicht funktionsfähigen Zustand zu versetzen. In einem System, das nur die bisher beschriebenen Sicherheitsaspekte berücksichtigt, wären beispielsweise folgende Attacken denkbar und ohne Probleme zu implementieren:

- Ein Objekt kann begrenzte Ressourcen anfordern und nicht wieder freigeben. Am einfachsten wäre hier die Anforderung von mehr und mehr Speicher, was dazu führen würde, daß das System nicht mehr ordnungsgemäß arbeiten kann.
- Die Implementierung eines if-Guards, der ständig aktiv ist, verhindert, daß das System dem Objekt den zugewiesenen Thread wieder entziehen kann. Dies zieht zum einen nach sich, daß neue Threads generiert werden müssen und andererseits wird, wenn dies bei mehreren Objekten der Fall ist, sehr viel Rechenzeit damit vergeudet, diese Guards abzuarbeiten. Durch derartige Objekte kann auch die Persistenz in Mitleidenschaft gezogen werden, denn das Datenbank-Log dieser Objekte wächst stetig an. Sind Objekte nicht ständig aktiv, können die nicht mehr benötigten Teile des Logs gelöscht werden.

Diese Problematik verdeutlicht, daß es nicht ausreichend ist, ein bereits instanziiertes Objekt zu überwachen. Da man mit dem SecurityManager von Java diese Sicherheitspro-

bleme nicht beheben kann, müssen alternative Lösungen gesucht werden, die denial-of-service-Attacken verhindern. Da über den Speicher- und Rechenzeitbedarf eines autonomen Objektes keine pauschalen Angaben möglich sind, können diese Informationen nicht als Kriterium dazu verwenden werden, derartige Attacken zu erkennen.

Diese können nur zuverlässig verhindern werden, indem der Code einer Klasse vor der Ausführung von einem verantwortlichen AutO-Sicherheitbeauftragten (siehe Abschnitt 5.6) geprüft wird. Nach erfolgreicher Prüfung wird diese durch eine Signatur der Klassendatei bestätigt. Das AutO-System ist damit in der Lage, beim Laden einer Klasse festzustellen, daß sie signiert wurde, und daß deshalb gefahrlos Instanzen dieser Klasse erzeugt werden können.

Da der Zeitaufwand für die Prüfung des Codes einer Klasse sehr hoch ist, kann diese Lösung nicht direkt eingesetzt werden. Ein weiterer Punkt, der gegen diesen Ansatz spricht, ist, daß bei der Überprüfung Fehler begangen werden können. Deshalb wird der Signatur einer Klasse der verantwortliche Programmierer hinzugefügt. Dadurch ist es im Falle einer Attacke möglich, den Verursacher festzustellen und diesen zur Verantwortung zu ziehen.

5.6 Erzeugung und Überprüfung der Signaturen

Bisher wurde lediglich die Notwendigkeit von Signaturen erläutert, um die Authentizität und Integrität sowohl der Sicherheitsklassen der Rechner als auch der Klassen autonomer Objekte zu gewährleisten. Dabei wurde noch nicht darauf eingegangen, wie man diese Signaturen erzeugen und verwenden kann. Eine Möglichkeit, einen derartigen Signaturdienst zu strukturieren, wird nun vorgestellt.

5.6.1 Die Struktur des Signaturdienstes

Um die angestrebte Funktionalität zu erreichen, erscheint ein zentraler Signaturserver als geeignet und ausreichend. Ein verteilter Ansatz ist hierbei nicht notwendig, da im laufenden Betrieb nur wenige Anfragen an diesen Server zu erwarten sind. Die gewählte Struktur für den Signaturdienst zeigt Abbildung 5.6. In den weiteren Abschnitten wird genauer auf die einzelnen Komponenten eingegangen.

Der Signaturserver generiert auf Anfrage der Signaturclients Signaturen für Sicherheitsklassen bzw. Java-Klassen. Dazu wird von den zu signierenden Daten ein digitaler Fingerabdruck berechnet. Dieser wird dann mit dem geheimen Schlüssel eines Schlüsselpaares (Signaturschlüsselpaar) unter Verwendung des RSA-Algorithmus [Sch96, Oak98] signiert. Der dazugehörige öffentliche Schlüssel ist jedem Rechner im System bekannt und kann dazu verwendet werden, die Gültigkeit der Signatur zu verifizieren.

Der Sicherheitsadministrator kann Sicherheitsbeauftragte benennen, die autorisiert sind, Rechner zu klassifizieren und Klassen autonomer Objekte zu prüfen. Die Signaturclients dienen zur sicheren Kommunikation der Sicherheitsbeauftragten mit dem Signaturserver. Um feststellen zu können, ob Anfragen an den Server von einem Sicherheitsbeauftragten kommen, generiert der Sicherheitsadministrator des AutO-Systems für jeden Sicherheits-

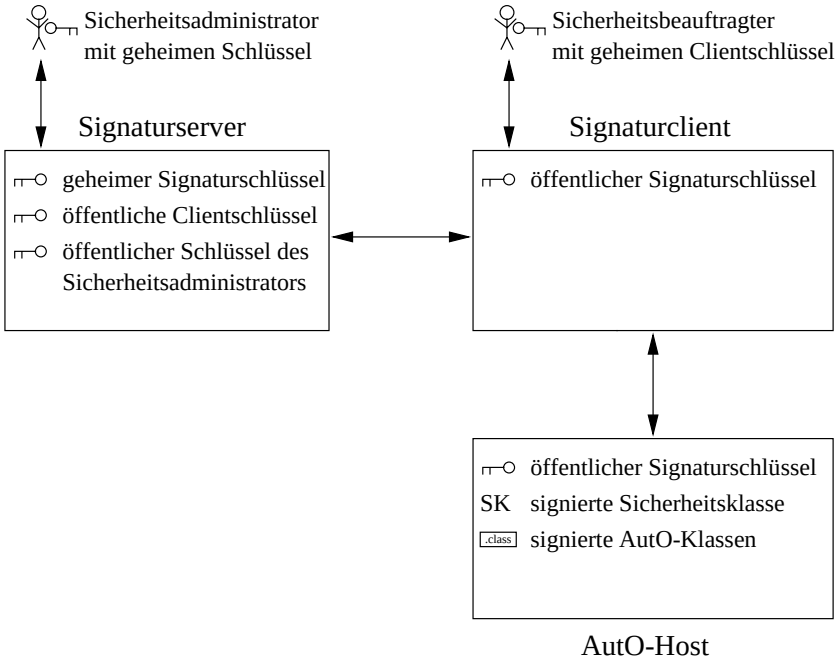


Abbildung 5.6: Struktur des Signaturdienstes

beauftragten ein eigenes Schlüsselpaar (Clientschlüsselpaar). Die öffentlichen Schlüssel werden dabei im Signaturserver gespeichert, die geheimen Schlüssel sind nur den jeweiligen Sicherheitsbeauftragten bekannt. Dadurch ist es möglich, die Anfragen an den Signaturserver zu signieren. Dieser kann dadurch eindeutig die Herkunft feststellen und unautorisierte Anfragen zurückweisen.

Ein weiteres Schlüsselpaar ist für die Kommunikation zwischen Sicherheitsadministrator und Signaturserver nötig. Dadurch wird sichergestellt, daß nur der Sicherheitsadministrator Verwaltungarbeiten vornehmen kann.

5.6.2 Der Signaturserver

Der Signaturserver dient dazu, Signaturen zu erstellen und die öffentlichen Schlüssel der Sicherheitsbeauftragten zu verwalten. Dazu wird ein systemweit gültiges Schlüsselpaar generiert, dessen öffentlicher Schlüssel allen Rechnern im System bekannt sein muß. Um diese Aufgabe bewältigen zu können, muß der Signaturserver folgende Schnittstellen anbieten:

- **Administrationsschnittstelle:** Diese Schnittstelle ist dafür bestimmt, den Server zu administrieren. Dazu müssen folgende Aufgaben unterstützt werden:
  - Hinzufügen/Entfernen von Sicherheitsbeauftragten
  - Ändern des geheimen Signaturschlüssels
  - Ändern des Schlüsselpaares des Sicherheitsadministrators

Damit diese Funktionalität auch ausschließlich vom zuständigen AutO-Sicherheitsadministrator genutzt werden kann, muß dieser alle Nachrichten an den Signaturserver mit seinem geheimen Schlüssel signieren. Die Funktionalität der Administrationsschnittstelle wird vom *SATool* genutzt, das im Anhang beschrieben wird.

- **Clientschnittstelle:** Hier nimmt der Server Aufträge von Signaturclients entgegen. Um sicherzustellen, daß diese von autorisierten Sicherheitsbeauftragten stammen, müssen auch diese Aufträge signiert werden. Auf diesem Wege kann ein Sicherheitsbeauftragter die Erzeugung einer Sicherheitsklasse für einen Rechner oder die Signatur einer Klassendatei veranlassen. Die Funktionalität der Clientschnittstelle wird vom *SRTool* abgedeckt, das ebenfalls im Anhang beschrieben wird.

5.6.3 Der Signaturclient

Der Signaturclient bildet das Bindeglied zwischen den Sicherheitsbeauftragten und dem Signaturserver. Startet ein Sicherheitsbeauftragter einen Client mit seinem geheimen Schlüssel, kann er die Dienste des Servers in Anspruch nehmen. Um einem Rechner eine Sicherheitsklasse zuzuweisen, muß dies dem Server mitgeteilt werden. Der Server kann dann eine entsprechende Sicherheitsklasse generieren und signiert zurückschicken, damit sie auf dem entsprechenden Rechner installiert werden kann.

Da Klassendateien relativ groß sein können und auch nur verschlüsselt übertragen werden sollten, wird hier ein etwas anderer Weg gewählt. Es werden nicht die kompletten Klassendateien zum Server gesendet, sondern nur die digitalen Fingerabdrücke der Klassen, die durch eine one-way Hashfunktion berechnet werden. Aus diesen Fingerabdrücken kann nicht mehr auf den Inhalt der Klassendatei zurückgeschlossen werden. Außerdem benötigen sie viel weniger Speicherplatz. Diese Fingerabdrücke werden vom Signaturserver signiert und an den Client zurückschickt. Damit besteht die Möglichkeit, zu prüfen, ob von einer Klasse Instanzen generiert werden dürfen. Dazu wird beim Laden der Klasse der Fingerabdruck berechnet und verifiziert, ob dieser mit dem signierten Fingerabdruck übereinstimmt. Ist das der Fall, kann die Klassendatei als sicher gelten. Migriert ein Objekt auf einen anderen Rechner, auf dem die Klassendatei noch nicht vorhanden ist, muß außer der Klassendatei natürlich auch die Signatur der Klasse mitgeschickt werden, damit das migrierende Objekt auf dem Zielrechner wieder instanziiert werden kann.

5.7 Aufbau der signierten Strukturen

Um zu gewährleisten, daß z.B. eine signierte Sicherheitsklasse nicht auch von anderen Rechnern verwendet werden kann, muß man gewisse Anforderungen an die zu signierenden Daten stellen. In den nächsten beiden Abschnitten wird gezeigt, wie die Signaturen der Klassendateien und der Sicherheitsklassen der Rechner aussehen müssen, um größtmöglichen Schutz zu bieten.

5.7.1 Aufbau der Sicherheitsklassen

Um zu gewährleisten, daß eine Sicherheitsklasse eindeutig einem Rechner zuzuordnen ist, muß die IP-Adresse des Rechners in der Sicherheitsklasse enthalten sein. Um später noch nachvollziehen zu können, wer die Sicherheitseinstufung eines Rechners vorgenommen hat, ist es auch sinnvoll, den Namen des Sicherheitsbeauftragten zu speichern. Damit beinhaltet eine Sicherheitsklasse folgende Daten:

- Sicherheitsstufe des Rechners
- Menge von Kategorien
- IP-Adresse des Rechners, dem die Sicherheitsklasse zugeordnet ist
- Name des Sicherheitsbeauftragten, der die Klassifizierung vorgenommen hat

Diese Daten werden mit dem geheimen Signaturschlüssel des Signaturservers verschlüsselt. Dadurch kann die Sicherheitsklasse gefahrlos über das Netz geschickt werden und die Authentizität der Klasse kann mit dem öffentlichen Signaturschlüssel überprüft werden.

5.7.2 Aufbau der Signatur von Klassen

Bei einer signierten Klasse ist es von Interesse, wer diese programmiert hat und von wem die Signatur veranlaßt worden ist. Aus den schon oben erläuterten Gründen wird nicht die gesamte Klassendatei signiert sondern nur ein digitaler Fingerabdruck davon. Darum wird eine eigene Klasse `ClassSignature` erzeugt, die folgende Daten enthält:

- Fingerabdruck der Klassendatei
- Name des Sicherheitsbeauftragten, der die Signierung veranlaßt hat
- Name des Programmierers der Klasse

Beim Laden einer Klasse kann der digitale Fingerabdruck der Klassendatei und der zusätzlichen Informationen der ClassSignature berechnet werden. Dieser wird mit dem Fingerabdruck verglichen, der in der ClassSignature enthalten ist. Stimmen beide überein, dürfen Instanzen der Klasse erstellt werden. Stimmen sie nicht überein, wurde entweder die Klassendatei modifiziert oder die Signatur ist aus anderen Gründen nicht gültig. In diesen Fällen wird das Laden der Klasse abgebrochen.

Dieser Ansatz einer seperaten Datei, die die Signatur der zugehörigen Klassendatei enthält, wurde der in Java standardmäßig zur Verfügung stehenden Möglichkeit, JAR-Dateien zu verwenden, vorgezogen, da es mit den Mitteln, die Java zur Verfügung stellt, nicht möglich ist, mit vertretbarem Aufwand neue signierte Dateien zu einer JAR-Datei hinzuzufügen. Hierzu wäre es nötig, eine neue JAR-Datei zu erstellen, die die Klassen des alten JAR-Archivs und die neu hinzukommenden Klassen enthält. Dies stellt einen großen Aufwand dar, wenn die Anzahl der im System installierten Objekttypen wächst.

Kapitel 6

Eine Beispielanwendung für AutO

Dieser Teil der Diplomarbeit beschreibt eine Anwendung für das AutO-System, die zusammen mit Markus Keidl [Kei99] entwickelt wurde, um die Arbeitsweise und Funktionsfähigkeit von AutO demonstrieren zu können. Da eine Datenbankanwendung sehr wenig Einblick in ihre Arbeitsabläufe gewährt, wurde zusätzlich eine Komponente implementiert, die es erlaubt, diese Abläufe zu visualisieren.

6.1 Die Anwendung

Die hier vorgestellte Anwendung modelliert ein sehr einfach gehaltenes Support-Center, das aus Filialen, Teams und Angestellten besteht. Ein Beispiel eines solchen Unternehmens zeigt Abbildung 6.1, dargestellt mit dem eigens für diese Anwendung entwickelten Visualisierungstool.

Das Fenster ist dabei in mehrere horizontale Abschnitte aufgeteilt, die verschiedene Rechner repräsentieren. In der gezeigten Abbildung wurde die Firmenhierarchie auf zwei Rechner verteilt. Der Rechner, der durch den oberen Teil des Fensters repräsentiert wird, beherbergt die Passauer Filiale des Unternehmens. Diese besitzt zwei Teams: MSOffice_Passau und AutO_Passau, die jeweils zwei Angestellte beschäftigen. Der andere an der Demonstration beteiligte Rechner beherbergt die zwei Filialen Landshut und Regensburg, die jeweils zwei Teams haben. Je eines dieser Teams beschäftigt dabei zur Zeit keine Arbeitnehmer und kann deshalb auch keine Aufträge annehmen.

Filialen, Teams und Angestellte werden in AutO als autonome Objekte modelliert. Um die Struktur des Unternehmens aufzubauen, werden zuerst alle nötigen autonomen Objekte generiert und initialisiert. Nach diesem ersten Schritt wird jedem Angestellten im System eine Nachricht gesendet, die die OID des Teams enthält, dem er zugewiesen ist. Nach Erhalt dieser Nachricht registrieren sich die Objekte, die die Angestellten repräsentieren, bei den jeweiligen Teams. Jedem Team wird eine Menge von Schlüsselwörtern zugewiesen, die später dazu benutzt werden, Kundenaufträge im System zu verteilen. Außerdem wird jedes Team einer Filiale zugeteilt. Nach Abschluß dieser Initialisierungsphase ist das System einsatzbereit.

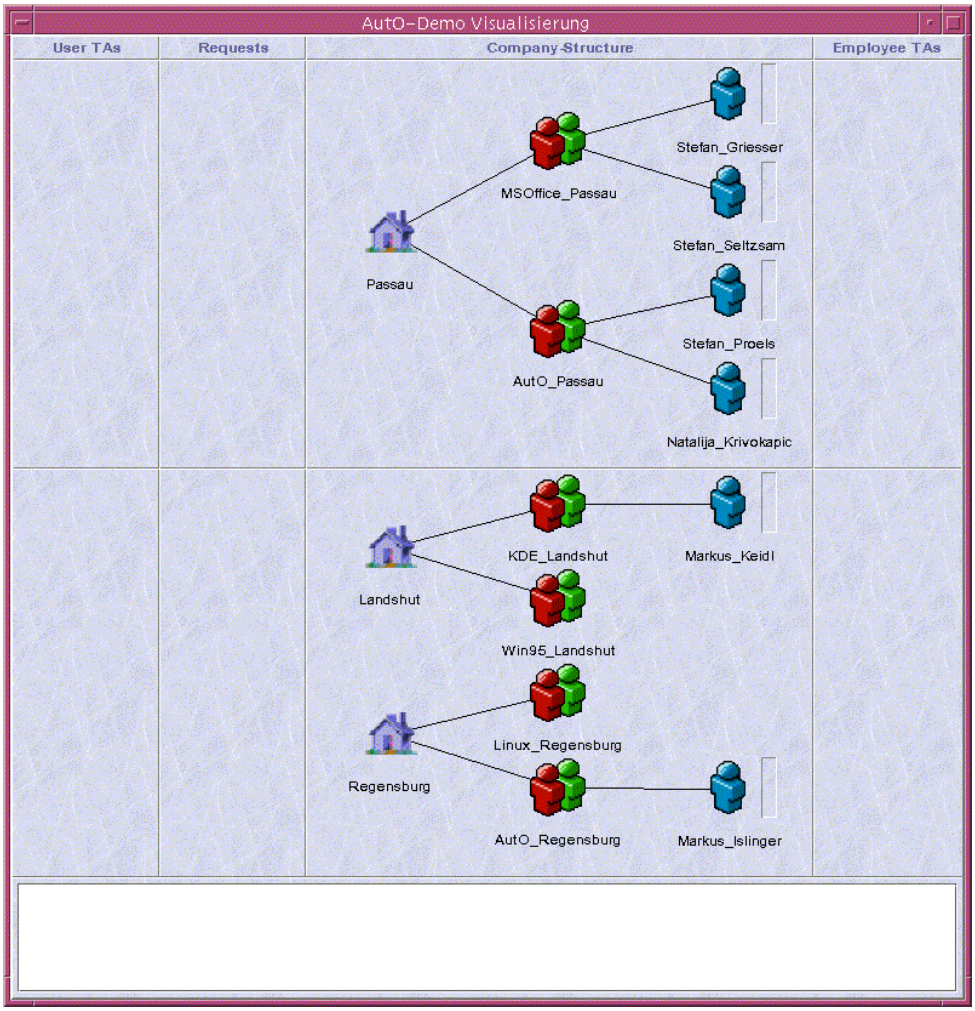


Abbildung 6.1: Beispiel-Struktur eines Support-Centers

Ein weiterer Objekttyp ist der Auftrag, der von Kunden generiert wird. Ein solcher Auftrag enthält eine Priorität, den Namen des Kunden, eine Menge von Schlüsselwörtern und die genaue Problembeschreibung. Außerdem definiert das aktive Verhalten dieses Objekttyps die Strategie, nach der sich Aufträge im System verteilen. Nach der Generierung wird ihm eine Nachricht zugesendet, die ihn veranlaßt, sich selbst mit Hilfe des Informationservice die nächstgelegene Filiale zu suchen und sich dort zu registrieren. Diese ist die „Heimat-Filiale“ des Auftrages und speichert immer den aktuellen Status des Objektes. Außerdem wird dem Auftrag von seiner Heimat-Filiale eine eindeutige Auftragsnummer zugewiesen, die aus einer Seriennummer und dem Namen der Heimat-Filiale besteht.

Das Vorgehen bei der Verteilung von Aufträgen im System ist in Abbildung 6.2 dargestellt. Dabei behält immer das Auftragsobjekt die Kontrolle über den Vorgang. Nach der Registrierung bei einer Filiale sendet der Auftrag eine Nachricht an diese, um das Team zu erfragen, zu dem der Auftrag am besten paßt, das also am meisten Übereinstimmungen der Team-Schlüsselwörter mit denen des Auftrages aufweist. Das Ergebnis

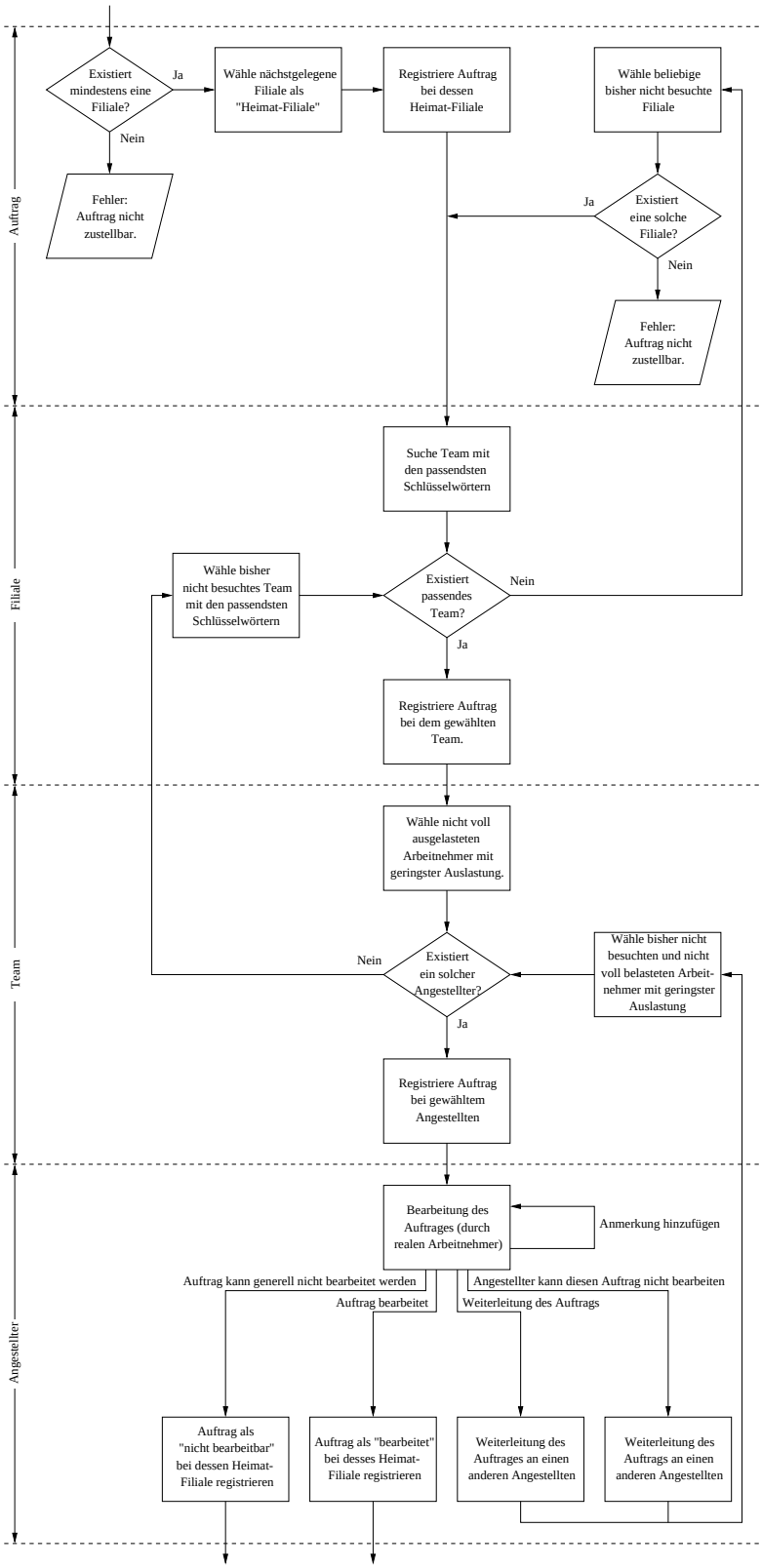


Abbildung 6.2: Zustellung eines Auftrages

dieser Team-Auswahl sendet die Filiale als Antwort zurück. Der Auftrag seinerseits kann bei diesem Team nachfragen, welchem Angestellten der Auftrag zugewiesen werden soll. Dabei wird stets der Arbeitnehmer gewählt, der zur Zeit am wenigsten Aufträge hat, wobei kein Arbeitnehmer mit mehr als 5 Aufträgen gleichzeitig belastet wird. Bei dem gewählten Angestellten registriert sich dann das Auftragsobjekt zur Bearbeitung. Tritt innerhalb dieses Ablaufs ein Fehler auf, weil z. B. alle Arbeitnehmer eines Teams voll ausgelastet sind, wird versucht, eine Alternative innerhalb des Unternehmens zu finden, wie in Abbildung 6.2 dargestellt.

6.2 Anbindung der Visualisierungskomponente

Das Visualisierungstool ist eine Anwendung, die vollkommen unabhängig vom AutO-System läuft. Da AutO keine Möglichkeiten vorsieht, Daten aus dem System nach außen weiterzuleiten, muß eine Möglichkeit gefunden werden, dem Visualisierungstool die Informationen zukommen zu lassen, die nötig sind, um die Abläufe im Support-Center darzustellen.

Die hier gewählte Lösung basiert auf einer Client/Server-Architektur. Die Visualisierungskomponente übernimmt die Aufgabe des Servers und wartet auf Informationen über die Generierung von Objekten, das Löschen von Objekten, Versenden von Nachrichten und ähnliches. Die autonomen Objekte der Beispielanwendung wurden so implementiert, daß sie die gewünschten Informationen per Socket-Verbindung an den Server schicken; so wird z. B. bei jedem Versenden einer Nachricht asynchron der Server über diese Nachricht informiert.

Nach der Implementierung des Sicherheitssystems in AutO, das in Kapitel 5 vorgestellt wurde, ist es autonomen Objekten normalerweise nicht erlaubt, Kommunikationsverbindungen aufzubauen. Außerdem ist autonomen Objekten der Zugriff auf das Dateisystem verwehrt. Sie dürfen die Konfigurationsdatei nicht lesen, die unter anderem die Informationen enthält, auf welchem Rechner und Port der Server läuft. Um die Beispielanwendung ausführen zu können, ist es nötig, den autonomen Objekten der Beispielanwendung diese Rechte zu gewähren. Eine Konfigurationsdatei, die genau dies leistet, zeigt Abbildung 6.3.

6.3 Generierung eines Auftrages

Damit Kunden einen Auftrag erzeugen können, steht ihnen eine komfortable grafische Benutzeroberfläche zur Verfügung, die in Abbildung 6.4 gezeigt wird.

Diese basiert auf einem Standard-Terminal-Client, den das AutO-System zur Verfügung stellt. Damit hat man die Möglichkeit, eine Verbindung zu einem Terminal-Server aufzubauen, einem speziellen AutO-Prozeß, der Befehle von Terminal-Clients entgegennimmt und ausführt. Um einen Auftrag zu generieren, wird eine Transaktion durch den Terminal-Client gestartet, die das Auftragsobjekt erzeugt und initialisiert. Abbildung 6.5 zeigt die

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/" {
      permission java.security.AllPermission;
};

// permissions for the AutO-system
grant codeBase "file:${auto.home}/src/" {
      permission java.security.AllPermission;
};

// permissions for user-definded objects (demo-application)
grant codeBase "file:${userclasses.home}/" {
      permission java.net.SocketPermission "Braves:22222", "connect";
      permission java.util.PropertyPermission "user.*", "read";
      permission java.io.FilePermission "/home/db/seltzsam/AutODemo.config", "read";
      permission java.io.FilePermission "/home/db/seltzsam/AutODemo.config.braves", "read";
      permission java.io.FilePermission "/home/db/seltzsam/AutODemo.config.indians", "read";
};
```

Abbildung 6.3: Beispiel einer „AutO.policy“ für die Beispielanwendung

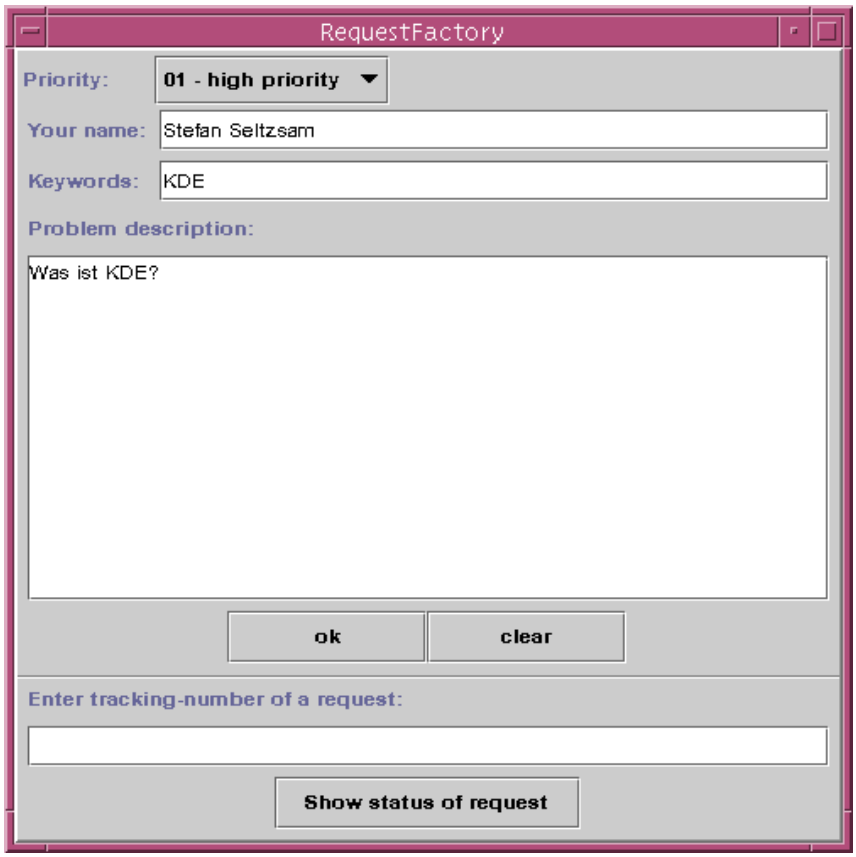


Abbildung 6.4: Benutzerschnittstelle zur Erteilung von Aufträgen

für den Benutzer arbeitende Transaktion `CreateRequest`, die ein Auftragsobjekt erzeugt hat und es nun initialisiert. Dazu sendet sie eine Nachricht an das Objekt, im Bild durch die Linie zwischen diesen beiden Objekten repräsentiert. Das Schloß auf der Nachrichtenlinie deutet an, daß die Kommunikation über sichere, d. h. verschlüsselte, Kanäle stattfindet.

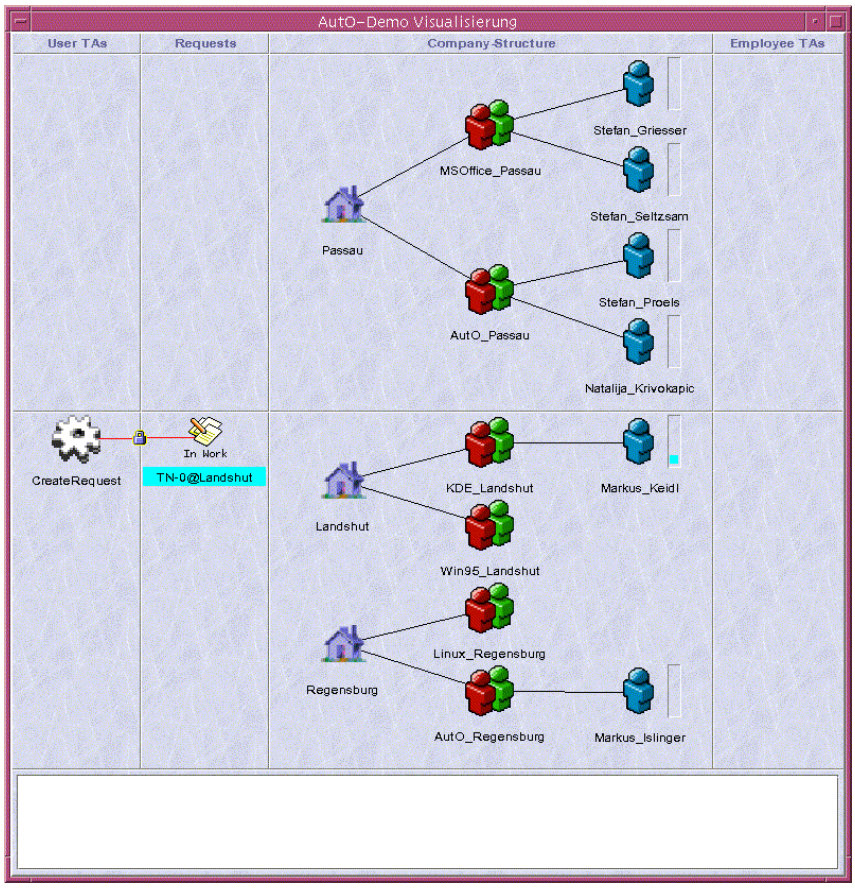


Abbildung 6.5: Generierung eines Auftragobjektes

Nach dem Commit der Transaktion wird dem Benutzer angezeigt, unter welcher Auftragsnummer sein Auftrag bei dem Unternehmen registriert ist. Dem in diesem Beispiel generierten Auftrag wurde `TN-0@Landshut` zugewiesen (siehe Abbildung 6.6). Mit Hilfe dieses Identifikationsmerkmales kann ein Benutzer des Systems mit der hier vorgestellten Benutzerschnittstelle den aktuellen Stand seines Auftrages abfragen.

6.4 Bearbeitung der Aufträge

Nach der Generierung der Kundenaufträge müssen diese bearbeitet werden. Die Objekte, die die Angestellten im System repräsentieren, können diese Aufgabe nicht selbst übernehmen. Sie benötigen dazu die Unterstützung des realen Arbeitnehmers, den sie im System repräsentieren. Auch für diesen Zweck steht eine grafische Benutzeroberfläche auf der Basis eines Terminal-Clients zur Verfügung, der sogenannte AutO-Desktop. Um die

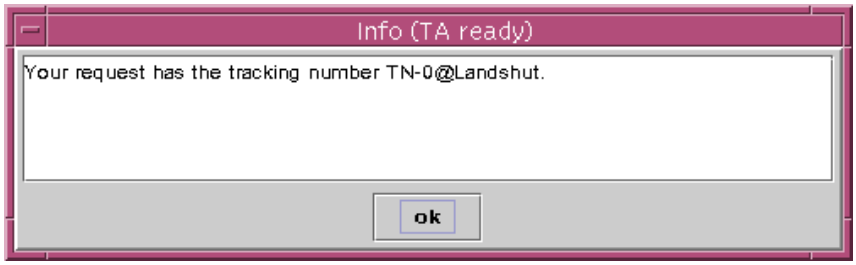


Abbildung 6.6: Information über das generierte Auftragsobjekt

Arbeitsweise dieser Benutzerschnittstelle zu demonstrieren, wurden dem System mehrere Aufträge erteilt, wie in Abbildung 6.7 gezeigt. Die kleinen (farbigen) Kästchen neben den Angestellten symbolisieren die Aufträge, die den einzelnen Arbeitnehmern zugeteilt sind. Der Angestellte Markus Keidl hat in diesem Beispiel zwei Aufträge zur Bearbeitung vorliegen. Diese werden ihm auch von seinem AutO-Desktop angezeigt, wie Abbildung 6.8 zeigt.

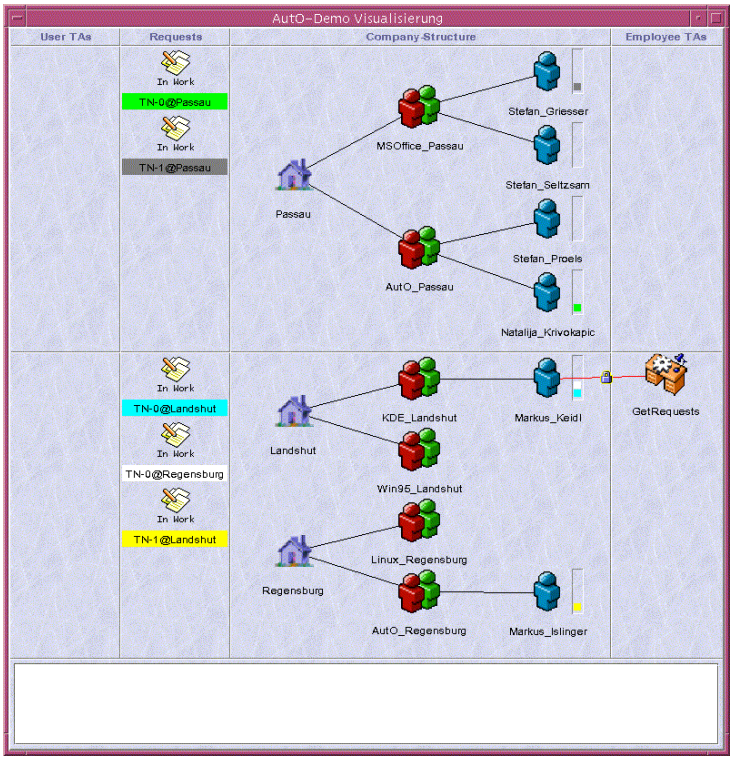


Abbildung 6.7: Mehrere Aufträge verteilt auf verschiedenen Arbeitnehmer.

Mit Hilfe des AutO-Desktops kann der Arbeitnehmer in Erfahrung bringen, welche Aufträge er zu bearbeiten hat. Dafür wird eine Transaktion (die in Abbildung 6.7 zu sehen ist) gestartet, die das Objekt, das den Angestellten im System repräsentiert, befragt, welche Aufträge ihm zugewiesen sind. Die Antwort dieses Objektes wird vom Desktop ausgewer-

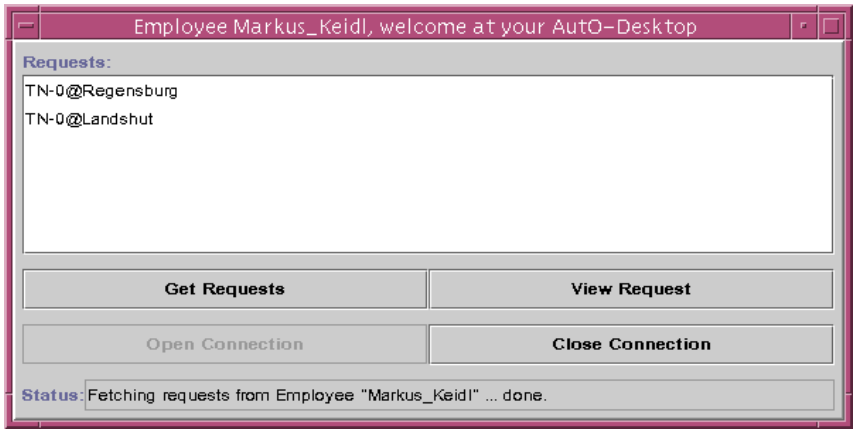


Abbildung 6.8: Der AutO-Desktop des Angestellten Markus Keidl

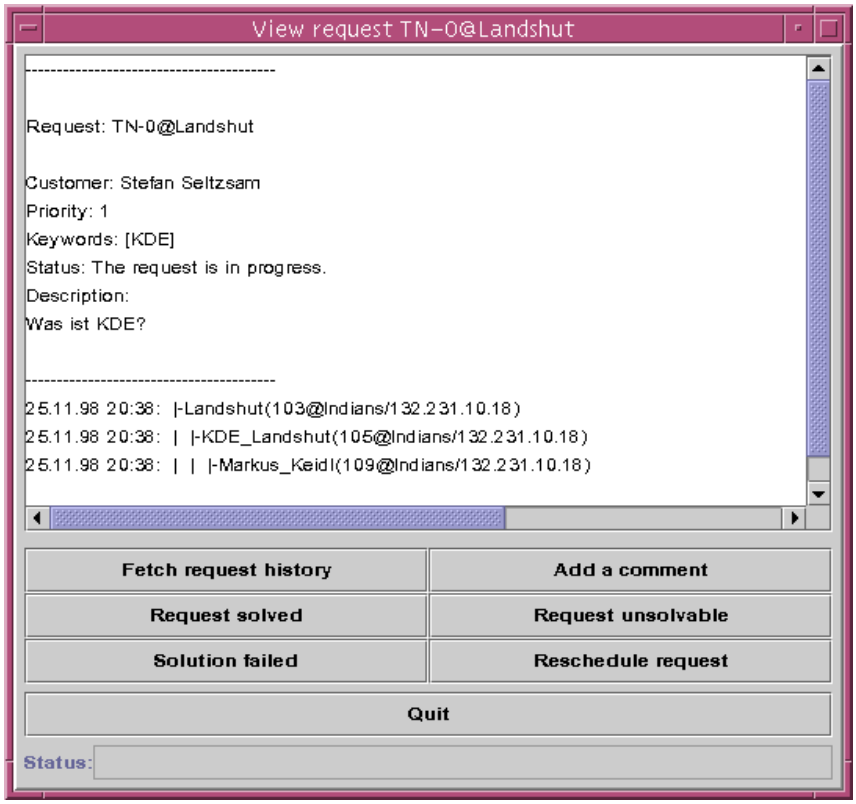


Abbildung 6.9: Die Detail-Ansicht eines Auftrages

tet und angezeigt. Dieser kann nun einen der zur Wahl stehenden Aufträge bearbeiten (Abbildung 6.9).

In dieser Beispielanwendung stehen Angestellten folgende Möglichkeiten zur Verfügung:

- **Die im Auftragsobjekt gespeicherten Informationen erneut abrufen:** Dies wäre zum Beispiel nötig, wenn das Auftragsobjekt nicht erreichbar war, weil z. B. eine Netzverbindung ausgefallen ist.
- **Kommentar hinzufügen:** Fügt einen Kommentar zu den im Auftrag gespeicherten Informationen hinzu. Denkbar wäre dabei etwa ein Lösungsansatz, den der Angestellte erst später ausarbeiten, die Idee dazu aber sofort festhalten will.
- **Auftrag lösen:** Hier kann der Angestellte die Frage des Kunden beantworten und den Auftrag als fertig bearbeitet markieren.
- **Auftrag unlösbar:** Diese Option ist sinnvoll, wenn ein Angestellter feststellt, daß die Frage des Kunden generell nicht beantwortet werden kann, weil z. B. eine zu unspezifische Frage gestellt wurde. Der Auftrag wird daraufhin mit einer Begründung versehen und als unlösbar markiert.
- **Auftrag für Angestellten nicht lösbar:** Kann der Angestellte einen Auftrag aus irgendwelchen Gründen nicht bearbeiten, kann er diese Option wählen und damit dem Auftrag einen Kommentar hinzufügen. Der Auftrag wird daraufhin an einen anderen Angestellten innerhalb des Unternehmens weitergeleitet, solange noch mindestens ein geeigneter Arbeitnehmer existiert, der den Auftrag noch nicht bearbeitet hat.
- **Auftrag weiterleiten:** Einen Auftrag weiterzuleiten ist sinnvoll, wenn z. B. dringende Aufträge anstehen, die der Angestellte nicht alle gleichzeitig erledigen kann, ohne damit in Verzug zu geraten.

Alle diese Vorgänge werden durch das Visualisierungstool dargestellt. Abbildung 6.10 zeigt einen gelösten Auftrag und einen als unlösbar eingestuften. Wird das nächste Mal von einem Kunden der Status eines dieser Aufträge abgefragt, wird dieser zurückgeliefert und die Aufträge registrieren sich daraufhin beim Archiv (in der Visualisierungskomponente unten angezeigt), das alle bereits komplett erledigten Aufträge beinhaltet. Dieses Archiv könnte damit z. B. verwendet werden, um zu festgelegten Zeiten die in ihm enthaltenen Objekte auf Band zu speichern und daraufhin aus dem System zu entfernen.

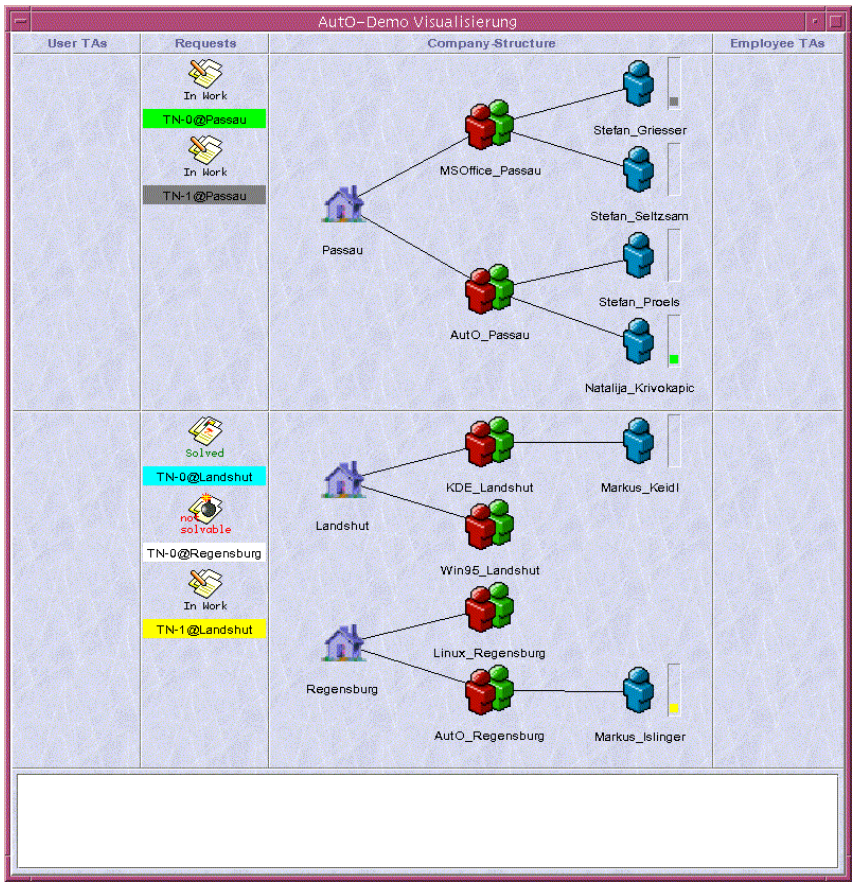


Abbildung 6.10: Darstellung bearbeiteter und unlösbarer Aufträge

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurden verschiedene Aspekte von AutO, einem persistenten, verteilten System autonomer Objekte betrachtet. Die erste Aufgabe war es, die zwei verschiedenen Migrationsstrategien Good-LAN und Best-LAN zu vergleichen. Diese Untersuchungen haben gezeigt, daß die Heuristik, mit der die Good-LAN Strategie arbeitet, bei den Einsparungen an Kommunikationskosten sehr gut mit der vollständigen Suche, die Best-LAN durchführt, mithalten kann und dabei signifikant geringere CPU-Kosten anfallen.

Danach wurde gezeigt, wie der verteilte Nameservice von André Eickler an das AutO System angebunden werden kann. Dazu wurde ein Protokoll festgelegt, das vom Nameservice-Treiber auf der Seite von AutO und vom multithreaded Nameserver auf der Seite des verteilten Nameservice benutzt wird, um miteinander zu kommunizieren. Außerdem wurde gezeigt, wie die verschiedenen Objektidentifikatoren aufeinander abgebildet und die Nameserver des verteilten Dienstes sinnvoll im System verteilt werden können.

Den zentralen Teil der Arbeit bildete die Erweiterung des bestehenden Sicherheitskonzeptes in AutO um eine Komponente, die sich um die Sicherheitsaspekte der Migration kümmert. Zuerst wurden verschiedene Möglichkeiten der Kategorisierung der Objekte und Rechner in AutO aufgezeigt. Das Ziel dieser Kategorisierung war es, einen Maßstab vorzugeben, mit dessen Hilfe festgestellt werden kann, auf welche Rechner ein Objekt migrieren darf, um sicherzustellen, daß diesem stets angemessener Schutz zuteil wird. Damit die Strategien Good-LAN und Best-LAN in einem derart modifizierten System effizient arbeiten können, wurden sie so angepaßt, daß sie stets nur Rechner als Migrationsziel in Betracht ziehen, die auch eine geeignete Klassifikation besitzen. Anschließend wurde ein auf den Sicherheitsmechanismen von Java basierendes Verfahren vorgestellt, das die AutO-Rechner und das AutO-System selbst vor unerlaubten Handlungen autonomer Objekte und Transaktionen schützt, um den Gefahren vorzubeugen, die mobiler Code mit sich bringt. Abschließend wurde ein Signaturserver beschrieben, der es ermöglicht, Daten für verschiedene Zwecke zu signieren.

Als abschließende Aufgabe dieser Arbeit wurde noch ein einfaches Support-Center als Beispielanwendung für AutO implementiert. Um einen Einblick in die Arbeitsabläufe innerhalb des Systems zu erlangen, wurde eine Visualisierungskomponente entwickelt, die es unter anderem erlaubt, die Kommunikation zwischen den Objekten darzustellen.

7.2 Ausblick

Im Rahmen einer Diplomarbeit ist es nicht immer möglich, alle Aspekte einer Aufgabe perfekt zu lösen. Manchmal ist es aus Zeit- oder Komplexitätsgründen notwendig, Kompromisse in der Flexibilität einer Lösung einzugehen oder gewisse Teilaspekte eines Problems auszublenden. Auch diese Arbeit bietet noch einige Ansatzpunkte für Verbesserungen und weitere Untersuchungen, wie die folgenden Punkte zeigen:

- Die Anbindung des verteilten Nameservice an AutO wirft die interessante Frage auf, welche Auswirkungen dieser auf die Performance des Systems hat. Dazu müßten verschiedene Benchmarks durchgeführt werden. Um den Einfluß des Nameservice genau beurteilen zu können, wären Messungen in verschiedenen Netzwerkstrukturen und mit verschiedenen Nameserveranordungen und -einstellungen nötig. Die Doktorarbeit von André Eickler [Eic98] enthält Messungen, die den Nameservice separat betrachten. Inwieweit diese Ergebnisse auf AutO übertragbar sind, müßte untersucht werden.

- Die bisherige Anbindung des verteilten Nameservice bietet die Funktionalität, die einem Nameservice im AutO-System ursprünglich zugedacht war. Denkbar wäre es hier, weitergehende Informationen zwischen AutO und dem Nameservice auszutauschen, um dynamisch ein sinnvolles Replikationsschema der Indexknoten im Nameservice zu erstellen.

- Die Komponente AutOSystemStructure, die den Migrationsstrategien Good-LAN und Best-LAN Informationen über die Netzstruktur und die Sicherheitsklassen der Rechner im System liefert, ist statisch. Die Informationen über die Kommunikationskosten zwischen den LANs im System werden aus einer Datei gelesen. Die Sicherheitsklassen der Rechner werden bei der Instanziierung der AutOSystemStructure von den beteiligten Rechnern angefordert. Um während des Betriebs von AutO flexibel zu sein, wäre es nötig, daß zusätzliche Rechner zum System hinzugefügt bzw. bereits ins System integrierte entfernt werden könnten. Kommen neue Rechner hinzu, wäre es zumindest für die Good-LAN Strategie nötig, die Hierarchie der Rechner neu zu bestimmen. Außerdem wäre es wünschenswert, daß das System die Kommunikationskosten zwischen den LANs selbständig während des normalen Betriebs bestimmt und diese Informationen regelmäßig aktualisiert. Eine sinnvolle Erweiterung dieser Komponente wäre auch, Änderungen der Sicherheitsklasse eines Rechners durch ein geeignetes Protokolls an alle Rechner im System zu propagieren.

- Um nachzuweisen, daß das in dieser Arbeit erweiterte Sicherheitskonzept von AutO wirklich „sicher“ ist, wäre eine formale Verifikation nötig.

Anhang A

Konfiguration und Benutzung des AutO-Systems

Im Rahmen dieser Diplomarbeit wurde eine Reihe von Hilfsprogrammen entwickelt, die verschiedene Aufgaben in AutO übernehmen. Außerdem wurden dem System neue Komponenten hinzugefügt und bestehende erweitert. Dieser Abschnitt erklärt die Benutzung der Anwendungen und die neuen Konfigurationsmöglichkeiten des Systems.

A.1 Die Datei `AutO.config`

Jeder Prozeß des AutO-Systems liest zu Beginn eine Konfigurationsdatei ein, die im Home-Verzeichnis des Benutzers gesucht wird, der den Prozeß startet. Zuerst wird dabei nach einer Datei namens `AutO.config.`*`Rechnername`* gesucht, wobei *`Rechnername`* den Namen des Rechners angibt, auf dem der Prozeß gestartet wird. Wird keine derartige Datei gefunden, wird die Datei `AutO.config` geladen. Existiert auch diese nicht, wird die Ausführung des Prozesses beendet. Diese Datei ist in verschiedene Sektionen unterteilt, die angeben, für welche Komponente in AutO die Parameter bestimmt sind. Dabei stehen einige Variablen zur Verfügung, die innerhalb der Konfigurationsdatei verwendet werden können und beim Lesen der Datei durch ihre Werte ersetzt werden:

- `AUTO_HOME`: Enthält den Verzeichnisnamen, im dem das AutO-System installiert wurde.

- `home`: Wird durch den Verzeichnisnamen ersetzt, der das Home-Verzeichnis des Benutzers repräsentiert.

Soll eine dieser Variablen verwendet werden, muß der entsprechende Name zwischen geschweiften Klammern angegeben werden.

Sektion [NameService]

- `nameserverlinktype`: Hier kann man den Namen der Klasse angeben, die AutO verwendet, um mit dem Nameservice zu kommunizieren.
  Erlaubte Werte:
  - `nameservice.DistributedNameServerLink` (verteilter Nameservice)
  - `nameservice.NameServerLink` (zentraler Nameservice)

  Voreingestellter Wert: `nameservice.NameServerLink`

Sektion [Security]

- `allowGraphicalPasswordDialog`: Dieses Flag gibt an, ob das AutO-System einen graphischen Benutzerdialog verwenden darf, wenn ein Paßwort eingegeben werden soll. Vorteil dieses Dialogfensters ist es, daß das Paßwort nicht am Bildschirm sichtbar ist.
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `on`

- `securitymanager`: Dieses Flag erlaubt es, den SecurityManager von AutO abzuschalten. Dadurch erhalten benutzerdefinierte autonome Objekte und Transaktionen uneingeschränkten Zugriff auf die Ressourcen des Rechners. Aus diesem Grund wird dringend davon abgeraten, den AutOSecurityManager zu deaktivieren.
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `on`

- `allowSystemExit`: Mit Hilfe dieses Schalters ist es möglich, autonomen Objekten und Transaktionen zu erlauben, die Java Virtual Machine zu beenden. Gedacht war dieser Schalter dazu, Persistenz und Recovery testen zu können. Aus Kompatibilitätsgründen wurde diese Möglichkeit auch in den AutOSecurityManager übernommen. Von der Aktivierung dieser Option wird dringend abgeraten!
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `off`

- `policyfileclass`: Hier kann der Name der Klasse eingetragen werden, die verwendet wird, um die Konfigurationsdatei `AutO.policy`, die der AutOSecurityManager verwendet, einzuladen. Auf diese Datei wird in einem späteren Abschnitt noch eingegangen werden. Die Klasse, die hier angegeben wird, muß kompatibel zu der von SUN bei Java 1.2 mitgelieferten Klasse `java.security.PolicyFile` sein.
  Voreingestellter Wert: Wird hier kein Klassenname angegeben, wird die Klasse verwendet, die in der Konfigurationsdatei `java.security` von Java 1.2 angegeben ist.

- `policyfilename`: Definiert den Namen der Konfigurationsdatei, die vom AutO-ClassLoader verwendet wird.
  Voreingestellter Wert: {AUTO_HOME}/AutO.policy

- `userclasses`: Spezifiziert den Namen des Verzeichnisses, in dem die Klassen benutzerdefinierter autonomer Objekte und Transaktionen abgelegt sind. Hier werden auch Klassendateien gespeichert, die vom AutOClassLoader aus dem Netz geladen werden. Dieser Pfad muß angegeben werden!

- `forceUserClassSignatures`: Müssen die Klassendateien benutzerdefinierter autonomer Objekte und Transaktionen vom Signaturserver signiert sein, kann man hier `on` angeben. Der AutOClassLoader verweigert dann den Ladevorgang für alle Klassen, die keine gültige Signatur besitzen.
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `off`

- `permittedclasses`: Erlaubt es, eine Liste von Klassennamen anzugeben, die von autonomen Objekten und Transaktionen geladen werden dürfen. Alle derartigen Klassen, die für einen normalen Betrieb von AutO notwendig sind, sind fest im AutOClassLoader eingetragen und müssen (und dürfen!) hier nicht angegeben werden. Die einzelnen Elemente der Liste werden durch Kommata voneinander getrennt.
  Voreingestellter Wert: Leere Liste

- `securityclass-file`: Gibt den Namen der Datei an, die die Sicherheitsklasse für diesen Rechner enthält.
  Voreingestellter Wert: {AUTO_HOME}/.AutO-SecurityClass

- `use-securityclasses`: Schaltet die Verwendung von Sicherheitsklassen an oder aus. Falls die Verwendung von Sicherheitsklassen aktiviert ist, muß eine signierte Sicherheitsklasse für den Rechner vorliegen. Außerdem sind in diesem Fall nur Migrationen von Objekten auf Rechner möglich, die den Sicherheitsansprüchen der Objekte genügen. Falls die Verwendung der Sicherheitsklassen abgeschaltet ist, kann ein Objekt auf jeden beliebigen Rechner im System migrieren. (Hinweis: Autonome Objekte besitzen stets eine Sicherheitsklasse, diese Option schaltet nur die Prüfung der Sicherheitsklasse bei Migrationen und die Verwendung der Sicherheitsklasse des Rechners an oder aus.)

  Soll sichergestellt werden, daß Objekte nur auf Rechner migrieren, die ihren Sicherheitsansprüchen genügen, muß diese Option eingeschaltet werden.
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `off`

Sektion `[SignatureServer]`

- `signatureserver-host`: Spezifiziert den Rechner, auf dem der Signaturserver läuft.

- `signatureserver-port`: Spezifiziert den Port, auf dem der Signaturserver Verbindungen entgegennimmt.
  Voreingestellter Wert: 30001

- `signtureserver-keyfile`: Hier kann der Name der Datei eingetragen werden, die die Schlüssel des Signaturservers enthält. Dieser Parameter wird nur auf dem Rechner benötigt, auf dem der Signaturserver läuft.
  Voreingestellter Wert: {AUTO_HOME}/.sigserv_keystore

- `security-administrator-keyfile`: Gibt den Namen der Datei an, die die Schlüssel des Sicherheitsadministrators enthält. Diese Option wird vom SATool (siehe A.5) ausgewertet.
  Voreingestellter Wert: {AUTO_HOME}/.admin_keystore

- `security-representative.name`: Gibt den Namen des Sicherheitsbeauftragten an, der für diesen Rechner verantwortlich ist. Dadurch ist es nicht nötig, diesen Namen bei jedem Start des SRTools (siehe A.6) anzugeben.

- `security-representative.keystore`: Spezifiziert den Namen der Datei, die die Schlüssel des Sicherheitsbeauftragten enthält. Dadurch muß dieser Dateiname nicht bei jedem Start des SRTools angegeben werden.
  Voreingestellter Wert: {home}/.Security-Representative-Keystore

A.2 Die Datei `AutO.policy`

Der AutOSecurityManager (siehe Abschnitt 5.5.1) überprüft anhand der Klassen, die auf dem Ausführungsstack liegen, ob eine Aktion, die gerade ausgeführt werden soll, erlaubt ist oder nicht. Dabei ist er standardmäßig so konfiguriert, daß benutzerdefinierte Klassen keinerlei spezielle Rechte haben und das AutO-System selbst keinen Restriktionen unterliegt. Da diese sehr restriktive Sicherheitspolitik nicht für alle Fälle geeignet ist, weil z. B. autonomen Objekten der Aufbau von Netzwerkverbindungen erlaubt werden soll, ist diese konfigurierbar. Dazu wird vom AutOSecurityManager bei seiner Instanzierung die Datei `AutO.policy` im Home-Verzeichnis gelesen (dieser Dateiname kann in der Konfigurationsdatei AutO.config festgelegt werden). Die Syntax dieser Datei stimmt mit der überein, die JDK 1.2 verwendet, um seine Sicherheitsstrategie festzulegen (durch die Datei `java/lib/security/java.policy`). Aus diesem Grund kann zur Bearbeitung dieser Datei alternativ zu einem normalen Texteditor das *PolicyTool* verwendet werden, das das JDK 1.2 anbietet. Eine Beschreibung der verwendeten Syntax findet sich im Installationspfad von Java 1.2 im Verzeichnis `docs/guide/security/PolicyFiles.html` oder im Internet [Sun98a]. Dabei stehen zwei zusätzliche Properties zur Verfügung, die die Definition der Sicherheitspolitik vereinfachen sollen:

- `auto.home`: Wird durch den Verzeichnisnamen ersetzt, der in der AutO.config dem Parameter AUTO_HOME zugewiesen wurde.

- `userclasses.home`: Wird durch den Verzeichnisnamen ersetzt, der in der Datei AutO.config als Verzeichnis für benutzerdefinierte autonome Objekte und Transaktionen festgelegt wurde.

Die Standardkonfiguration des AutOClassLoaders ist in Abbildung A.1 wiedergegeben. Hier werden den Klassen, die im Installationspfad von AutO liegen, alle Rechte zugewiesen. Da Java 1.2 das Konzept der Standarderweiterungen kennt, werden auch die Klassen, die im System als Standarderweiterungen registriert sind, den normalen Systemklassen gleichgestellt und unterliegen damit keinerlei Restriktionen. Benutzerdefinierten Klassen werden in der Standardkonfiguration keine Rechte zugewiesen.

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/" {
      permission java.security.AllPermission;
};

// permissions for the AutO-system
grant codeBase "file:${auto.home}/src/" {
      permission java.security.AllPermission;
};

// permissions for user-definded objects
grant codeBase "file:${userclasses.home}/"
{
};
```

Abbildung A.1: Standard-Version der Datei AutO.policy

Rechte können dabei auch in Abhängigkeit von der Signatur einer Klasse vergeben werden. Dies kann dazu genutzt werden, gewisse Aktionen signierten Klassen vorzuenthalten.

A.3 Der Signaturserver

Der Signaturserver (`SignatureServer`) ist ein Server, der verschiedene sicherheitsrelevante Dienste anbietet. Er signiert die Sicherheitsklassen von Rechnern, Klassendateien und verschiedene öffentliche Schlüssel, die im AutO-System verwendet werden. Bevor der Server eingesetzt werden kann, müssen die dazu nötigen Schlüssel unter Verwendung des Hilfsprogramms *CreateInitialKeystores* generiert werden.

Aufruf:

```
java [properties] security.signatureserver.SignatureServer [parameters]
```

Properties:

Properties sind in Java Umgebungsvariablen, die beim Start der Java Virtual Machine gesetzt werden können (siehe Syntax von Java) und dann global für alle Klassen, die die JVM lädt, zur Verfügung stehen. Der Signaturserver kennt die folgenden Properties:

- `sigserv.keystore.password`: Hiermit kann man das Paßwort für den Keystore (Datei, die Schlüssel für kryptographische Algorithmen enthält) des Signaturservers setzen. Wird das Paßwort hier nicht angegeben, wird während des Starts des Signaturservers danach gefragt. Diese Option ist nur für Testzwecke gedacht, da das Paßwort im Klartext auf dem Bildschirm erscheint.
- `DEBUG`: Setzt man dieses Flag auf `true`, gibt der Signaturserver Debug-Informationen aus.

Kommandozeilenparameter:

- `-help`: Zeigt einen Hilfetext an.
- `-keystore <file>`: <file> gibt den Keystore an, in dem die Schlüssel für den Signaturserver gespeichert sind. Wird dieser Parameter nicht angegeben, wird der Wert aus der Konfigurationsdatei AutO.config verwendet.

Der Signaturserver bietet seine Dienste ausschließlich über das Netz an. Um ihn zu verwenden, stehen die Hilfsprogramme SATool und SRTool zur Verfügung, die im folgenden noch erklärt werden.

A.4 Das Tool CreateInitialKeystores

CreateInitialKeystores generiert die initialen RSA-Schlüsselpaare, die zum Betrieb des Signaturservers nötig sind. Dabei handelt es sich um das Schlüsselpaar, das der Signaturserver zum Signieren von Daten verwendet, und um das Schlüsselpaar des Sicherheitsadministrators von AutO, der für die Verwaltung des Signaturservers verantwortlich ist.

Aufruf:

```
java security.signatureserver.CreateInitialKeystores
```

Das Tool ist interaktiv und fragt alle benötigten Informationen vom Benutzer ab. Voreinstellungen der Werte werden aus der Datei AutO.config gelesen und können vom Benutzer durch Drücken der Return-Taste übernommen werden. Die folgenden Daten werden jeweils für den Signaturserver und den Sicherheitsadministrator abgefragt:

- Dateiname für den Keystore
- Paßwort für den Keytore
- Länge der zu generierenden Schlüssel

Achtung: Werden mit diesem Tool neue Schlüssel generiert, werden alle Schlüssel im System, die mit dem alten Schlüssel des Signaturservers signiert wurden, ungültig und müssen neu signiert werden!

A.5 Das SATool

Das SATool (*Security Administrator Tool*) bietet dem Sicherheitsadministrator von AutO die Möglichkeit, mit dem Signaturserver zu kommunizieren. Es kann nur eingesetzt werden, wenn ein Signaturserver läuft und dem Tool die Schlüssel des Sicherheitsadministrators zur Verfügung stehen.

Aufruf:

```
java [properties] security.signatureserver.SATool [parameters] [command]
```

Properties:

- `DEBUG`: Setzt man dieses Flag auf true, werden Debug-Informationen ausgegeben.
- `security-admin.password`: Hiermit kann man das Paßwort für den Keystore des Sicherheitsadministrators setzen. Wird das Paßwort hier nicht angegeben, wird es während des Programmstarts abgefragt. Diese Option ist nur für Testzwecke gedacht, da das Paßwort im Klartext auf dem Bildschirm erscheint.

Kommandozeilenparameter:

- `-help`: Zeigt einen Hilfetext an.
- `-v`: Mit dieser Option gibt das SATool Informationen über interne Arbeitsabläufe aus.
- `-keystore <file>`: <file> gibt den Keystore an, der die Schlüssel des Sicherheitsadministrators enthält. Wird dieser Parameter nicht angegeben, wird der Wert aus der Konfigurationsdatei AutO.config verwendet.

- `-server <host>`: Durch die Angabe von <host> wird der Rechner spezifiziert, auf dem der Signaturserver läuft. Auch dieser Parameter muß nicht angegeben werden, wenn der entsprechende Wert in der AutO.config-Datei eingetragen ist.
- `-port <port>`: Spezifiziert den Port, auf dem der Signaturserver Verbindungen akzeptiert. Dieser Wert kann ebenfalls alternativ in der Datei AutO.config definiert werden.

Kommandos:

- `list`: Gibt eine Liste aller Sicherheitsbeauftragten aus, die dem Signaturserver bekannt sind und von ihm akzeptiert werden.
- `add <name> <paßwort> <schlüssellänge> <dateiname>`: Generiert Schlüssel für einen neuen Sicherheitsbeauftragten und registriert diesen beim Signaturserver. Die Parameter haben folgende Bedeutung:
  - `name`: Spezifiziert den Namen des neuen Sicherheitsbeauftragten.
  - `paßwort`: Hier kann man das Paßwort angeben, mit dem der Keystore geschützt wird, der die Schlüssel des neuen Sicherheitsbeauftragten enthält. Gibt man hier „`***`“ an, wird das Paßwort mittels einer Dialogbox abgefragt.
  - `schlüssellänge`: Gibt die Länge der zu generierenden Schlüssel an. Wie dieser Wert interpretiert wird, ist abhängig vom verwendeten Verschlüsselungsalgorithmus.
  - `dateiname`: Spezifiziert den Dateinamen des Keystores, der für den neuen Sicherheitsbeauftragten angelegt wird.
- `remove <name>`: Löscht den angegebenen Sicherheitsbeauftragten aus der Liste der vom Signaturserver akzeptierten Beauftragten. Diesem ist es damit nicht mehr möglich, dem Signaturserver Aufträge zu erteilen.
- `shutdown`: Empfängt der Signaturserver diesen Befehl, akzeptiert er keine neuen Verbindungen mehr. Sind alle gerade aktiven Aufträge abgearbeitet, beendet er sich.
- `panic-shutdown`: Empfängt der Signaturserver dieses Kommando, beendet er sich sofort, ohne die aktiven Verbindungen ordungsgemäß zu beenden.

A.6 Das SRTool

Das SRTool (*Security Representative Tool*) bietet den Sicherheitsbeauftragten die Möglichkeit, den Signaturserver anzusprechen. Es kann nur von Sicherheitsbeauftragten verwendet werden, die beim Signaturserver registriert sind. Mit Hilfe dieses Tools können unter anderem folgende Aktionen ausgeführt werden:

- Signierung/Zertifizierung von öffentlichen Schlüsseln,
- Generierung von signierten Sicherheitsklassen für die Rechner im AutO-System (siehe Abschnitt 5.2.3),
- Erstellung von Signaturen von Klassendateien (siehe Abschnitt 5.5.3).

Dieses Tool entstand in Zusammenarbeit mit Markus Keidl und wird in [Kei99] ausführlich beschrieben. An dieser Stelle sei nur darauf verwiesen, daß der Befehl

```
java security.srtool.SRTool help
```

eine ausführliche Beschreibung der Syntax des Programms auf dem Bildschirm ausgibt.

Anhang B

Konfiguration und Benutzung der Beispielanwendung

Für die Beispielanwendung, die ein einfaches Support-Center modelliert, wurden verschiedene Benutzeroberflächen entwickelt, die die komfortable Bedienung der Anwendung erlauben. Außerdem wurde eine Visualisierungskomponente implementiert, die es gestattet, die Vorgänge innerhalb des Systems zu verdeutlichen. In diesem Abschnitt wird die Handhabung dieser Programme erklärt.

B.1 Die Datei `AutODemo.config`

Die verschiedenen Teile der Beispielanwendung verwenden eine gemeinsame Konfigurationsdatei, die jeweils beim Start einer Komponente geladen wird. Dabei wird zuerst im lokalen Verzeichnis nach einer Datei mit dem Namen `AutODemo.config.`*`Rechnername`* gesucht. Wird keine derartige Datei gefunden, wird versucht, die Datei `AutODemo.config` im lokalen Verzeichnis zu laden. Scheitert auch dieser Versuch, wird nach der Datei `AutODemo.config.`*`Rechnername`* und dann nach `AutODemo.config` im Home-Verzeichnis des Benutzers gesucht. Die Konfigurationsdatei verwendet die Syntax, die Java selbst auch für Property-Dateien verwendet: In jeder Zeile wird ein Schlüssel-Wert-Paar angegeben, z. B.

```
bigbrother.server.port = 22222
```

Konfigurationsmöglichkeiten

- `home`: Gibt das Installationsverzeichnis der Beispielanwendung an.
  Voreingestellter Wert: {user-home}/autojava/application
  Dabei entspricht user-home dem Home-Verzeichnis des Benutzers der Anwendung.
- `debug.level.app`: Setzt den Debug-Level der Anwendung. Je höher dieser Wert ist, um so mehr Informationen werden ausgegeben. Diese Option ist nur für die

Fehlersuche gedacht.
Voreingestellter Wert: 0

- `debug.level.bigbrother`: Setzt den Debug-Level des BigBrothers, also der Visualisierungskomponente der Beispielanwendung.
  Voreingestellter Wert: 0

- `server.host`: Gibt an, auf welchem Rechner der Terminal-Server läuft, zu dem die verschiedenen Terminal-Clients eine Verbindung aufbauen sollen.
  Voreingestellter Wert: der lokale Rechner

- `server.port`: Der Port, auf dem der Teminal-Server neue Verbindungen erwartet.
  Voreingestellter Wert: 2222

- `bigbrother.server.host`: Gibt den Rechner an, auf dem der BigBrother (= Visualisierungskomponente) läuft, der die Abläufe innerhalb der Anwendung anzeigt.
  Voreingestellter Wert: braves

- `bigbrother.server.port`: Der Port, auf dem der BigBrother neue Verbindungen entgegennimmt.
  Voreingestellter Wert: 22222

- `employee.name`: Der Name eines Angestellten. Dieser wird verwendet, wenn ein EmployeeDesktop, also die Benutzeroberfläche eines Angestellten, gestartet wird.
  Voreingestellter Wert: Der Name, unter dem der Benutzer beim Betriebssystem registriert ist.

- `employee.keystore`: Der Name des Keystores, der die Schlüssel des Angestellten für den Verbindungsaufbau mit dem Server enthält [Kei99].
  Voreingestellter Wert: {home}/.{employee.name}-KeyStore

- `factory.login`: Der Name, der dazu verwendet wird eine RequestFactory, also die Benuzeroberfläche zum Generieren von Aufträgen, zu starten. Dieser Name wird benötigt, um die Schlüssel für eine sichere Verbindung zum Server im angegebenen Keystore zu finden.
  Voreingestellter Wert: Der Name, unter dem der Benutzer beim Betriebssystem registriert ist.

- `factory.keystore`: Der Name des Keystores der RequestFactory.
  Voreingestellter Wert: {home}/.{factory.login}-KeyStore

- `security.messages.transactions.outgoing.mac`: Gibt an, ob bei Nachrichten von Transaktionen an Objekte ein *Message Authentication Code* (MAC) [U.S85] berechnet werden soll. Durch einen MAC wird die Integrität der Nachricht sichergestellt [Kei99].
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `on`

- `security.messages.transactions.incoming.mac`: Gibt an, ob Nachrichten, die von Transaktionen empfangen werden, einen MAC enthalten müssen.
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `on`

- `security.messages.transactions.outgoing.encryption`: Gibt an, ob Nachrichten von Transaktionen verschlüsselt werden sollen.
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `on`

- `security.messages.transactions.incoming.encryption`: Gibt an, ob Nachrichten, die von Transaktionen empfangen werden, verschlüsselt sein müssen.
  Erlaubte Werte: `on`, `off`
  Voreingestellter Wert: `on`

B.2 Der BigBrother

Der BigBrother ist die Visualisierungskomponente der Beispielanwendung. Er hat die Aufgabe, alle Objekte, die generiert werden, am Bildschirm anzuzeigen und den Nachrichtenverkehr zwischen den Transaktionen und Objekten darzustellen. Der BigBrother ist ein Server, mit dem Transaktionen und Objekte Verbindung aufnehmen und Informationen an den BigBrother übermitteln.

Aufruf:

```
java bigbrother.BigBrother [parameters]
```

Kommandozeilenparameter:

- `-p <INT>`: Gibt den Port an, auf dem der BigBrother neue Verbindungen entgegennimmt. Wird dieser Wert nicht gesetzt, wird er aus der Konfigurationsdatei AutO-Demo.config gelesen.

Der BigBrother an sich hat lediglich Anzeigefunktionalität, der Benutzer kann keine interaktiven Handlungen vornehmen.

B.3 Die RequestFactory

Die RequestFactory stellt eine graphische Benutzeroberfläche zur Generierung von Aufträgen zur Verfügung. Außerdem bietet sie die Möglichkeit, den aktuellen Status eines Auftrags abzufragen und auf dem Bildschirm auszugeben. Die RequestFactory baut dabei auf

einem Terminal-Client auf, benötigt also einen Terminal-Server, mit dem sie Verbindung aufnehmen kann.

Aufruf:

```
java [properties] gui.RequestFactory [parameters]
```

Properties:

- `auto-keystore.password`: Hier kann man das Paßwort für den Keystore der RequestFactory angeben. Dies ist nur für Testzwecke gedacht, da das Paßwort im Klartext auf dem Bildschirm erscheint.

Kommandozeilenparameter:

- `-v`: Aktiviert den Verbose-Mode. Dadurch gibt das Programm zusätzliche Informationen aus.
- `-s <STRING>`: Gibt den Rechner an, auf dem der Terminal-Server läuft, zu dem eine Verbindung aufgebaut werden soll.
- `-sl`: Gibt an, daß der Terminal-Server, zu dem eine Verbindung aufgebaut werden soll, auf dem lokalen Rechner läuft.
- `-p <INT>`: Gibt den Port an, auf dem der Terminal-Server zu erreichen ist.

B.4 Der EmployeeDesktop

Der EmployeeDesktop ist die Benutzeroberfläche, die den Mitarbeitern des Support-Centers zur Verfügung steht. Hier können die Aufträge abgefragt, bearbeitet und weitergeleitet werden. Auch der EmployeeDesktop ist ein erweiterter Terminal-Client und benötigt deshalb einen Terminal-Server, zu dem er eine Verbindung aufbauen kann.

Aufruf:

```
java [properties] java gui.EmployeeDesktop [parameters]
```

Properties:

- `auto-keystore.password`: Hier kann das Paßwort für den Keystore des Benutzers des EmployeeDesktops angegeben werden. Dies ist nur für Testzwecke gedacht, da das Paßwort im Klartext auf dem Bildschirm erscheint.

Kommandozeilenparameter:

- `-h` oder `-help`: Gibt einen kurzen Hilfetext zur Bedienung des Programms aus.
- `-n <STRING>` oder `-name <STRING>`: Gibt den Namen des Angestellten an, für den der EmployeeDesktop gestartet wird. Wird dieser Parameter nicht angegeben, wird er aus der Konfigurationsdatei gelesen.
- `-k <FILE>` oder `-keystore <FILE>`: Spezifiziert den Namen der Keystore-Datei, die vom EmployeeDesktop geladen wird, um die für eine Verbindung mit dem Terminal-Server nötigen Schlüssel zu erhalten. Auch dieser Parameter wird aus der Konfigurationsdatei übernommen, falls er nicht explizit angegeben wird.
- `-s <STRING>` oder `-server <STRING>`: Gibt den Rechner an, auf dem der Terminal-Server läuft, zu dem eine Verbindung hergestellt werden soll. Auch dieser Parameter kann in der Konfigurationsdatei festgelegt werden.
- `-sl`: Gibt an, daß der Terminal-Server auf demselben Rechner läuft, auf dem auch der EmployeeDesktop selbst ausgeführt wird.
- `-p <INT>` oder `-port <INT>`: Gibt den Port an, auf dem der Terminal-Server neue Verbindungen entgegennimmt.

Literaturverzeichnis

[AG98] Arnold, K.; Gosling, J.: *The Java Programming Language*. Reading, MA, USA: Addison-Wesley, 1998

[CDF 94] Carey, M. J.; DeWitt, D. J.; Franklin, M. J.; Hall, N. E.; McAuliffe, M. L.; Naughton, J. F.; Schuh, D. T.; Solomon, M. H.; Tan, C. K.; Tsatalos, O. G.; White, S. J.; Zwilling, M. J.: Shoring Up Persistent Applications. **In:** SIGMOD (siehe [SIG94]), S. 383–394

[CFMS95] Castano, S.; Fugini, M. G.; Martella, G.; Samarati, P.: *Database Security*. Reading, MA, USA: Addison-Wesley, 1995 (ACM Press)

[DH76] Diffie, W.; Hellmann, M. E.: New Directions in Cryptography. Erschienen in: *IEEE Transactions on Information Theory* 22 (1976), November, Nr. 6, S. 644–654

[Eic98] Eickler, A.: *Nameservices in Objektbanken*. D-94030 Passau, Universität Passau, Diplomarbeit, Februar 1998

[Gri97] Griesser, S.: *Persistenz und Recovery in AutO, einem verteilten System autonomer Objekte*. D-94030 Passau, Universität Passau, Diplomarbeit, Oktober 1997

[HCF97] Hamilton, G.; Cattell, R.; Fisher, M.: *JDBC database access with Java: A Tutorial and Annotated Reference*. Reading, MA, USA: Addison-Wesley, 1997

[Isl97] Islinger, M.: *Migration in AutO, einem verteilten System autonomer Objekte*. D-94030 Passau, Universität Passau, Diplomarbeit, Oktober 1997

[KE96] Kemper, A.; Eickler, A.: *Datenbanksysteme – Eine Einführung*. R. Oldenbourg Verlag, 1996

[Kei99] Keidl, M.: *Autorisierung und Kryptographie in AutO, einem verteilten System autonomer Objekte*. D-94030 Passau, Universität Passau, Diplomarbeit, Januar 1999

[KIK98] Krivokapić, N.; Islinger, M.; Kemper, A.: Migrating Autonomous Objects in a WAN Environment / Universität Passau. 94030 Passau, Germany: Universität Passau, August 1998 (MIP-9811). – Technical Report

[KKG96] KRIVOKAPIĆ, N.; KEMPER, A.; GUDES, E.: Deadlock Detection Agents: A Distributed Deadlock Detection Scheme / Universität Passau. 94030 Passau, Germany, November 1996 (MIP-9617). – Technical Report

[KLMW94] KEMPER, A.; LOCKEMANN, P. C.; MOERKOTTE, G.; WALTER, H.-D.: Autonomous Objects: A Natural Model for Complex Applications. Erschienen in: *Journal of Intelligent Information Systems (JIIS)* 3 (1994), Nr. 2, S. 133–150

[Kop98] KOPP, Markus: Javas Sandbox-Modell und signierte Applets. Erschienen in: *iX* 6 (1998), S. 130–133

[Kri97] KRIVOKAPIĆ, N.: Synchronization in a Distributed Object System. **In:** DITTRICH, K. R. (Hrsg.); GEPPERT, A. (Hrsg.): *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications (BTW)*. New York, Berlin, etc.: Springer-Verlag, 1997, S. 332–341

[Kri98] KRIVOKAPIĆ, N.: *Control Mechanisms in Distributed Object Bases*. D–94030 Passau, University of Passau, Diplomarbeit, Dec 1998

[KS91] KORTH, H. F.; SILBERSCHATZ, A.: *Database System Concepts*. second. New York, San Francisco, Washington, D.C.: McGraw-Hill, Inc., 1991

[Oak98] OAKS, Scott: *Java Security*. Sebastopol, CA, USA: O'Reilly & Associates, 1998

[Pos98] POSEGGA, Joachim: Die Sicherheitsaspekte von Java. Erschienen in: *Informatik Spektrum* 21 (1998), S. 16–22

[Prö97] PRÖLS, S.: *Implementierung und Simulation von Deadlockerkennungsalgorithmen*. D-94030 Passau, Universität Passau, Diplomarbeit, Oktober 1997

[Rul93] RULAND, Christop: *Informationssicherheit in Datennetzen*. Bergheim: Datacom Verlag, 1993

[Sch96] SCHNEIER, Bruce: *Applied Cryptography, Second Edition*. Chichester, UK: John Wiley & Sons, 1996

[SDBT] SMARKUSKY, D. L.; DEMURJIAN, S. A.; BASTARRICA, M. C.; TING, T. C. Security Capabilities and Potentials of Java

[SIG94] *Proc. of the ACM SIGMOD Conf. on Management of Data* Minneapolis, MI, USA, Mai 1994

[Sun97a] Sun Microsystems. `http://java.sun.com/products/jdk/1.1/docs/guide/security/JavaSecurityOverview.html`: *Java Security API Overview*. 1997

[Sun97b] Sun Microsystems. `http://java.sun.com/products/jdk/1.1/docs/guide/security/CryptoSpec.html`: *JCA API Specification & Reference*. 1997

[Sun98a] Sun Microsystems. `http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html`: *Default Policy Implementation and Policy File Syntax*. 1998

[Sun98b] Sun Microsystems. `http://java.sun.com/security/JCE1.2/early-access/index.html`: *JCE API Specification Version 1.2 (Draft)*. 1998

[TV96] Tardo, J.; Valente, L.: Mobile Agent Security and Telescript. **In:** *In Proc. COMPCON*, 1996, S. 58–63

[U.S85] U.S. Department of Defense. Trusted Computer Security Evaluation Criteria. DOD 5200.28-STD. 1985

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, daß ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle wörtlich oder sinngemäß übernommenen Ausführungen wurden als solche gekennzeichnet. Weiterhin erkläre ich, daß ich diese Arbeit in gleicher oder ähnlicher Form nicht bereits einer anderen Prüfungsbehörde vorgelegt habe.

Passau, den 8. Januar 1999 .....

Stefan Seltzsam