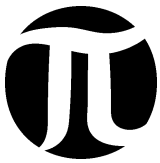
Diplomarbeit

Autorisierung und Kryptographie in AutO, einem verteilten System autonomer Objekte

Markus Keidl
Donau-Schwaben-Str. 7
D–94036 Passau



Lehrstuhl für Dialogorientierte Systeme
Fakultät für Mathematik und Informatik
Universität Passau

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

8. Januar 1999

Zusammenfassung

AutO ist ein verteiltes System autonomer Objekte. Autonome Objekte reagieren auf Ereignisse, indem sie bestimmte Aktionen ausführen. Die Einhaltung der ACID-Eigenschaften gewährleisten ein Transaktionssystem sowie Persistenz und Recovery. Migration sorgt bei Lastverschiebungen für eine dynamische Anpassung des Systems zur Laufzeit. AutO ist vollständig in der Sprache Java programmiert und somit plattformunabhängig. Dadurch ist es insbesondere für heterogene WAN-Netzwerke wie etwa das Internet geeignet.

Sicherheit ist in einem derartigen System von entscheidender Bedeutung. Es kann z. B. nicht davon ausgegangen werden, daß Daten, die über ein WAN übertragen werden, vor Manipulationen geschützt sind. Es müssen also Mechanismen existieren, die die Sicherheit in diesem System gewährleisten.

Im Rahmen dieser Diplomarbeit wurden verschiedene grundlegende Sicherheitskonzepte in AutO integriert. Der Schutz der Daten des Systems wird mit Hilfe kryptographischer Mittel erreicht. Diese werden eingesetzt, wenn Daten über ein möglicherweise unsicheres Netzwerk übertragen werden. Ein Autorisierungssystem, basierend auf rollenbasierter Zugriffskontrolle, stellt sicher, daß nur Benutzer mit der notwendigen Autorisierung auf geschützte Informationen zugreifen können.

Den Abschluß dieser Arbeit bildet die Beschreibung einer Beispielanwendung, die entwickelt wurde, um die Fähigkeiten von AutO zu demonstrieren. Ihre Aufgabe besteht in der Unterstützung der Auftragsannahme und -bearbeitung in einem fiktiven Supportunternehmen. Eine graphische Visualisierungskomponente gewährt dabei Einblick in die internen Abläufe der Anwendung.

Ein großes Dankeschön ...

an Frau Natalija Krivokapić und Herrn Alfons Kemper, für die hervorragende Betreuung dieser Diplomarbeit und dafür, daß sie es immer wieder geschafft haben, mich in Richtung des rechten Weges zurückzuweisen ...

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

an Herrn Stefan Seltzsam, der bei den vielen Diskussionen nie mit kritischen Bemerkungen sparte ...

an meine „Leidensgenossen" im Rechnerraum dafür, daß sie oftmals mehrstündige Screenlocks hinnahmen und unsere AutO-Diskussionen immer ohne Murren ertrugen ...

und schließlich an die tapfere kleine Kaffeemaschine im Rechnerraum, die geduldig mehrere Tassen Kaffee pro Tag spendete, ohne dessen belebende Wirkung mancher Arbeitstag unmöglich gewesen wäre.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Das Internet und andere Netzwerke haben in den letzten Jahren in jeden Sektor der Computerindustrie und auch der meisten anderen Lebensbereiche Einzug gehalten. Daten werden in diesen Netzwerken auch heute noch meistens unverschlüsselt übertragen, d. h. jeder, der die Möglichkeit besitzt, den Datenstrom im Netz abzuhören, erhält Zugriff auf die übermittelten Informationen, egal ob es sich dabei um HTML-Seiten, E-Mails, Dateien oder Kreditkartennummern handelt. Erst seit kurzem setzt sich die Erkenntnis durch, daß es äußerst gefährlich sein kann, Daten ungesichert über ein öffentliches Netzwerk wie das Internet zu übertragen. Deshalb werden immer öfter kryptographische Methoden eingesetzt, um Daten nur demjenigen zugänglich zu machen, für den sie auch bestimmt sind.

Für die Verwaltung von großen Datenmengen wurden Datenbanksysteme entworfen. Sie ermöglichen einen effizienten Zugriff auf die in ihnen gespeicherten Informationen. Mit der stetigen Weiterentwicklung des Internets und der allgemeinen Netzwerktechnik wurden auch diese Konzepte in moderne Datenbanksysteme integriert. Verteilte Datenbanksysteme, bei denen sowohl der Datenbestand als auch anfallende Arbeiten auf verschiedene, durch ein Netzwerk verbundene Rechner verteilt werden, nutzen die Vorteile, die sich durch diese Vernetzung ergeben, soweit möglich aus. Die Gefahr, daß beim Informationsaustausch zwischen verschiedenen Rechnern Daten abgehört oder modifiziert werden, darf nicht vernachlässigt werden. Gerade bei Unternehmen kann ihre wirtschaftliche Existenz von der Geheimhaltung bestimmter Informationen abhängen. Jede Datenbank, die Daten über ein Netzwerk übertragen will, muß auf solche sicherheitsrelevanten Probleme eingehen und entsprechende Lösungen anbieten.

Neben verteilten Datenbanken haben in den letzten Jahren vor allem auch objektorientierte Datenbanksysteme die Entwicklung der Informatik im Bereich Datenbanken bestimmt. Ihr objektorientierter Ansatz bietet unter anderen den Vorteil, daß sich die Modellierung der realen Welt bzw. des darin interessierenden Ausschnitts in einer Datenbank einfacher gestaltet.

AutO (**Aut**onomous **O**bjects) ist ein verteiltes System autonomer Objekte. Es beinhaltet die Vorteile von verteilten als auch von objektorientierten Datenbanksystemen. Die Kommunikation einzelner, auf verschiedenen Rechnern liegender Komponenten geschieht mit Hilfe eines dazwischenliegenden Netzwerks. Die Sicherheit der übermittelten Daten ist, wie schon weiter oben erwähnt, eine zentrale Voraussetzung für ein derartiges Datenbanksystem und muß in AutO dementsprechend berücksichtigt werden. Der nun folgenden Abschnitt geht genauer auf die Sicherheitsaspekte ein, die im Rahmen dieser Diplomarbeit betrachtet wurden.

1.2 Aufgabenstellung

In AutO kommunizieren Systemkomponenten miteinander, die auf unterschiedlichen Rechnern liegen können. Es muß sichergestellt werden, daß die übermittelten Daten geschützt sind, d. h. unautorisierte Dritte dürfen gesendete Daten weder abhören noch modifizieren können. Kryptographische Methoden, wie sie auch im Internet eingesetzt werden, bieten die Möglichkeit, Daten so zu übertragen, daß nur der autorisierte Empfänger diese lesen kann. Außerdem ist es mit Hilfe solcher Methoden möglich, die Integrität der Daten zu überprüfen.

Neben dem Schutz von AutO, oder auch jedes anderen Datenbanksystems, vor Angriffen Dritter von *außen*, gilt es auch, den Zugriff autorisierter Benutzer auf Informationen eines Datenbanksystems zu kontrollieren. Den verschiedenen Mitarbeitern eines Unternehmens sind im allgemeinen nicht die gleichen Informationen zugänglich. Ein normaler Angestellter wird nie das Gehalt seines Kollegen aus einem firmeninternen Datenbanksystem auslesen dürfen. Dem Manager des Unternehmens mag dies durchaus erlaubt sein. Diese Art von Kontrolle über den Zugriff auf Informationen eines Datenbanksystems bezeichnet man als *Autorisierung*. Die Implementierung eines geeigneten Autorisierungssystems war, neben der Nutzung von kryptographischen Methoden zur Sicherung von Netzwerkverbindungen, die zweite Aufgabe dieser Diplomarbeit.

1.3 Überblick

Das nun folgende Kapitel beschreibt das AutO-System, so wie es sich vor Integration der in dieser Diplomarbeit beschriebenen Konzepte darstellte. Besonders wird dabei auf diejenigen Komponenten eingegangen, die für diese Diplomarbeit von Belang sind.

In Kapitel 3 werden verschiedene kryptographische Grundlagen erläutert, die für das Verständnis der Diplomarbeit wichtig sind.

Daran anschließend wird in Kapitel 4 beschrieben, wie die eingeführten kryptographischen Verfahren in AutO eingesetzt werden, um Daten vor dem unautorisierten Zugriff Dritter zu schützen.

Das Autorisierungssystem von AutO, basierend auf rollenbasierter Zugriffskontrolle, stellt Kapitel 5 vor. Es gewährleistet, daß Benutzer nur auf Informationen zugreifen können,

für die sie die notwendige Autorisation besitzen.

Eine Beispielanwendung, die die Fähigkeiten von AutO inklusive der im Rahmen dieser Diplomarbeiten eingebrachten Mechanismen veranschaulicht, wird in Kapitel 6 beschrieben.

Den Abschluß bilden in Kapitel 7 eine Zusammenfassung und ein Ausblick auf zusätzliche Sicherheitskonzepte, die in AutO integriert werden könnten.

Im Anhang wird zuerst eine Reihe von kryptographischen Algorithmen beschrieben. Zusätzlich werden Messungen vorgestellt, die veranschaulichen, inwieweit der Einsatz von Kryptographie die Leistungsfähigkeit von AutO beeinflußt. Eine Beschreibung von verschiedenen Werkzeugen zur Verwaltung der implementierten Komponenten sowie eine Auflistung der neu hinzugekommenen Konfigurationsmöglichkeiten von AutO schließen den Anhang ab.

Kapitel 2

Das AutO-System

AutO basiert auf dem in [KLMW94] vorgestellten Modell autonomer Objekte. Darauf aufbauend wurde es zu einem verteilten, persistenten und objektorientierten Datenbanksystem weiterentwickelt, das auch Recovery und Migration bietet. Der nun folgende Abschnitt stellt das AutO-System so vor, wie es sich zu Beginn dieser Diplomarbeit präsentierte. Besonders hervorgehoben werden diejenigen Teile des Systems, die für die Implementierung der in dieser Arbeit vorgestellten Sicherheitskonzepte von Belang waren, etwa weil sie direkt von Änderungen betroffen waren oder einen wesentlichen Einfluß auf das Design der implementierten Konzepte besaßen.

Zuerst wird das Modell autonomer Objekte vorgestellt, das die Grundlage von AutO bildet. Daran anschließend wird genauer auf die verteilte Architektur des Systems eingegangen. Zuletzt werden autonome Objekte und Transaktionen beschrieben, mit denen Anwendungen in AutO entwickelt werden können.

2.1 Ein Modell autonomer Objekte

In der realen Welt werden Aktivitäten durch verschiedene Ereignisse ausgelöst. Sie können durch äußere Umstände, durch das Erreichen eines bestimmten Zeitpunkts oder durch eine interne Zustandsänderung angestoßen werden. Aus Datenbanksicht hat dies zur Entwicklung von *aktiven* Datenbanken geführt, die es ermöglichen, solche Vorgänge in eine Datenbank zu integrieren. Leider läßt sich dieser Ansatz nur schwer mit objektorientierten Methoden, wie sie in objektorientierten Datenbanken angewandt werden, verbinden.

In [KLMW94] wird deshalb eine Erweiterung vorgeschlagen. *Autonome Objekte* werden mit aktivem Verhalten ausgestattet, so daß sie auf Ereignisse reagieren können. Diese Objekte sind dabei „normale“ Objekte gemäß dem herkömmlichen, objektorientierten Verständnis, d. h. sie besitzen sowohl eine interne *Struktur*, repräsentiert durch ihre Daten, als auch ein *Verhalten*, widergespiegelt durch ihre Operationen, die sie zur Verfügung stellen, um auf ihre Struktur zuzugreifen und sie zu manipulieren. Weiterhin existiert eine Klassifizierung der Objekte in einzelne Typen, die miteinander durch eine Ableitungshierarchie verbunden sind.

Jedes autonome Objekt besitzt weiterhin *aktives Verhalten*, d. h. es hat die Möglichkeit, auf verschiedene Ereignisse zu reagieren:

- auf Nachrichten, die von anderen Quellen an das Objekt gesendet werden,
- auf das Erreichen eines vorbestimmten Zeitpunkts und
- auf das Erkennen eines ausgezeichneten Zustands oder eines Zustandsübergangs.

Der Wahrnehmungsmechanismus für ein solches Ereignis ist in jedem Objekt vorhanden und wird als *Guard* bezeichnet. Ein Guard besteht unter anderem aus einer Bedingung, die als erstes ausgewertet wird, und einer Aktion, die ausgeführt wird, falls die Bedingungsprüfung erfolgreich verläuft.

Objektautonomie bedeutet schließlich, daß jedes Objekt selbst bestimmt, welche Ereignisse es in welcher Reihenfolge abarbeitet. Der dahinterstehende Algorithmus wird auch als *Ausführungsmodell* des Objektes bezeichnet.

Zusammenfassend läßt sich also sagen, daß autonome Objekte Objekte im herkömmlichen Sinne sind, die zusätzlich ein aktives Verhalten und eine Menge von Guards besitzen, die auf Ereignisse entsprechend eines festgelegten Ausführungsmodells reagieren.

2.1.1 Kommunikation zwischen autonomen Objekten

Die Autonomie der einzelnen Objekte erfordert es, daß man keinen einfachen Prozedurenaufruf verwenden kann, um mit einem solchen Objekt zu kommunizieren. Statt dessen geschieht dies durch den Austausch von Nachrichten. Neben den entsprechenden Mechanismen, um Nachrichten abzuschicken und zu empfangen, benötigt ein autonomes Objekt zusätzlich einen Puffer, in dem eingehende Nachrichten gespeichert werden können. Nachrichten selbst beinhalten den Namen des Guards, für den sie bestimmt sind, und eine Menge von Parametern, die dem Guard übergeben werden.

Der Nachrichtenaustausch, wie er in [KLMW94] vorgestellt wird, kann sowohl synchron als auch asynchron erfolgen.

2.1.2 Die Behavioral Map eines autonomen Objekts

Autonome Objekte werden in [KLMW94] mit Hilfe einer sogenannten *Behavioral Map* festgelegt. Sie enthält neben der Struktur und dem Verhalten der Objekte auch deren Guards. Es stehen drei Arten von Guards zur Verfügung:

- **On-Guards**: Sie starten eine Aktion als Folge einer von einer Transaktion oder einem anderen autonomen Objekt erhaltenen Nachricht. Die Aktion wird allerdings nur ausgeführt, wenn der Guard passierbar ist. Das bedeutet:
  - Im Nachrichtenpuffer des autonomen Objekts ist eine Nachricht enthalten, die diesen Guard auslösen kann.

  - Die Bedingung des Guards liefert bei ihrer Auswertung wahr zurück. Dabei werden bei der Überprüfung der Bedingung die Parameter der Nachricht beachtet.

- **At-Guards**: Für At-Guards gilt dasselbe wie für On-Guards, allerdings mit der Ausnahme, daß bei ihnen nicht das Vorhandensein einer Nachricht im Puffer des Objekts überprüft wird, sondern das Erreichen bzw. Überschreiten eines bestimmten Zeitpunkts.

- **If-Guards**: If-Guards sind passierbar, wenn ihre Bedingung wahr zurückliefert. Es ist kein Ereignis von außen nötig, um sie auszulösen. Sie reagieren vielmehr auf eine Änderung des Objektzustands.

Ein Beispiel für eine Behavioral Map ist in Abbildung 2.1 zu sehen. Sie zeigt die (vereinfachte) Modellierung eines Typs `Warehouse`, also eines Lagerhauses, das Artikel verwaltet. Zusätzlich enthält sie einen Verweis auf einen Lieferanten, bei dem knapp gewordene Artikel nachbestellt werden können.

persistent autonomous type Warehouse **is**
  body
   int quantity [10];
   Oid supplier;
   ...
  operations ...
  behavioral map
   on setSupplier(s) **priority** 5 **do self**.setSupplier(s);
   on consume(item$_i$, q) **if** $q \leq 20$ **and** $q \leq$ **self**.quantity [item$_i$] **priority** 9
      do self.consume(item$_i$, q);
   on consume(item$_i$, q) **if** $q \leq$ **self**.quantity [item$_i$] **priority** 1
      do self.consume(item$_i$, q);
   on supply(item$_i$, q) **priority** 90 **do self**.supply(item$_i$, q);
   on quantity(item$_i$) **do return self**.quantity(item$_i$);
   at 12/31/1997 every 1 year **do self**.inventory();
   if self.quantity [item$_i$] ≤ 5 **priority** 5 **do self**.order(item$_i$, 20);
  implementation ...
end type Warehouse;

Abbildung 2.1: Behavioral Map des `Warehouse`–Beispiels [Gri97]

Der Typ `Warehouse` stellt verschiedene Operationen zur Verfügung. Mit `supply` kann die verfügbare Menge eines Artikels vergrößert, mit `consume` entsprechend verringert werden, falls die Menge an zur Verfügung stehenden Artikeln noch ausreichend ist. `quantity` gibt die vorhandene Zahl von Einheiten eines bestimmten Artikels zurück und `inventory` liefert den Bestand aller Artikel. Mit `setSupplier` schließlich ist es möglich, den Lieferanten zu setzen.

Die Operationen `supply`, `consume`, `quantity` und `setSupplier` werden von den gleichnamigen On-Guards aufgerufen, die selbst wiederum durch das Eintreffen einer entsprechenden Nachricht ausgelöst werden. Dadurch, daß für `consume` zwei Guards mit unterschiedlichen Prioritäten (siehe dazu auch Abschnitt 2.1.3) definiert wurden, werden Anfragen für kleinere Quantitäten bevorzugt bearbeitet. Der At-Guard `inventory` löst einmal pro Jahr eine Inventur aus. Der einzige definierte If-Guard sorgt dafür, daß Einheiten eines Artikels nachbestellt werden, wenn die vorhandene Zahl unter einen bestimmten Schwellwert fällt.

2.1.3 Ausführungsmodell

Das in [KLMW94] vorgestellte Modell autonomer Objekte sieht ein Ausführungsmodell vor, in dem Objekte nicht-deterministisch einen Guard bestimmen, für den dann überprüft wird, ob er passierbar ist. Dabei werden die Prioritäten einzelner Guards berücksichtigt, d. h. je höher die Priorität eines Guards ist, desto höher ist die Wahrscheinlichkeit, daß er ausgewählt wird.

Ist ein Guard ausgewählt, wird zuerst seine Passierbarkeit, wie unter 2.1.2 beschrieben, überprüft. Ist diese Überprüfung erfolgreich, wird die zum Guard gehörende Aktion ausgeführt. Nachdem diese beendet wurde, wird ein neuer Guard ausgewählt und der eben dargestellte Prozeß beginnt von vorne.

Autonome Objekte unterstützen keinen Intra-Objekt-Parallelismus, d. h. ein Objekt ist zu jedem Zeitpunkt nur mit der Abarbeitung maximal einer Aktion beschäftigt. Es können nicht mehrere Aktionen und damit auch Guards zur gleichen Zeit bearbeitet werden. Es ist auch nicht möglich, die Abarbeitung einer Aktion zu unterbrechen und später wieder mit ihr fortzufahren. Die Abarbeitung einer Aktion ist also atomar.

2.2 Die Systemarchitektur von AutO

AutO ist ein verteiltes System, das entsprechend des Modells in [KLMW94] und im Hinblick auf den Einsatz in einem Wide-Area-Network (WAN) konzipiert wurde. Die Struktur eines solchen Netzwerks ist selten homogen, deshalb wurde bei der Entwicklung auf Plattformunabhängigkeit Wert gelegt, und AutO vollständig in Java [AG98] implementiert. Abbildung 2.2 zeigt vereinfacht den Aufbau eines AutO-Systems, bestehend aus mehreren vernetzten Rechnern. Jeder dieser Rechner, auch *(AutO-)Node* genannt, zeichnet sich durch zwei Arten von Prozessen aus, welche auf ihm ausgeführt werden:

- je einem *Node-Manager*, dessen Hauptaufgabe die Verbindung lokaler AutO-Komponenten mit Teilsystemen auf anderen Rechnern ist. Er leitet unter anderem Nachrichten von lokalen Objekten an andere Node-Manager weiter oder nimmt von solchen Nachrichten entgegen, um sie dann den lokalen Empfängern zuzustellen.
- einem oder mehreren *AutO-Prozessen*, die für die Verwaltung der lokalen autonomen Objekte zuständig sind.

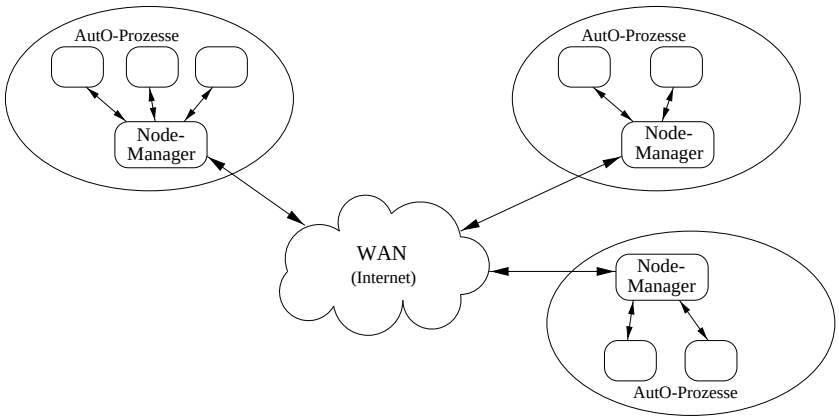


Abbildung 2.2: Die Architektur des AutO-Systems

Zur Kommunikation zwischen verschiedenen Node-Managern über das sie verbindende Netzwerk oder auch zwischen verschiedenen Prozessen desselben Rechners werden gewöhnliche Java-Sockets verwendet, die das TCP/IP-Protokoll benutzen. Über eine solche Verbindung werden dann Nachrichten[1] übermittelt.

Der Node-Manager

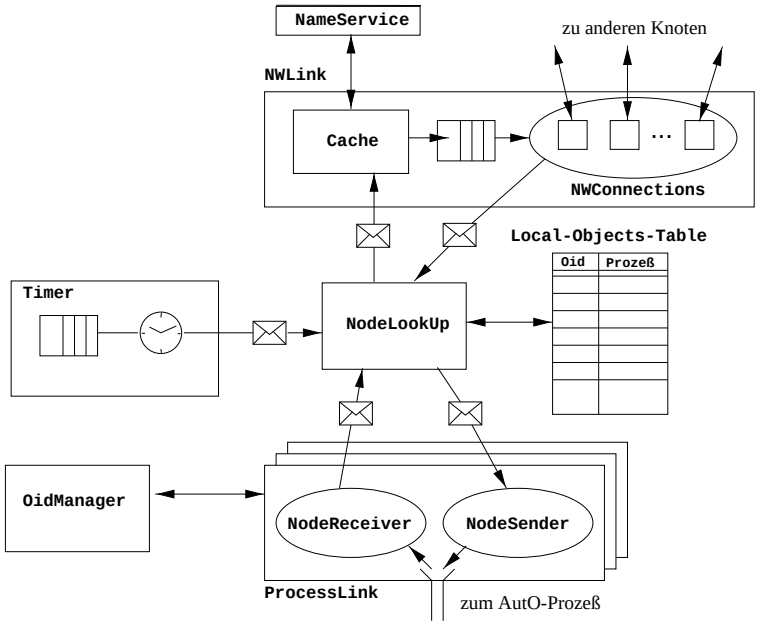


Abbildung 2.3: Die Architektur des AutO-Node-Managers [Gri97]

[1]Nachrichten sind in AutO immer Instanzen von Klassen, die von der AutO-Klasse `common.messages.Message` abgeleitet wurden.

Die primäre Aufgabe eines Node-Managers besteht in der Weiterleitung von Nachrichten. Er nimmt Nachrichten von lokalen AutO-Prozessen entgegen und leitet sie an ihren Empfänger weiter. Ebenso verteilt er vom Netzwerk eingehende Nachrichten an diejenigen Prozesse, die die jeweiligen Empfänger beinhalten.

Nachrichtenweiterleitung: Den genauen Aufbau eines Node-Manager-Prozesses zeigt Abbildung 2.3. Jeder Node-Manager besitzt permanente Verbindungen zu allen lokalen AutO-Prozessen. Eine derartige Verbindung wird mit Hilfe einer Instanz der Klasse `ProcessLink` realisiert. Sie enthält zwei Threads, den `NodeSender` und den `NodeReceiver`. Der `NodeSender` leitet Nachrichten aus dem zum `ProcessLink` gehörenden AutO-Prozeß an den `NodeLookUp` weiter, der `NodeReceiver` nimmt Nachrichten von diesem entgegen und gibt sie an den AutO-Prozeß weiter.

Die Verbindung eines Node-Managers zu anderen Node-Managern wird mit Hilfe der Klasse `NWLink` erreicht. Nachdem ein `NWLink` eine Nachricht entgegengenommen hat, bestimmt er den Rechner, auf dem der Empfänger der Nachricht zu finden ist. Dazu befragt er zuerst einen lokalen Cache und dann, falls in diesem keine entsprechenden Informationen gefunden werden, den *Name-Service* von AutO.

Die Entscheidung, wohin eine Nachricht gesendet werden muß, trifft die Klasse `NodeLookUp`. Sie besitzt eine Tabelle *Local-Objects-Table*, die alle lokalen Objekte, die Nachrichten empfangen können, und den sie enthaltenden AutO-Prozeß enthält. Trifft eine Nachricht über den `NWLink` ein, stellt der `NodeLookUp` denjenigen Prozeß fest, der den Empfänger der Nachricht enthält und leitet die Nachricht an den zugehörigen `ProcessLink` weiter. Erhält andererseits der `NodeLookUp` eine Nachricht von einem `ProcessLink`, so stellt er zuerst mit Hilfe der Local-Objects-Table fest, ob der Empfänger in einem lokalen AutO-Prozeß liegt. Ist dies der Fall, wird die Nachricht an den entsprechenden `ProcessLink` weitergeleitet. Ansonsten wird sie dem `NWLink` übergeben, der sie dann über das Netzwerk an den Empfänger weiterleitet.

Der Timer: Der *Timer* eines Node-Managers verwaltet die At-Guard-Ereignisse aller lokalen autonomen Objekte. Einem normalen Objekt wäre dies nicht möglich, da es etwa in der Datenbank abgelegt sein und damit das Erreichen bzw. Verstreichen eines bestimmten Zeitpunkts nicht feststellen kann [Gri97].

Deshalb registriert ein autonomes Objekt jeden seiner At-Guards bei dem Timer des lokalen Node-Managers. Dazu wird dem Timer neben einem Objekt der Klasse `TimerEvent`, das alle notwendigen Informationen über das Zeitereignis enthält, auch eine Nachricht übergeben, die beim Erreichen des im `TimerEvent` angegebenen Zeitpunkts an das Objekt zurückgesendet werden soll. Auch periodisch wiederkehrende Zeitpunkte können mit Hilfe eines `TimerEvent` festgelegt werden.

Der AutO-Prozeß

Java-Prozesse, die Objekte des AutO-Systems enthalten, werden *AutO-Prozesse* genannt. Jeder AutO-Prozeß unterhält eine konfigurierbare Anzahl von Verbindungen zu einem

Datenbanksystem (DBMS), in dem persistent alle Daten des Prozesses, darunter fallen auch alle autonomen Objekte, abgelegt werden.

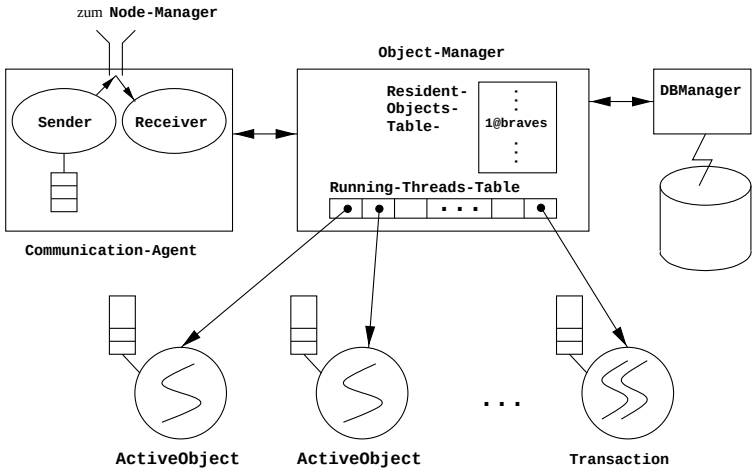


Abbildung 2.4: Die Architektur eines AutO-Prozesses [Gri97]

Die wesentlichen Aufgaben eines AutO-Prozesses bestehen in der Verwaltung von Objekten – dies übernimmt der *Object-Manager* – und in der Weiterleitung von Nachrichten – dies übernimmt der *Communication-Agent*. Den schematischen Aufbau eines AutO-Prozesses mit seinen wichtigsten Komponenten zeigt Abbildung 2.4.

Der Communication-Agent: Der Communication-Agent ist für die Zustellung und Weiterleitung von Nachrichten innerhalb eines AutO-Prozesses zuständig. Dazu unterhält er eine permanente Verbindung zum lokalen Node-Manager. Über diese werden ausgehende Nachrichten gesendet, falls der Empfänger der Nachricht nicht im lokalen Prozeß gefunden wird. Um dies festzustellen wird der Object-Manager befragt, der alle Objekte des Prozesses kennt. Eingehende Nachrichten werden über den Object-Manager an ihre Empfänger weitergeleitet.

Der Object-Manager: Diese Komponente verwaltet alle Objekte eines AutO-Prozesses in einer *Resident-Objects-Table* genannten Tabelle und ebenso alle Threads des Prozesses in der Tabelle *Running-Threads-Table*. Der Object-Manager ist dafür zuständig, Objekte zu laden, zu erzeugen und zu löschen. Außerdem stellt er Nachrichten, die er vom Communication-Agent erhält, an ihre Empfänger zu.

Das Terminal: Ein besonderer AutO-Prozeß ist der *Terminal-Prozeß*. In einem solchen kann ein Benutzer interaktiv Transaktionen starten, Objekte erzeugen oder löschen, Nachrichten senden, etc. Einem Benutzer stehen zwei Möglichkeiten zur Verfügung, die Funktionalität eines Terminal-Prozesses zu nutzen:

- Der Benutzer startet einen eigenen Terminal-Prozeß lokal auf seinem Rechner. Da es sich bei diesem Prozeß um einen normalen AutO-Prozeß handelt, muß auf dem Rechner außerdem ein Node-Manager-Prozeß vorhanden sein, wenn aus dem Terminal-Prozeß heraus nicht-lokale AutO-Komponenten angesprochen werden sollen.

- Es besteht auch die Möglichkeit, sich mit einem bestehenden Terminal-Prozeß auf einem beliebigen Rechner zu verbinden. Dazu muß der Terminal-Prozeß einen *Terminal-Server*, implementiert in der Klasse `shell.clientserver.TerminalServer`, enthalten. Existiert ein solcher, kann man eine Socket-Verbindung zu diesem öffnen und damit die Funktionalität des Terminal-Prozesses nutzen.

Zum Starten eines eigenen Terminal-Prozesses und auch zum Verbinden mit einem Terminal-Server muß das Java-Programm `shell.Terminal` benutzt werden. Spezifiziert man beim Start einen Server durch dessen Adresse (und Portnummer), verbindet sich das Programm mit dem angegebenen Server über eine Socket-Verbindung. Anderenfalls wird ein lokaler Terminal-Prozeß gestartet.

Die Bedienung eines Terminals, d. h. eines lokalen Terminal-Prozesses oder eines Terminal-Servers, erfolgt mit Hilfe einer graphischen Oberfläche. Eine Online-Hilfe erklärt die wichtigsten Befehle. Im folgenden wird die Oberfläche ebenfalls als *Terminal* bezeichnet, wobei es unwichtig ist, ob dabei ein lokaler Terminal-Prozeß gestartet wurde oder ob die Verbindung zu einem Terminal-Prozeß über einen Terminal-Server hergestellt wird.

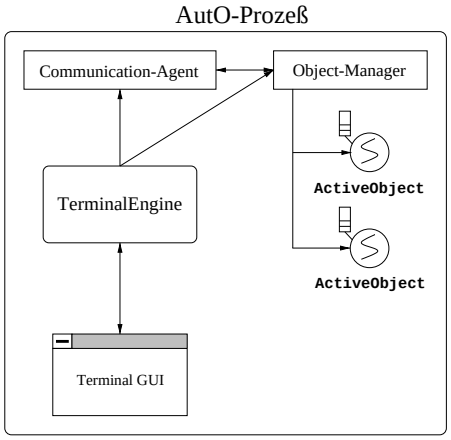


Abbildung 2.5: Der Aufbau eines Terminal-Prozesses ohne Terminal-Server

Die Abbildungen 2.5 und 2.6 zeigen den Aufbau eines Terminal-Prozesses mit und ohne Terminal-Server. Die Klasse `TerminalEngine` ist in beiden Fällen dafür zuständig, interaktive Kommandos, die von einem Benutzer eingegeben werden, um etwa eine Transaktion zu starten oder ein autonomes Objekt zu erzeugen, zu verarbeiten und die damit verbundenen Aktionen auszuführen. Erfolgt die Verbindung zu einem Terminal-Prozeß über einen Terminal-Server, dann ist auf der Client-Seite der *Terminal-Client* für die Verwaltung der Verbindung zuständig.

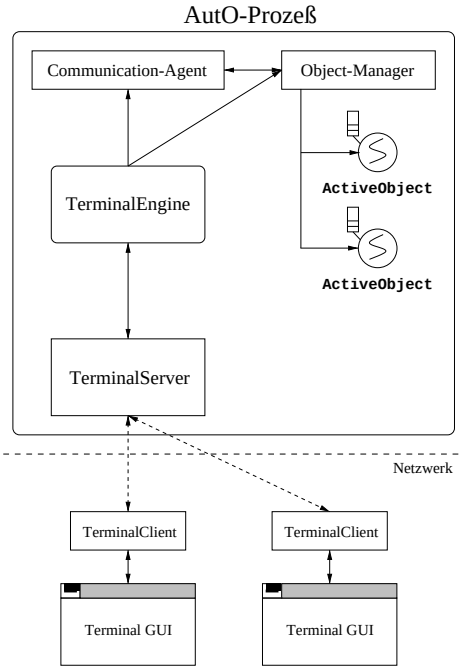


Abbildung 2.6: Der Aufbau eines Terminal-Prozesses mit Terminal-Server

2.3 Autonome Objekte

Autonome Objekte in AutO entsprechen denjenigen, die in [KLMW94] vorgestellt wurden. Die Klasse `ActiveObject` realisiert die Basisfunktionalität, die ein autonomes Objekt benötigt. Darunter fallen das Senden und Empfangen von Nachrichten, das Ausführungsmodell für Guards und weitere Funktionen, die für Migration, Persistenz und Recovery notwendig sind [Gri97, Isl97]. Benutzerdefinierte Daten und Operationen werden in einer Java-Klasse definiert, die von `UserObject` abgeleitet ist. Ferner existiert für jeden Typ von autonomen Objekten eine Typinformation, die die Daten der Behavioral Map enthält, die allen Objekten des gleichen Typs gemeinsam sind.

Jedes autonome Objekt wird eindeutig durch eine logische OID identifiziert. Diese enthält neben der Adresse des Rechners, auf dem das Objekt erzeugt wurde, außerdem eine auf dem erzeugenden Rechner eindeutige Seriennummer.

Ein autonomes Objekt wird aktiv, d. h. es bekommt einen Thread zugeteilt, wenn ein Ereignis eintritt, das einen Guard auslösen kann. Dies kann geschehen, wenn eine Nachricht an das Objekt zugestellt wurde, oder sich der Zustand des Objekts geändert hat. Allerdings wird auch eine Zustandsänderung initial durch eine Nachricht ausgelöst. Sie ist aber trotzdem ein eigenes Ereignis.

Ein autonomes Objekt bleibt solange aktiv, wie noch unbearbeitete Ereignisse vorhanden sind, die einen Guard auslösen können. Erst wenn alle diese Ereignisse verarbeitet wurden, gibt das autonome Objekt seinen Thread ab.

Sperrvergabe: Um Isolation und Serialisierbarkeit von Transaktionen zu erreichen, verwendet AutO semantisches Locking. Dabei wird die Semantik von Aktionen, die durch das Auslösen von Guards abgearbeitet werden, ausgenutzt [Kor83, SS84]. Die Vergabe von Sperren geschieht nach dem strengen Zwei-Phasen-Sperrprotokoll [Kri97].

Dabei ist es nötig, vor dem Ausführen einer Operation, also einer Aktion, die mit einem Guard verbunden ist, die dafür notwendigen Sperren zu erwerben. Welche Sperren benötigt werden, hängt von der Semantik der Operation ab. Eine Sperre repräsentiert in AutO ein bestimmtes Zugriffsmuster auf die Daten eines autonomen Objekts. Je nachdem, auf welche Daten eine Operation zugreift, muß sie entsprechende Sperren anfordern.

Um zu überprüfen, ob zwei verschiedene Sperren kompatibel sind, d. h. die damit verbundenen Änderungen am Objekt können gleichzeitig durchgeführt werden, muß bestätigt werden, daß die Änderungen kommutativ sind, d. h. der resultierende Zustand des Objekts, auf dem die Modifikationen durchgeführt werden, ist unabhängig von der Reihenfolge der Änderungen. In AutO wird eine Kompatibilitätsmatrix benutzt, um festzulegen, welche Sperren kompatibel sind und welche nicht. Genauere Informationen zur Sperrverwaltung in autonomen Objekten geben [Isl97, Gri97].

Guardabarbeitung: In AutO wird ein Guard als aktiv bezeichnet, wenn ein Ereignis vorliegt, das geeignet ist, diesen Guard auszulösen. Für On- und At-Guards bedeutet dies das Vorhandensein einer passenden Nachricht. Bei If-Guards reicht eine Änderung des Objektzustands aus, um ihn aktiv werden zu lassen.

Ist in einem autonomen Objekt mindestens ein Guard aktiv, so wird nicht-deterministisch einer der aktiven Guards ausgewählt. Dabei ist die Wahrscheinlichkeit für die Auswahl eines bestimmten Guards direkt proportional zu seiner Priorität. Anschließend werden die Latches[2] angefordert, die für die Prüfung der Guardbedingung notwendig sind. Nachdem diese gewährt wurden, kann die Bedingung überprüft werden. Liefert diese Überprüfung den Wert falsch zurück, werden die Latches zurückgegeben und der Guard deaktiviert. Ansonsten werden sie in Sperren konvertiert und die Sperren für die Guard-Aktion angefordert. Wurden auch diese gewährt, kann die Aktion ausgeführt werden. Abschließend wird ein neuer Guard aus der Menge der aktiven Guards zur Abarbeitung ausgewählt [Gri97, Isl97].

2.4 Transaktionen

Transaktionen in AutO sind Instanzen von Java-Klassen, die von der Klasse `Transaction` abgeleitet sein müssen. Sie führen Code aus und greifen auf autonome Objekte zu, indem sie ihnen Nachrichten senden und dann auf eine Antwort warten. Solange keine Antwort eingetroffen ist, ist die Transaktion also blockiert. Das Transaktionssystem von AutO erfüllt die ACID-Eigenschaften, wie sie in [KE96] beschrieben werden.

[2]Latches sind eine besondere Art von Sperren. Sie werden wegen des strengen Zwei-Phasen-Sperrprotokolls benötigt.

Identifiziert wird eine Transaktion eindeutig durch eine `TID`. Sie enthält den Namen des Rechners, auf dem sie erzeugt wurde, eine auf dem erzeugenden Rechner eindeutige Seriennummer sowie einen Zeitstempel, der angibt, wann sie erzeugt worden ist.

Commit und Abort: Das Commit bzw. Abort einer Transaktion in AutO wird nach dem Zwei-Phasen-Commit-Protokoll [KS91, KE96] abgewickelt. Eine Transaktion sendet dabei an jedes autonome Objekt, das im Laufe ihrer Abarbeitung eine Aktion für diese Transaktion ausgeführt hat, eine entsprechende Nachricht. Das autonome Objekt führt das Commit oder Abort dann lokal durch und sendet der Transaktion eine bestätigende Antwort zurück.

Light-Weight-Transaktionen: If- und auch At-Guards werden nicht direkt durch die Nachricht einer Transaktion ausgelöst. Sie werden durch eine Zustandsänderung des autonomen Objekts oder durch das Erreichen eines bestimmten Zeitpunkts ausgelöst. Allerdings müssen auch die von ihnen ausgeführten Aktionen in das Transaktionssystem von AutO eingebunden werden, um sie etwa im Bedarfsfalle auch wieder rückgängig machen zu können.

Die Erzeugung einer normalen Transaktion wäre zu aufwendig, da If- bzw. At-Guards im allgemeinen nur lokale Aktionen ausführen. Deshalb gibt es in AutO *Light-Weight-Transaktionen*. Eine derartige Transaktion wird erzeugt, wenn bei der Ausführung einer Aktion eines If- oder At-Guards eine Abhängigkeit zu einer anderen Transaktion entsteht oder einem anderen autonomen Objekt eine Nachricht gesendet wird. Am Ende der Guard-Aktion führt diese Light-Weight-Transaktion dann ein normales Commit oder Abort aus. Werden in der Aktion des If- oder At-Guards nur lokale Änderungen durchgeführt, so wird keine Light-Weight-Transaktion erzeugt, sondern am Ende der Aktion ein lokales Commit oder Abort ausgeführt.

Verklemmungen: Bei AutO wird, wie schon in Abschnitt 2.3 angeführt, ein sperrbasiertes Protokoll verwendet, um verschiedene Transaktionen voneinander zu isolieren (ein wesentlicher Punkt der ACID-Eigenschaften). Dadurch kann es vorkommen, daß zwischen mehreren Transaktionen Abhängigkeiten entstehen, sogenannte *Verklemmungen*. Diese führen dazu, daß jede an der Verklemmung beteiligte Transaktion auf eine andere aus diesem Kreis wartet. Um solche Verklemmungen zu erkennen und auch aufzulösen, verwendet AutO *Deadlock Detection Agents*, kurz DDAs [KKG96, Prö97].

Wird in der Sperrverwaltung eines autonomen Objekts eine Abhängigkeit zwischen zwei Transaktionen festgestellt, wird diese an einen DDA weitergemeldet. Der DDA baut aus diesen Abhängigkeiten einen Wartegraphen auf und versucht, darin Zyklen zu finden. Wird ein solcher Zyklus gefunden, liegt eine Verklemmung vor. Diese wird durch den DDA aufgelöst, indem er eine oder auch mehrere an der Verklemmung beteiligte Transaktionen abbricht [Kri97, Gri97, Isl97].

2.5 Persistenz und Recovery

Die Lebensdauer von autonomen Objekten ist in AutO nicht auf die Existenz eines Prozesses beschränkt. Durch die Anbindung eines Datenbanksystems an AutO sind Objekte auch nach dem Beenden eines Prozesses vorhanden und können später aus dem Datenbanksystem wieder geladen werden. Bei diesem kann es sich um jedes Datenbanksystem handeln, das eine JDBC-Schnittstelle [HCF97] bereitstellt. Es wurde aber auch eine Anbindung an SHORE [CDF 94], eine Bibliothek zur Entwicklung von objektorientierten Datenbanksystemen, entwickelt, die alternativ verwendet werden kann.

Jeder AutO-Prozeß kann an ein eigenes DBMS gekoppelt sein, es können sich aber auch mehrere Prozesse, die auf verschiedenen Rechnern laufen können, ein gemeinsames Datenbanksystem teilen. Aus Effizienzgründen sollte mindestens eine Datenbank pro LAN (Local Area Network) existieren.

In einem Transaktionssystem wie AutO können verschiedene Fehler auftreten. *Recovery* sorgt in einem solchen Fall dafür, daß bei einem Neustart wieder ein transaktionskonsistenter Zustand hergestellt wird. Weiterführende Informationen zu Persistenz und Recovery in AutO können [Gri97] entnommen werden.

2.6 Migration

Migration, so wie sie in AutO und auch in dieser Arbeit verstanden wird, bedeutet den Wechsel des Aufenthaltsortes, also des Rechners, eines autonomen Objekts. Sie ist vollständig in das autonome Objektmodell integriert, d. h. Objekte entscheiden selbst, wann und wohin sie migrieren wollen. Allerdings besteht auch die Möglichkeit, Objekte explizit durch Aufrufen eines speziellen On-Guards zur Migration zu zwingen. Das Ziel dieser Art von Migration ist eine Verbesserung der Performance des gesamten Systems durch die Senkung der Kommunikationskosten.

Ein autonomes Objekt sammelt Statistiken über die Zugriffe von verschiedenen Rechnern und Netzwerken. Es merkt sich, wann und wie oft diese zugegriffen haben. Eine Migrationsstrategie, die für jedes Objekt individuell gestaltet und auch während der Laufzeit ausgetauscht werden kann, bestimmt dann regelmäßig mit Hilfe der gesammelten Statistiken, ob und wohin das Objekt migrieren soll.

Eine ausführlichere Beschreibung der Migrationskomponente in AutO sowie eine Bewertung verschiedener Strategien im Hinblick auf eine Senkung der Kommunikationskosten und einer Steigerung des Transaktionsdurchsatzes ist in [KIK98, Isl97] zu finden. Auf die verschiedenen Aspekte der Sicherheit in der Migration geht [Sel99] ein.

Kapitel 3

Eine Einführung in Kryptographie

Kryptographie ist die Wissenschaft von den Methoden der Ver- und Entschlüsselung von Daten mit dem Ziel, Originaldaten mathematisch so zu transformieren, daß es einzig und allein dem Empfänger möglich ist, aus den transformierten die ursprünglichen Daten zu berechnen [Rul93].

Lange Zeit war wirklich starke Kryptographie[1] ausschließlich militärischen Kreisen und ihnen angeschlossenen Agenturen zugänglich. Sie wendeten Milliarden Dollar dazu auf, ihre eigene Kommunikation immer sicherer zu machen und die anderer Parteien zu brechen. Privatpersonen oder auch zivile Unternehmen hatten weder das Wissen noch die Mittel, Daten vor diesen Institutionen zu schützen [Sch96].

Während der letzten 20 Jahre nahm die öffentliche akademische Forschung auf dem Gebiet der Kryptographie einen rasanten Aufschwung. *State-of-the-art*-Kryptographie ist nun auch außerhalb des Militärs zu finden. Damit ist es einer Firma möglich, kritische Daten, die Schlüsselbereiche des Unternehmens betreffen, sicher über ein Netzwerk an Filialen zu übertragen.

Kryptographie dient aber nicht nur dazu, Informationen geheim zu halten. Vielmehr ist es damit auch möglich, die Unversehrtheit von Daten zu erkennen sowie Kommunikationspartner und den Ursprung von Daten zu authentifizieren. Sie kann also überall dort verwendet werden, wo die Sicherung von Kommunikationsverbindungen nötig ist. Die nun folgende, kurze Einführung in Kryptographie soll helfen, die anschließenden Abschnitte dieser Diplomarbeit zu verstehen.

3.1 Grundlagen und Begriffe

Verschlüsselung und Entschlüsselung sind die entscheidenden Begriffe in der Kryptographie.

[1] Kryptographie wird als *stark* bezeichnet, wenn das Brechen der Verschlüsselung, d. h. das Entschlüsseln ohne die dazu erforderlichen Informationen wie etwa einem passenden Schlüssel, nur mit sehr großem Aufwand möglich ist.

- *Verschlüsselung* bezeichnet den Vorgang der mathematischen Transformation der ursprünglichen in die transformierten Daten. Die Originaldaten werden meist als *Klartext M* bezeichnet. Sie können ein Bitstrom, eine Textdatei, eine Bitmap, ein Video oder sonstige binäre Daten sein. Die transformierten Daten, häufig mit *Schlüsseltext C* bezeichnet, sind ebenfalls binäre Daten, allerdings ohne erkennbaren Informationsgehalt. Häufig wird der Vorgang der Verschlüsselung mit Hilfe einer mathematischen Funktion ausgedrückt:

  $$C = E(M) \qquad \text{(E = Encryption)}.$$

- Mit *Entschlüsselung* bezeichnet man den Vorgang der Transformation von Schlüsseltext in Klartext:

  $$M = D(C) \qquad \text{(D = Decryption)}.$$

Kryptographische Algorithmen lassen sich entsprechend ihrer Arbeitsweise in verschiedene Kategorien einordnen [Sch96]:

- *Beschränkte* Algorithmen werden geheimgehalten. Die Ver- und Entschlüsselung wird allein mit Hilfe des Algorithmus vorgenommen. Dies bedeutet, daß seine Kenntnis ausreicht, Klartext zu verschlüsseln und Schlüsseltext zu entschlüsseln. Offensichtlich besitzt diese Art von Algorithmen einige Nachteile.

  Will eine große Gruppe von Menschen miteinander auf diese Weise kommunizieren, muß jedes Mitglied den Algorithmus kennen. Scheidet jemand aus der Gruppe aus oder wird der Algorithmus bekannt, muß ein neuer Algorithmus benutzt werden.

  Weit wichtiger ist aber, daß eine Qualitätskontrolle und Standardisierung solcher Algorithmen nicht stattfinden kann, da sie geheim bleiben müssen. Besitzt eine Gruppe keinen versierten Kryptographen, so kann sie nie sicher sein, daß der von ihr benutzte Algorithmus wirklich seinen Zweck erfüllt. Auch die Nutzung von Standardhard- und -software ist nicht möglich.

- Moderne Algorithmen sind *schlüsselbasiert*, wobei ein Schlüssel einem möglichst langen Bitvektor und damit – binär interpretiert – einer sehr großen Zahl entspricht. Sowohl zur Ver- als auch zur Entschlüsselung wird ein Schlüssel benötigt. Die Sicherheit der Verschlüsselung basiert nur auf den verwendeten Schlüsseln, nicht aber auf dem Algorithmus selbst, der demzufolge veröffentlicht und analysiert werden kann. Abbildung 3.1 illustriert diesen Vorgang. Zur besseren Unterscheidbarkeit wird der Schlüssel für den Verschlüsselungsvorgang im folgenden mit K_e bezeichnet, der Schlüssel, der zur Entschlüsselung benutzt wird, mit K_d.

Gute Algorithmen sind heutzutage immer schlüsselbasiert [Sch96], beschränkte sind nur noch von historischem Interesse, da sie sich als zu anfällig erwiesen haben. Deshalb werden im folgenden ausschließlich schlüsselbasierte Algorithmen betrachtet. Um genauer auf sie eingehen zu können, ist eine weiter Unterscheidung dahingehend hilfreich, wieviele *verschiedene* Schlüssel sie verwenden.

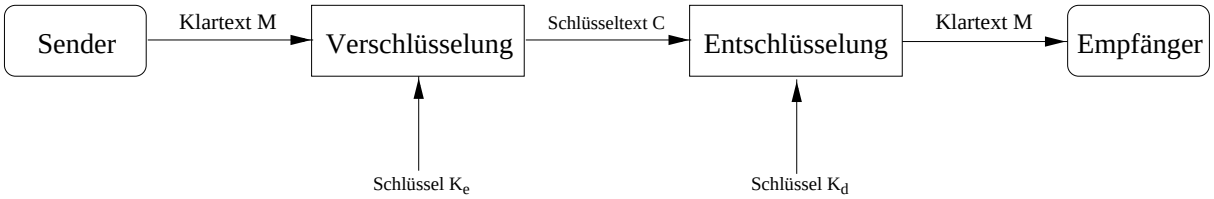


Abbildung 3.1: Die Arbeitsweise von schlüsselbasierten Algorithmen

3.2 Symmetrische Algorithmen

Kann man K_e mit wenig Aufwand aus K_d oder umgekehrt K_d aus K_e berechnen[2] oder ist $K_e = K_d$, dann bezeichnet man einen Algorithmus als *symmetrisch*. Da damit nur ein Schlüssel nötig ist - den zweiten kann man gegebenenfalls berechnen - spricht man bei symmetrischen Algorithmen immer vom Schlüssel K, der sowohl für Ver- als auch Entschlüsselung benutzt wird.

$C = E_K(M)$ (E_K = Verschlüsselung mit dem Schlüssel K) und
$M = D_K(C)$ (D_K = Entschlüsselung mit dem Schlüssel K).

Bei einer verschlüsselten Kommunikation müssen sich Sender und Empfänger vorher also auf einen *gemeinsamen* Schlüssel einigen. Dieser wird häufig als *geheimer Schlüssel* bezeichnet.

3.2.1 Block- und Streamchiffren

Die meisten symmetrischen Algorithmen sind sogenannte *Blockchiffren*. Sie operieren auf einzelnen Bitgruppen fester Länge, Blöcke genannt. Ein einzelner Block wird getrennt von allen anderen ver- und entschlüsselt. Der DES-Algorithmus (siehe Anhang A.1) etwa ist eine Blockchiffre, die auf 64-Bit-Blöcken operiert. Eine Blockchiffre kann in verschiedenen Betriebsarten arbeiten. Dabei werden aufeinanderfolgende Blöcke auf eine fest definierte Weise miteinander verknüpft, um damit die Entschlüsselung einzelner Blöcke zu erschweren. Bekannte Betriebsarten sind etwa ECB, CBC, CFB und OFB. Ihre Arbeitsweise kann in [Sch96] und [Rul93] nachgelesen werden.

Andere symmetrische Algorithmen operieren auf einem einzelnen Bit oder Byte und sind deshalb besonders gut dazu geeignet, einen Datenstrom zu verschlüsseln. Sie werden deshalb *Streamchiffren* genannt. RC4 (siehe Anhang A.1) ist ein Beispiel dafür. Bei Streamchiffren wird der Schlüssel zur Initialisierung eines Pseudo-Zufallszahlengenerators verwendet. Dieser generiert eine lange Folge binärer Werte. Der Klartext und die Ausgabe des Zufallszahlengenerators werden dann mit Hilfe einer Funktion, wie etwa XOR, miteinander verknüpft. Die Entschlüsselung geschieht auf analoge Weise, indem die inverse Funktion angewendet wird.

[2] Die Berechnung kann also mit Blick auf den Zeitaufwand für Ver- und Entschlüsselung vernachlässigt werden.

3.2.2 Paddingverfahren

Ein Problem von Blockchiffren besteht darin, daß sie nur auf kompletten n-Bit-Blöcken arbeiten können. Ist der Klartext zu kurz, d. h. der letzte Block ist nicht genau n Bit lang, muß er auf die geforderte Länge aufgefüllt werden. Für dieses *Padding* genannte Verfahren gibt es verschiedene Algorithmen, die in [Sch96] oder [Rul93] beschrieben werden.

Symmetrische Algorithmen besitzen mehrere Nachteile:

- Die Schlüsselverteilung muß geheim erfolgen. Ansonsten kann jeder, der den Schlüssel besitzt, damit verschlüsselte Nachrichten entschlüsseln. Gerade bei weltweit vernetzten System ist diese geheime Verteilung sehr schwierig.
- Es ist eine große Zahl von verschiedenen Schlüsseln nötig. Für jede Kommunikationsverbindung, die sicher sein soll, muß ein eigener Schlüssel vorhanden sein.

3.3 Asymmetrische Algorithmen

Unterscheiden sich die beiden bei Ver- und Entschlüsselung verwendeten Schlüssel, und ist es unmöglich (oder zumindest extrem zeitaufwendig), den einen aus dem anderen zu berechnen, so wird ein Algorithmus *asymmetrisch* genannt. K_e wird öffentlich bekannt gemacht, so daß jeder eine Nachricht mit ihm verschlüsseln kann. K_d ist dagegen geheim; nur sein Besitzer ist in der Lage, mit K_e verschlüsselte Nachrichten zu entschlüsseln:

$$C = E_{K_e}(M) \text{ und } M = D_{K_d}(C)$$

K_e wird deshalb auch als öffentlicher Schlüssel bezeichnet, während K_d geheimer Schlüssel genannt wird. Man spricht in diesem Fall auch von Public-Key Kryptographie und Public-Key Algorithmen [DH76].

Leider sind asymmetrische Algorithmen im Vergleich zu symmetrischen oft signifikant langsamer, so daß eine alleinige Anwendung dieser Algorithmen in den meisten Fällen ausscheidet. Hybride Verfahren, die beide Arten von Algorithmen benutzen, versuchen, die jeweiligen Nachteile der einzelnen Algorithmenarten zu umgehen. Ein in 4.3 vorgestelltes Verfahren basiert auf diesem Ansatz.

3.4 Digitale Signaturen

Eine Unterschrift erfüllt in unserer Gesellschaft eine Vielzahl wichtiger Funktionen [Sch96]:

- Eine Unterschrift ist authentisch. Sie bezeugt, daß der Unterzeichner das Dokument wohlüberlegt unterschrieben hat.
- Unterschriften sind nicht fälschbar.

- Man kann eine Unterschrift nicht wiederverwenden, indem man sie auf ein anderes Dokument überträgt.
- Der Inhalt eines unterschriebenen Dokuments kann nicht mehr verändert werden.
- Die Unterschrift eines Dokuments kann nicht mehr verleugnet werden.

Natürlich hält keine dieser Behauptungen der Wirklichkeit stand. Unterschriften können sehr wohl gefälscht oder erzwungen werden und Dokumente können nachträglich verändert worden sein. Und trotzdem geschieht es selten, da die dabei zu überwindenden Schwierigkeiten unverhältnismäßig groß sind.

Diese Form von Unterschrift wäre auch für elektronische Dokumente von Vorteil. Eine elektronische Unterschrift, auch *digitale Signatur* genannt, muß dieselben Vorteile wie eine herkömmliche bieten. Es existieren allerdings verschiedene Probleme, die digitale Signaturen schwierig machen:

- Computerdateien, und um solche handelt es sich bei elektronischen Dokumenten, sind einfach zu kopieren. Auch Teile davon, wie etwa die Signatur selbst, kann man einfach in andere Dateien übernehmen, ohne daß dies später zu erkennen wäre.
- Computerdateien sind einfach zu ändern. Eine solche Änderung ist zu einem späteren Zeitpunkt nicht mehr nachweisbar.

Kryptographie bietet mehrere Möglichkeiten, diese Probleme zu lösen. Es gibt verschiedene Ansätze, wie man digitale Signaturen mit kryptographischen Verfahren realisieren kann [Sch96]. Durchgesetzt haben sich Verfahren, die auf Public-Key Kryptographie basieren. Sie werden in Abschnitt 3.4.2 genauer erklärt.

Eine Möglichkeit ist, die zu signierenden Daten mit einem geheimen Schlüssel und einem dazu passenden Public-Key Algorithmus zu verschlüsseln. Jeder Empfänger einer Nachricht könnte einwandfrei den Absender der Nachricht ermitteln, indem er sie mit dem entsprechenden öffentlichen Schlüssel entschlüsselt. Funktioniert die Entschlüsselung, dann kann niemand anderer als der Besitzer des entsprechenden geheimen Schlüssels die Nachricht verschlüsselt haben. Verschlüsselt man Daten mit einem geheimen Schlüssel, bezeichnet man dies als *Signierung*. Der umgekehrte Vorgang der Bestätigung einer Signatur wird *Verifikation* genannt.

Wie schon in Abschnitt 3.3 erwähnt, sind asymmetrische Verfahren meist sehr langsam, so daß das Signieren von längeren Nachrichten zu lange dauert. Eine Alternative bietet hier die Verwendung von One-way Hashfunktionen zusammen mit Public-Key Kryptographie.

3.4.1 One-way Hashfunktionen

Die Funktionsweise einer normalen Hashfunktion ist bekannt. Aus einer langen Eingabe wird ein relativ kurzer Wert konstanter Länge, *Hashwert* genannt, berechnet. Dabei kann es durchaus vorkommen, daß zwei verschiedene Eingaben denselben Hashwert produzieren.

One-way Hashfunktionen funktionieren auf dieselbe Weise. Sie berechnen zu einer langen Eingabe einen relativ kurzen Hashwert. Allerdings werden weitere Eigenschaften gefordert:

- Während der Hashwert sehr leicht und schnell zu berechnen sein muß, darf der umgekehrte Vorgang, d. h. die Berechnung einer Eingabe zu einem Hashwert, nicht oder nur mit sehr großem zeitlichen Aufwand möglich sein.
- Zwei verschiedene Eingaben sollen fast immer zwei verschiedene Hashwerte liefern. Diese Eigenschaft wird auch *Kollisionsfreiheit* genannt.

Man bezeichnet diese Art von Funktionen häufig auch als Message Digest, digitalen Fingerabdruck oder digitale Prüfsumme.

3.4.2 Public-Key Kryptographie und One-way Hashfunktionen

Besitzt man eine One-way Hashfunktion, kann auf einfache Weise eine digitale Signatur berechnet werden. Zuerst bestimmt man den Hashwert der gegebenen Daten. Dann signiert man nur diesen Hashwert. Daten und signierter Hashwert können anschließend zusammen abgeschickt werden. Verglichen mit der Signierung der kompletten Daten ist der Zeitaufwand erheblich geringer, da die Größe des Hashwertes im Vergleich zu der Größe der Datenmenge gering ist.

Oft wird der Vorgang der Signierung als *Verschlüsselung mit einem geheimen Schlüssel* bezeichnet. Entsprechend nennt man die Verifikation häufig auch *Entschlüsselung mit dem öffentlichen Schlüssel*. Genaugenommen ist dies aber nur zulässig, wenn es sich bei dem Public-Key Algorithmus um RSA handelt. Verallgemeinern lassen sich diese Bezeichnungen nicht. Vielmehr ist es so, daß Signaturalgorithmen allesamt Public-Key Algorithmen sind, die geheime Informationen nutzen, um eine Signatur zu berechnen, und die öffentliche Informationen benutzten, um eine Signatur zu verifizieren.

Das Signieren einer Nachricht M wird mit

$$C = S_K(M)$$

abgekürzt, während das Verifizieren einer Signatur mit

$$M = V_K(M)$$

notiert wird. Obwohl bei beiden Vorgängen eigentlich zwei verschiedene Schlüssel verwendet werden, wird aus Gründen der Übersichtlichkeit stets von einem einzigen Schlüssel K gesprochen. Es sollte aus dem textuellen Zusammenhang stets klar sein, welcher Schlüssel gemeint ist.

3.5 Message Authentication Codes (MAC)

Ein *Message Authentication Code* ist eine One-Way Hashfunktion, wobei zusätzlich ein geheimer Schlüssel verwendet wird, um einen Ausgangswert, den Hash- oder MAC-Wert, zu

berechnen. Der MAC-Wert ist dadurch neben dem Klartext auch vom geheimen Schlüssel abhängig. Nur wer diesen besitzt, ist in der Lage, den MAC-Wert zu verifizieren.

MACs sind oft abgewandelte Hashfunktionen oder Blockchiffren. Allerdings existieren auch andere Verfahren, die nur zur Berechnung eines MACs verwendet werden können [Sch96].

Kapitel 4

Der Einsatz von Kryptographie in AutO

Kryptographie dient in erster Linie dazu, die Vertraulichkeit von Daten zu gewährleisten. Durch die Tatsache, daß immer mehr Dokumente auch in elektronischer Form vorhanden sind, wurden für Kryptographie weitere Einsatzgebiete erschlossen: Authentifizierung (Feststellen der Identität des Autors eines Dokuments) und Integritätsüberprüfung (Erkennen von Änderungen an einem Dokument) sind zwei Beispiele hieraus [Sch96].

Um Kryptographie sinnvoll in AutO einsetzen zu können, steht an erster Stelle die Analyse der Orte, wo Daten in AutO auftreten und wie sie dort geschützt werden können:

- Auf Festspeichern sind sowohl die eigentlichen Programmdaten gespeichert, d. h. alle Dateien, die in einer AutO-Installation enthalten sind, als auch Benutzerdaten, in Form von serialisierten autonomen Objekten. Während erstere durch das verwendete Betriebssystem verwaltet werden, sind die Objektdaten in einem DBMS abgelegt, das selbstständig für den Schutz der in ihm gespeicherten Daten sorgen kann.
- Betrachtet man ein laufendes AutO-System, so lagern alle benutzten Daten im Hauptspeicher der verwendeten Rechner (Der Swap-Speicher, sofern einer verwendet wird, zählt dabei zum Hauptspeicher, auch wenn er eigentlich auf einer Festplatte liegt). Darunter fallen sowohl Programm- als auch Benutzerdaten. Es ist die Aufgabe des Betriebssystems, die Sicherheit dieser Daten zu gewährleisten.
- Die einzelnen Komponenten verteilter Systeme kommunizieren untereinander über ein dazwischenliegendes Netzwerk. Dabei werden Daten in Form von Nachrichten von einem Rechner zu einem anderen gesendet. AutO benutzt dazu die von Java zur Verfügung gestellten Sockets, die Daten unverschlüsselt an den Zielrechner übermitteln. Ein Schutz dieser Verbindungen mit kryptographischen Methoden ist notwendig.

Die ersten beiden Aufgaben müssen, teils weil sie mit Java nur schwer oder gar nicht realisierbar sind, teils weil Betriebssysteme oder DBMSe diese Art von Funktionalität schon bieten, von den oben erwähnten Systemen übernommen werden. Die Entwicklung

und Implementierung einer sicheren Kommunikation auf den genutzten Verbindungen wird in AutO realisiert.

4.1 Kommunikationsverbindungen in AutO

AutO etabliert im Betrieb eine große Zahl von Verbindungen[1], die verschiedenen Zwecken dienen. Eine genaue Analyse der existierende Verbindungsarten ist nötig, um bestimmen zu können, welche Verbindungen auf welche Weise geschützt werden müssen:

- Node-Manager $\longleftrightarrow$ Node-Manager
  Das Austausch von Nachrichten zwischen AutO-Objekten läuft, sofern sie nicht im selben AutO-Prozeß residieren, immer über die entsprechenden Node-Manager der Rechner, auf denen die Objekte liegen. Sollten beide Kommunikationspartner auf demselben Rechner, aber in verschiedenen Prozessen liegen, wird die Kommunikation über den lokalen Node-Manager und dessen Verbindungen zu seinen AutO-Prozessen abgewickelt. Eine Verbindung zwischen zwei Node-Managern bedeutet Kommunikation zwischen den entsprechenden Rechnern, die über ein LAN oder ein WAN, wie etwa das Internet, verbunden sind. Gerade bei Nutzung des Internets als Verbindungsmedium kann man nicht von einer sicheren Datenübertragung ausgehen.

- Node-Manager $\longleftrightarrow$ AutO-Prozeß
  Obwohl diese Verbindung über gewöhnliche Java-Sockets hergestellt wird, werden nur zwei Prozesse auf *demselben* Rechner verbunden. Dabei sorgt die Implementierung des TCP/IP-Protokolls des verwendeten Betriebssystems, auf die ein Java-Socket zugreift, dafür, daß der Nachrichtenaustausch auf dieser Verbindung nicht über das Netzwerk läuft, sondern lokal auf dem Rechner abgewickelt wird. Das Betriebssystem gewährleistet dabei die Sicherheit der Kommunikation. Als einziges ist sicherzustellen, daß die Verbindung wirklich zwischen einem AutO-Node-Manager und einem lokalen AutO-Prozeß aufgebaut wird.

- Node-Manager $\longleftrightarrow$ Name-Service
  Der Nachrichtenaustausch bei einer derartigen Verbindung läuft über ein dazwischenliegendes Netzwerk, sofern Node-Manager- und Name-Service-Prozeß nicht auf demselben Rechner ausgeführt werden. Sie ist demzufolge als nicht sicher anzusehen und muß geschützt werden.

- Node-Manager $\longleftrightarrow$ DBMS (Persistenz, Information-Service)
  Diese Verbindung wird zwischen einem Node-Manager-Prozeß und einem Server-Prozeß des benutzten Datenbanksystems hergestellt. Verschiedene DBMSe bieten die Option an, ausgehende Daten zu verschlüsseln; es wäre also nötig, in die entsprechenden Java-Schnittstellen, die das DBMS ansprechen, die benötigten Kryp-

[1] Mit *Verbindung* wird im folgenden die Kommunikation zwischen zwei AutO-Komponenten über einen Java-Socket bezeichnet.

tographieroutinen einzubauen, um sicher mit den DBMS-Server kommunizieren zu können.

- Auto-Prozeß $\longleftrightarrow$ DBMS (Persistenz)
  Jeder AutO-Prozeß kann eine konfigurierbare Anzahl von Verbindungen zu einem DBMS aufbauen, um dort Daten, die persistent gehalten werden müssen, abzuspeichern (siehe dazu auch Abschnitt 2.2 und [Gri97]). Wie bei einer Verbindung zwischen Node-Manager und DBMS müßten auch hier Kryptographiealgorithmen in die entsprechenden Java-Schnittstellen integriert werden, um eine sichere Kommunikation zu ermöglichen.

- AutO-Prozeß (Terminal-Server) $\longleftrightarrow$ Terminal-Client
  Da der Terminal-Server vollständig in das AutO-System eingebettet ist, ist es beim Betrieb eines solchen nötig, diese Verbindung vor Angriffen zu schützen.

Neben den bisher angesprochenen Verbindungen nutzt AutO ebenfalls TCP/IP-Verbindungen, um verschiedene Server, wie etwa den Signature-Server (Abschnitt 4.4) oder die Security-Data-Administration (Abschnitt 5.4.4) anzusprechen. Die Programme, die zur Kommunikation mit solchen Servern benutzt werden, bauen eine Socket-Verbindung zum entsprechenden Server auf. Diese Verbindung muß mit Hilfe kryptographischer Methoden gesichert werden. Genauere Informationen dazu sind bei den Beschreibungen der einzelnen Hilfsprogramme, z. B. dem *SRTool*, dem *SATool* sowie dem *SDA Administration Tool*, in Anhang D zu finden.

Bevor damit begonnen werden kann, Maßnahmen zum Schutz der einzelnen Verbindungen zu entwerfen, ist es nötig, sich über die Angriffspunkte einer Verbindung klar zu werden.

4.2 Mögliche Angriffe gegen eine AutO-Verbindung

Will man die Sicherheit einer Verbindung zwischen verschiedenen Parteien gewährleisten, muß man verhindern, daß eine unautorisierte Partei den gesendeten Datenstrom abhören oder verfälschen kann [Sch96, Rul93].

- **Passive Angriffe**: Diese Art von Angriffen gegen eine Kommunikationsverbindung führt nicht zu einer Änderung der übertragenen Daten, sondern sie bedroht die Vertraulichkeit der übermittelten Informationen. Mögliche Arten von passiven Angriffen sind:

  - das Abhören von Vermittlungsinformationen, wie sie etwa Sender und Empfänger einer Nachricht darstellen,
  - das Abhören von Benutzerinformationen, d. h. von in einer Nachricht enthaltenen Nutzdaten oder
  - die Analyse des Verkehrsflusses.

  Eine geeignete Maßnahme gegen solche Angriffe besteht darin, die gesendeten Daten zu verschlüsseln.

- **Aktive Angriffe**: Werden einzelne Nachrichten oder der gesamte Nachrichtenstrom verfälscht oder verändert, spricht man von aktiven Angriffen gegen eine Kommunikationsverbindung. Vorstellbar sind etwa:
  - die Wiederholung oder Verzögerung einer Nachricht,
  - das Einfügen, Modifizieren oder Löschen von übermittelten Daten oder
  - das Vortäuschen einer falschen Identität durch einen der Kommunikationspartner.

  Das Signieren von Nachrichten oder die Berechnung eines MAC-Wertes für Nachrichten kann diese Art von Angriffen zwar nicht verhindern, ermöglicht es aber, sie zu entdecken und entsprechend darauf zu reagieren.

4.3 Der AutO-Socket

Java bietet eine einfache Möglichkeit, eine TCP/IP-Verbindung zwischen zwei Parteien aufzubauen, indem man die beiden Klassen `ServerSocket` und `Socket` aus dem Paket `java.net` verwendet. Eine Partei, die eine Verbindung aufbauen möchte, erzeugt zuerst ein Objekt der Klasse `Socket`, das mit Rechneradresse und Portnummer des gewünschten Verbindungspartners initialisiert wird. Wird die Verbindung akzeptiert, kann die Kommunikation der beiden Parteien über normale Java-Streams, die der Socket zur Verfügung stellt, abgewickelt werden. Alle AutO-Komponenten verwenden diese Socket-Klassen, um Verbindungen aufzubauen.

Um eine sichere Verbindung in AutO einfach zu realisieren, bietet es sich an, von den beiden Klassen `ServerSocket` und `Socket` neue Klassen abzuleiten. So ist es möglich, einerseits dieselbe Funktionalität wie die ursprünglichen Socket-Klassen zur Verfügung zu stellen, andererseits die neuen Klassen aber auch so zu erweitern, daß sie eine sichere Verbindung zwischen den beteiligten Parteien garantieren können. Die beiden abgeleiteten Klassen, `AutOServerSocket` und `AutOSocket` im Paket `security.connection`, können nun statt den originalen Java-Klassen verwendet werden, um Socket-Verbindungen in AutO aufzubauen.

4.3.1 Das Interface `security.connection.SecureMessage`

Ein wichtiger, von außen erkennbarer Unterschied zwischen AutO-Sockets und normalen Java-Sockets besteht darin, daß über einen AutO-Socket nur noch Objekte versendet werden können, die das Interface `security.connection.SecureMessage` implementieren. Diese Einschränkung ist nötig, da es in AutO nicht sinnvoll ist, alle Daten zu verschlüsseln und/oder mit einem MAC (Message Authentication Code, siehe Abschnitt 3.5) zu versehen, die über eine Verbindung gesendet werden. Manchmal müssen Nachrichten nur zum Teil verschlüsselt sein, oft auch gar nicht. Ebenso besteht die Möglichkeit, daß bestimmte Nachrichten nicht wichtig genug sind, um den Zeitaufwand zu rechtfertigen, der für die Berechnung/Verifikation eines MACs nötig ist.

In einer Klasse, die das Interface `SecureMessage` implementiert, können die Interface-Methoden sinnvoll implementiert sein. Die Klasse kann beispielsweise ganz verschlüsselt werden oder auch gar nicht. Außerdem können Subklassen ihre eigenen Implementierungen der Interface-Methoden beinhalten. Auf diese Art und Weise kann der Performanceverlust, der durch die Verwendung von AutOSockets entsteht - denn natürlich kostet der Einsatz von Kryptographie zur Verschlüsselung und MAC-Berechnung von Nachrichten CPU-Zeit - sehr genau kontrolliert werden. Allerdings bleibt festzuhalten, daß der Vorteil des AutOSockets, nämlich die Sicherung einer Verbindung vor Angriffen, den Nachteil in Form von geringen Performanceeinbußen signifikant übertrifft.

Folgende Methoden werden im Interface `SecureMessage` definiert:

- `public abstract boolean getMACState()`: Diese Methode teilt dem AutOSocket mit, ob für eine Nachricht ein MAC berechnet werden soll. Liefert sie `true` zurück, wird ein MAC berechnet, wodurch die Integrität einer Nachricht bei Erhalt festgestellt werden kann.

- `public abstract boolean getEncryptionState()`: Liefert diese Methode den Wert `true` zurück, wird eine ausgehende Nachricht verschlüsselt.

Werden nur die beiden bisher vorgestellten Methoden implementiert, wird immer die komplette Nachricht verschlüsselt. Oftmals sind aber nur bestimmte Teile einer Nachricht so wichtig, daß eine Verschlüsselung nötig ist. Deshalb bietet das Interface `SecureMessage` die Möglichkeit, zwei weitere Methoden zu implementieren. Wird dies gemacht, werden sie bei der Ver- bzw. Entschlüsslung aufgerufen. Dadurch kann die Nachricht selbst bestimmen, welche Teile von ihr gesichert übertragen werden müssen. Anzumerken ist, daß eine Nachricht, wenn sie diese beiden Methoden implementiert, auch dafür Sorge tragen muß, daß die Daten aller Basisklassen entsprechend behandelt werden. Der AutOSocket wird Daten von gegebenenfalls existierenden Basisklassen nicht ver- und entschlüsseln.

Für den Fall, daß eine Nachricht selbst bestimmen will, welche Daten von ihr und ihren Basisklassen verschlüsselt werden sollen, muß sie die folgenden beiden Methoden implementieren:

- ```
  public void encryptMessage(java.io.ObjectOutputStream out,
                             javax.crypto.Cipher cipher)
                             throws java.io.IOException
  ```

  Diese Methode wird aufgerufen, wenn die Nachricht verschlüsselt werden soll. Die Implementierung dieser Methode sollte das übergebene `Cipher`-Objekt nutzen, um Daten zu verschlüsseln und dann in den angegebenen `ObjectOutputStream` zu schreiben.

- ```
  public void decryptMessage(java.io.ObjectInputStream in,
                             javax.crypto.Cipher cipher)
                             throws java.io.IOException
  ```

  Der AutOSocket ruft diese Methode auf, um eine Nachricht zu entschlüsseln. Das `Cipher`-Objekt sollte genutzt werden, um die Daten, die im `ObjectInputStream` liegen, zu entschlüsseln. Der `ObjectInputStream` enthält diejenigen Daten, die beim

Aufruf der Methode `encryptMessage` in den `ObjectOutputStream` geschrieben wurden.

4.3.2 Die Arbeitsweise eines AutO-Sockets

Während die AutO-Socket-Klassen genauso benutzt werden wie die normalen Java-Socket-Klassen, unterscheiden sie sich intern sehr stark. Natürlich bauen sie auf der Klasse `Socket` auf und nutzen diese, um die Verbindung zum Zielrechner herzustellen. Anschließend beginnt die eigentliche Arbeit eines AutO-Sockets.

Um eine sichere Verbindung zwischen zwei Parteien aufbauen zu können, ist es nötig, einige wichtige Daten auszutauschen. Sie sind notwendig, um Nachrichten, d. h. Objekte, die `SecureMessage` implementieren, übertragen zu können. Dieser Datenaustausch geschieht mit Hilfe von verschiedenen Protokollen, die weiter unten vorgestellt werden. Gemeinsam ist allen Protokollen, daß sie einzig und allein dazu dienen, einen *Session-Key* zu übertragen. Momentan reicht es, sich unter einem Session-Key einen Schlüssel für einen symmetrischen Kryptographiealgorithmus vorzustellen; eine genauere Erklärung folgt, wenn auf die verschiedenen Protokolle eingegangen wird.

Die Protokolle des AutO-Socket

Die Protokolle und damit der AutO-Socket setzen voraus, daß beide Parteien über ein Schlüsselpaar, bestehend aus einem öffentlichem und einem geheimen Schlüssel, für ein asymmetrisches Verschlüsselungsverfahren wie etwa RSA verfügen. Zusätzlich muß der öffentliche Schlüssel signiert sein. Man bezeichnet einen derartigen Schlüssel, d. h. einen signierten öffentlichen Schlüssel, als *Zertifikat*. Im Falle des AutO-Sockets muß der öffentliche Schlüssel von einer bestimmten Instanz, dem *Signature-Server*, signiert worden sein. Dessen öffentlicher Schlüssel, ebenso wie das eigene Schlüsselpaar einer Partei, müssen dem AutO-Socket bei seiner Erzeugung übergeben werden.

Das Protokoll *InitializeConnection*: Will eine Partei A als Client Verbindung zu einer Partei B, die den Server darstellt, aufnehmen, instanziiert sie einen `AutOSocket`, wobei sie ihm ihr eigenes Schlüsselpaar, bestehend aus ihrem geheimen Schlüssel A_g und ihrem zertifizierten öffentlichen Schlüssel $A_{\ddot{o}}$, sowie den öffentlichen Schlüssel des Signature-Server $S_{\ddot{o}}$ übergibt. Analog muß der `AutOServerSocket` von B initialisiert werden. Nach dem Aufbau einer normalen Java-Socket-Verbindung führt der AutO-Socket das Protokoll *InitializeConnection* aus, das in Abbildung 4.1 dargestellt wird:

- Zuerst sendet A eine *CertificateMessage*. Sie enthält den zertifizierten öffentlichen Schlüssel $A_{\ddot{o}}$ von A. B kann nach Empfang dieser Nachricht das Zertifikat auf seine Echtheit überprüfen, indem sie es mit dem öffentlichen Schlüssel $S_{\ddot{o}}$ des Signature-Server verifiziert. Anschließend kann B sich sicher sein, wirklich den öffentlichen Schlüssel $A_{\ddot{o}}$ von A erhalten zu haben.

- B antwortet nun ebenfalls mit einer *CertificateMessage*, die den zertifizierten öffentlichen Schlüssel $B_{\ddot{o}}$ von B enthält. Nachdem A das Zertifikat verifiziert hat, kennen beide Parteien den öffentlichen Schlüssel der jeweils anderen Partei.

- A generiert nun einen Schlüssel für ein in der Datei `AutO.config` festgelegtes symmetrisches Verfahren. Dieser Schlüssel, *Session-Key* genannt, wird im folgenden dazu benutzt, Nachrichten zu ver- und entschlüsseln sowie den MAC einer Nachricht zu berechnen und zu verifizieren. Der Session-Key wird von A mit dem eigenen geheimen Schlüssel A_g signiert, dann mit dem öffentlichen Schlüssel $B_{\ddot{o}}$ von B verschlüsselt und schließlich in einer *TransmitSessionKeyMessage* an B gesendet:

  $$TransmitSessionKeyMessage = E_{B_{\ddot{o}}}(S_{A_g}(SessionKey)).$$

  Nur B ist in der Lage, die in der Nachricht enthaltenen Daten mit dem eigenen geheimen Schlüssel B_g zu entschlüsseln und die Signatur mit $A_{\ddot{o}}$ zu verifizieren:

  $$SessionKey = D_{B_g}(V_{A_{\ddot{o}}}(TransmitSessionKeyMessage)).$$

- Da A nicht sicher sein kann, daß B wirklich die Nachricht mit dem Session-Key erhalten hat, sendet sie daran anschließend eine *FinishedMessage*. Diese enthält eine Prüfsumme, die aus allen bisherigen Protokollnachrichten berechnet worden ist. Zugleich wird sie als erste Nachricht mit dem vorher generierten Session-Key verschlüsselt. Empfängt B diese Nachricht, kann die Prüfsumme entschlüsselt und mit dem eigenen berechneten Wert verglichen werden. Stimmen die beiden Prüfsummen überein, kann sich B sicher sein, daß A alle Nachrichten von B und B alle Nachrichten von A erhalten hat. B berechnet daraufhin eine neue aktuelle Prüfsumme, verschlüsselt sie und schickt sie in einer *FinishedMessage* an A, die sie analog überprüft.

Nach Abschluß dieses Protokolls sind beide Parteien – und nur sie – in Besitz des Session-Keys. Zugleich können sie sicher sein, daß die Partei am anderen Ende der Verbindung genau die ist, die sie vorgegeben hat zu sein.

Das Protokoll *ContinueConnection*: Immer wenn eine Nachricht an eine nichtlokale Komponente gesendet werden soll, baut AutO eine Verbindung auf und beendet sie meist, wenn die Nachricht gesendet wurde. Um nicht bei jedem Verbindungsaufbau einen neuen Session-Key generieren zu müssen – diese Aktion ist die bei weitem aufwendigste beim Instanziieren eines AutO-Sockets –, werden die Session-Keys und Zertifikate anderer Parteien gepuffert. Wird eine neue Verbindung aufgebaut, d. h. ein Objekt der Klasse `AutOSocket` erzeugt, wird zuerst im Cache nachgesehen, ob für die gewünschte Zielpartei bereits ein gültiger Session-Key existiert. Ist dies nicht der Fall, wird das obengenannte *InitializeConnection*-Protokoll ausgeführt. Wird allerdings ein gültiger Session-Key gefunden, kann ein einfacheres Protokoll verwendet werden, das *ContinueConnection* genannt wird. Dabei sendet A nur eine *FinishedMessage*, die wie in *InitializeConnection* eine Prüfsumme über alle bisher gesendeten Protokollnachrichten enthält. B beantwortet diese mit einer eigenen *FinishedMessage*, wobei jeweils die übertragenen Prüfsummen verifiziert werden. Abbildung 4.2 zeigt den genauen Ablauf.

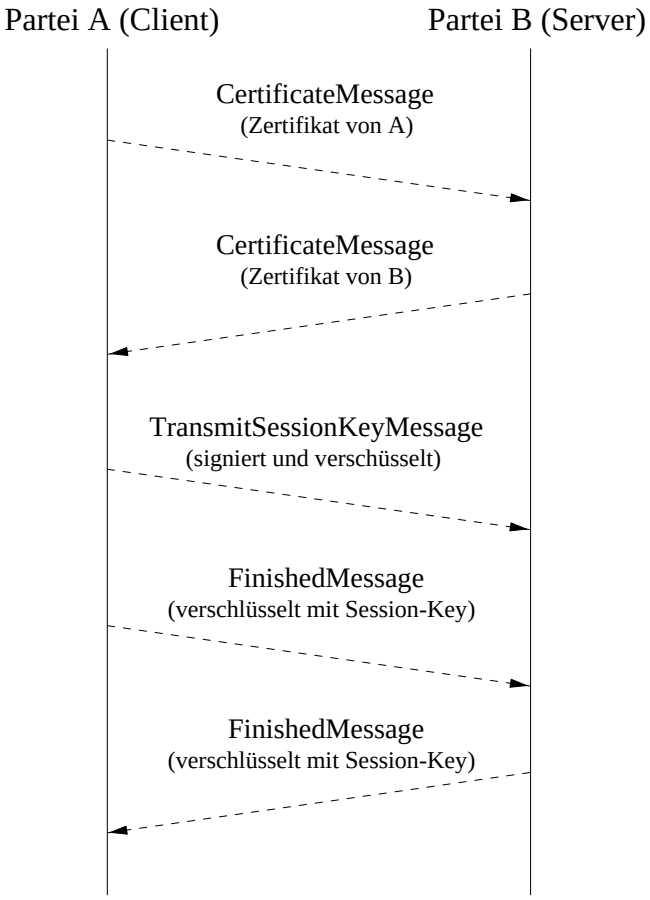


Abbildung 4.1: Das Protokoll *InitializeConnection*

Das Protokoll *UpdateSessionKey*: Die Verwendung von Session-Keys ist sinnvoll, da die Verschlüsselung mit einem asymmetrischen Verfahren zu langsam ist. Symmetrische Verfahren sind signifikant schneller [Sch96]. Aus mehreren Gründen ist die Verwendung eines solchen Session-Keys über einen längeren Zeitraum aber nicht zu empfehlen. Ist die Schlüssellänge ausreichend groß gewählt, so daß ein Brechen der Verschlüsselung in absehbarer Zeit nicht möglich ist, dann ist auch die Zeit, die für das Ver- bzw. Entschlüsseln von Nachrichten benötigt wird, groß. Wählt man hingegen eine kürzere Schlüssellänge, so daß Nachrichten ausreichend schnell verschlüsselt werden können, ist die Gefahr, daß die Verschlüsselung gebrochen wird, zu groß. Natürlich wächst die Gefahr, daß eine Verschlüsselung gebrochen wird, mit der Zeitdauer, die der dazu gehörende Schlüssel eingesetzt wird. Deshalb ist es wichtig, in regelmäßigen Intervallen veraltete Session-Keys zu invalidieren.

Ein eigener Thread überwacht dazu den Cache, der alle Session-Keys verwaltet. Jeder Session-Key besitzt einen Zeitstempel, der angibt, wann er generiert wurde. Entdeckt der Thread nun, daß ein Session-Key veraltet ist, markiert er ihn als ungültig. Allerdings geschieht dies nur bei Session-Keys, die in dem lokalen Prozeß erzeugt wurden. Denn nur bei diesen kann er das wirkliche Alter bestimmen, indem er die lokale Systemzeit mit dem

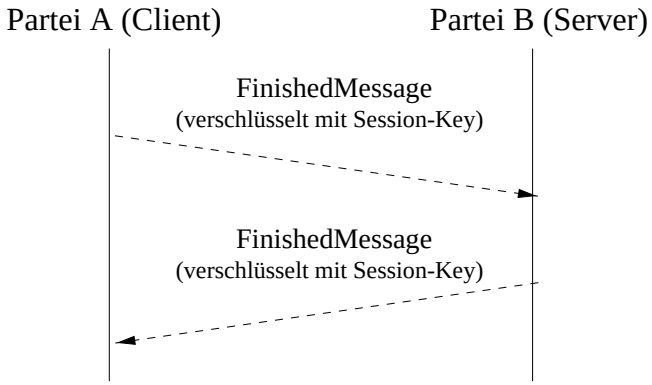


Abbildung 4.2: Das Protokoll *ContinueConnection*

Zeitstempel vergleicht. Bei Schlüsseln, die auf anderen Rechnern erzeugt wurden, muß die dortige Systemzeit nicht mit der lokalen übereinstimmen.

Ein AutO-Socket A, der auf einen ungültigen Session-Key zugreifen will, erkennt dies und führt daraufhin das Protokoll *UpdateSessionKey* aus. Dabei wird ein neuer Schlüssel generiert und an die gegenüberliegende Partei B gesendet, so daß auch sie ihren Cache aktualisieren und neue Nachrichten entschlüsseln kann.

- Nachdem A entdeckt hat, daß der zur aktuellen Verbindung gehörende Session-Key veraltet ist, wird ein neuer generiert. Er wird dann, wie auch beim Protokoll *InitializeConnection*, mit dem eigenen geheimen Schlüssel A_g signiert und mit dem öffentlichen Schlüssel $B_{\ddot{o}}$ von B verschlüsselt. Die so erhaltenen Daten werden mit einer *UpdateSessionKeyMessage* an B gesendet.

  $$UpdateSessionKeyMessage = E_{B_{\ddot{o}}}(S_{A_g}(SessionKey)).$$

  B entschlüsselt und verifiziert die Nachricht und ist damit im Besitz des neuen Session-Keys.

  $$SessionKey = D_{B_g}(V_{A_{\ddot{o}}}(UpdateSessionKeyMessage)).$$

- Analog zum *InitializeConnection*-Protokoll folgt nun die Endphase, in der zuerst A eine *FinishedMessage* an B sendet, und B darauf mit einer eigenen *FinishedMessage* antwortet. Jede der beiden Parteien verifiziert dabei die in der Nachricht enthaltene Prüfsumme.

Versionsverwaltung der Schlüssel für das asymmetrische Verfahren

Wenn man die Entwicklung der Leistungsfähigkeit von Computern betrachtet, wird ersichtlich, daß es vernünftig ist, Schlüssel regelmäßig neu zu generieren, gegebenenfalls sogar längere Schlüssel zu verwendet. Denn je leistungsfähiger ein Rechner ist, desto weniger

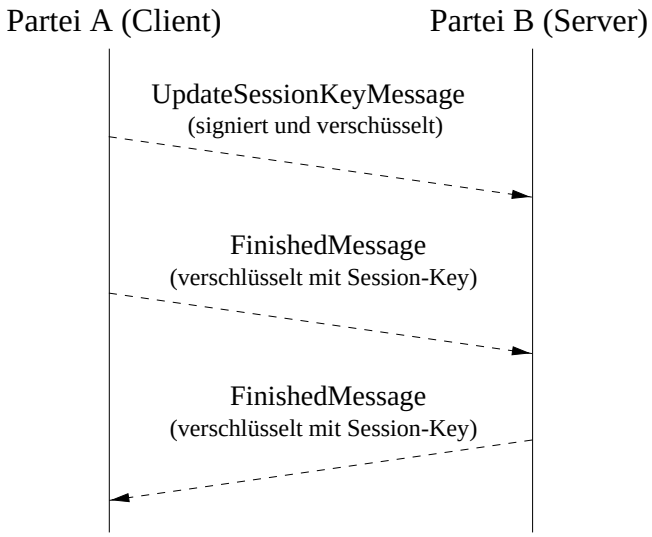


Abbildung 4.3: Das Protokoll *UpdateSessionKey*

Zeit benötigt er, um eine Verschlüsselung zu brechen. Schon heute werden Hunderte von Rechnern über Netzwerke, wie etwa das Internet, verbunden, womit es möglich wird, auch die Verschlüsselung von Daten, die mit langen Schlüsseln verschlüsselt wurden, zu brechen. Zusätzlich entwickelt sich auch das Gebiet der Kryptoanalyse, d. h. der Wissenschaft des Entschlüsselns von Schlüsseltexten ohne die dazu nötigen Schlüssel, immer weiter. Es können außerdem Schwachstellen in Algorithmen entdeckt werden, die es vereinfachen, Daten zu entschlüsseln, ohne den notwendigen Schlüssel zu kennen.

Während Session-Keys regelmäßig neu generiert werden, sofern dies gewünscht und in der Datei `AutO.config` spezifiziert wurde, geschieht dies nicht bei dem Schlüsselpaar, das jeder Kommunikationspartei zugeordnet ist. Ein Problem ist, daß der jeweilige öffentliche Schlüssel zertifiziert sein muß. Dieser Prozeß ist nicht automatisierbar und benötigt deshalb einige Zeit, so daß ein häufiges Neugenerieren dieses Schlüsselpaares nicht sinnvoll ist. Außerdem sollte die Länge der Schlüssel von Anfang an so gewählt werden, daß sie auf längere Zeit sicher sind. Um nun aber trotzdem die Möglichkeit zu bieten, dieses Schlüsselpaar gegen ein neueres und sichereres auszutauschen, wird jedes mit einer Versionsnummer versehen. Diese wird erhöht, falls eine neues Schlüsselpaar generiert wird.

Benutzt werden diese Schlüssel beim Senden der Protokollnachrichten *TransmitSessionKeyMessage* und *UpdateSessionKeyMessage*. Nur in der zweiten Nachricht müssen die beiden Versionsnummern, d. h. die des Schlüsselpaares des Absenders und die des Empfängers der Nachricht, vom Absender mitübertragen werden. Beim Senden einer *TransmitSessionKeyMessage* ist es nicht nötig, denn dabei werden zu Beginn des Protokolls sowieso die aktuellen Zertifikate, also die vom Signature-Server signierten öffentlichen Schlüssel, ausgetauscht. Somit kann ausschließlich beim Übertragen einer *UpdateSessionKeyMessage* der Fall eintreten, daß A, die Partei, die diese Nachricht abschickt, nicht über das aktuelle Zertifikat von B verfügt. B überprüft deshalb beim Empfang dieser Nachricht die beiden Versionsnummern. Stimmt eine nicht mit den im Cache gespeicherten überein,

bricht sie das laufende Protokoll ab und leitet das *InitializeConnection*-Protokoll ein, das mit dem Austausch der aktuellen Zertifikate beginnt.

4.4 Der Signature-Server

Jede Partei, die einen AutO-Socket benutzen möchte, muß ein Schlüsselpaar für ein asymmetrisches Kryptographieverfahren besitzen. Der öffentliche Schlüssel muß zertifiziert sein. Die Instanz, die dies erledigt, wird *Signature-Server* genannt.

Soll ein neuer Rechner in ein laufendes AutO-System integriert werden, muß bestätigt werden, daß der AutO-Code, der auf dem neuen Rechner ausgeführt wird, korrekt ist. Er darf also nicht modifiziert sein, so daß er sich gegebenenfalls anders verhält als das restliche System. In AutO erledigen diese Bestätigung ausgezeichnete Personen, im folgenden *Security-Representative* genannt. Sie untersuchen den neuen Rechner sowie die entsprechenden Java- und AutO-Klassen auf Korrektheit. Ist alles in Ordnung, generieren sie die verschiedenen sicherheitsrelevanten Daten, die ein neuer AutO-Node benötigt, also etwa das oben angesprochene Schlüsselpaar oder die Sicherheitsklasse des Rechners [Sel99]. Diese Daten müssen nun signiert werden, einerseits um allen anderen Kommunikationsparteien in AutO die Möglichkeit zu geben, ihre Echtheit zu prüfen, andererseits auch um sie gegen Modifikationen zu schützen. Der Security-Representative, der die Daten generiert hat, sendet sie über eine sichere Verbindung (AutO-Socket) an den Signature-Server. Dazu besitzt jeder Security-Representative ein eigenes individuelles Schlüsselpaar. Dieses wurde vom Signature-Server generiert, zertifiziert und dann dem Security-Representative übergeben. Der Signature-Server kann damit jederzeit feststellen, welcher Security-Representative gerade Kontakt zu ihm aufgenommen hat. Nach dem Erhalt der Daten signiert sie der Signature-Server auf die bekannte Weise (siehe Abschnitt 3.4) und sendet sie anschließend an den Security-Representative zurück. Dieser kann sie nun auf dem neuen Rechner installieren.

Die Administrierung des Signature-Servers wird von einem Security-Administrator vorgenommen. Auch der Security-Administrator besitzt ein individuelles Schlüsselpaar, so daß er auf sichere Weise mit dem Signature-Server kommunizieren kann.

Die genaue Struktur des Signature-Servers mit allen beteiligten Personen, Security-Administrator und Security-Representatives, zeigt Abbildung 4.4. Ein neuer Rechner H soll in ein laufendes AutO-System integrieren werden. Der Security-Representative B untersucht zuerst die Korrektheit des Rechners und der benutzten Java-Klassen. Dann errichtet er eine sichere Verbindung (mit Hilfe eines AutO-Sockets) zum Signature-Server. Dazu benötigt er neben seinem eigenen Schlüsselpaar auch den öffentlichen Schlüssel des Signature-Servers (genaueres dazu ist in der Beschreibung des AutO-Sockets, Abschnitt 4.3, zu finden). Über diese Verbindung überträgt er den öffentlichen Schlüssel von H, den er zuvor zusammen mit dem geheimen Schlüssel für H generiert hat. Der Signature-Server zertifiziert den Schlüssel und sendet ihn zurück an B. Nun kann der Security-Representative das Schlüsselpaar für H sowie den öffentlichen Schlüssel des Signature-Server auf dem neuen Rechner installieren.

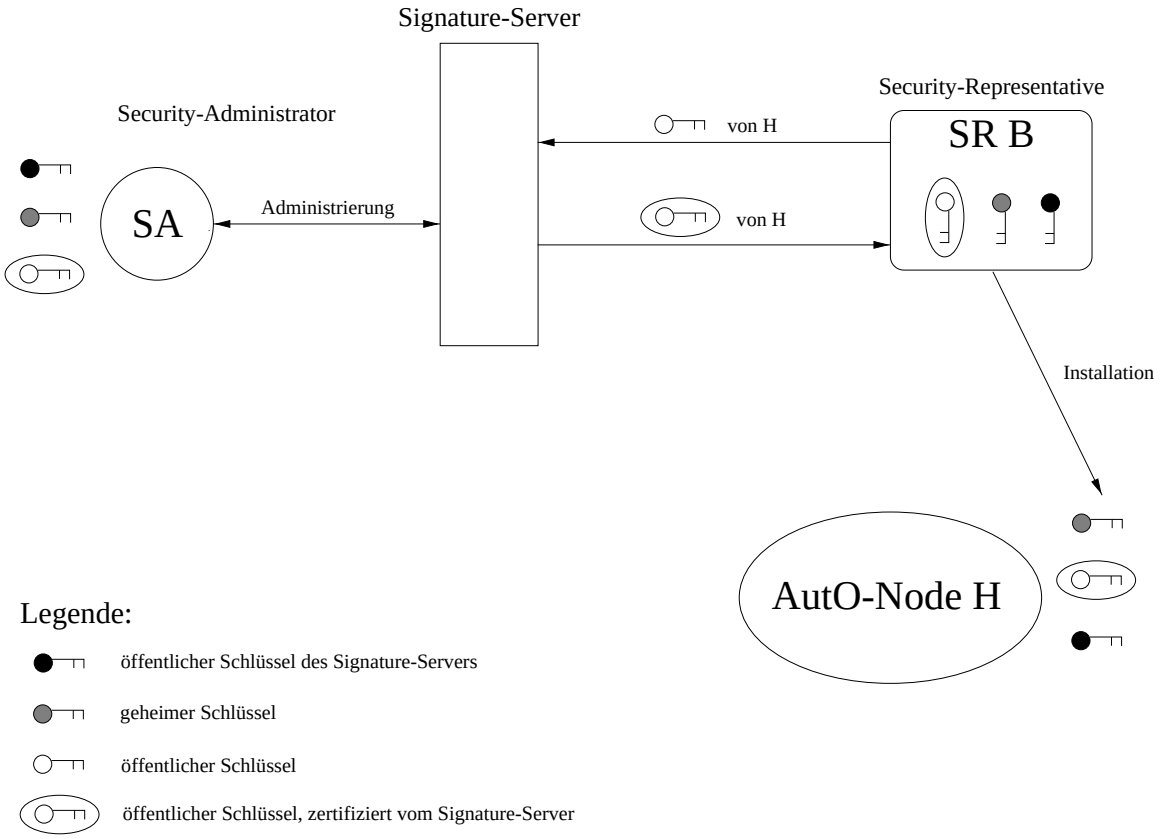


Abbildung 4.4: Die Struktur des Signature-Servers

Neben dem Zertifizieren von öffentlichen Schlüsseln dient der Signature-Server auch dazu, Sicherheitsklassen, die für eine sichere Migration benötigt werden, sowie allgemeine Benutzerklassen zu signieren. Genauere Informationen dazu finden sich in [Sel99].

4.5 Die Kommunikation zwischen Node-Managern

In Abschnitt 4.1 wurde untersucht, welche Verbindungstypen in AutO vor Angriffen geschützt werden müssen. Das Prinzip, wie die Sicherheit einer Verbindung gewährleistet werden kann, ist dabei bei allen Typen gleich. Deshalb beschränkt sich diese Diplomarbeit darauf, exemplarisch die Einbindung des in Abschnitt 4.3 vorgestellten AutO-Sockets in die Kommunikation zwischen verschiedenen AutO-Node-Managern zu erläutern.

Jeder Node-Manager benötigt ein Schlüsselpaar $(K_g, K_{\ddot{o}})$. Nachdem der öffentliche Schlüssel beim Signature-Server zertifiziert wurde, kann der neue Node-Manager mit anderen kommunizieren. Dazu werden die Verbindungen zwischen einzelnen Node-Managern nun nicht mehr über normale Java-Sockets, sondern über AutO-Sockets hergestellt.

Berechnung des MAC-Wertes einer Nachricht

Wie schon in Kapitel 3 erwähnt, dient ein MAC hauptsächlich dazu, die Identität des Absenders und die Unverfälschtheit einer Nachricht festzustellen. Ein Node-Manager hat deshalb die Möglichkeit, jede Nachricht M, die er abschickt, mit einem MAC zu versehen. Für die Berechnung des MAC-Wertes wird dabei neben der Nachricht auch der Session-Key der aktuellen Verbindung verwendet (S repräsentiert den Algorithmus, der den MAC-Wert berechnet):

$$C = S_K(M)$$

Verifizieren des MAC-Wertes einer Nachrichten

Erreicht eine mit einem MAC versehene Nachricht C einen Node-Manager, wird der MAC verifiziert:

$$M = V_K(C)$$

Verschlüsseln von Nachrichten

In den meisten Fällen reicht es aus, versendete Nachrichten mit einem MAC zu versehen. Enthält eine Nachricht jedoch kritische Daten, wie es etwa bei `OnGuardMessage`- und `OperationAckMessage`-Nachrichten der Fall ist, sollte diese zusätzlich verschlüsselt werden. Dazu wird eine Nachricht M, die mit einem MAC versehen sein kann oder nicht, mit dem aktuellen Session-Key der Verbindung, über die die Nachricht übertragen werden soll, verschlüsselt:

$$C = E_K(M)$$

Entschlüsseln von Nachrichten

Das Entschlüsseln einer Nachricht geschieht mit Hilfe des zuständigen Session-Keys:

$$M = D_K(M)$$

4.5.1 System- und Benutzernachrichten

AutO verwendet eine Vielzahl von verschiedenen Nachrichtentypen. Jede wird durch eine Subklasse von `common.messages.Message` repräsentiert. Man könnte für jeden einzelnen Typ definieren, ob er beim Senden verschlüsselt bzw. mit einem MAC versehen werden muß und beim Empfang verschlüsselt bzw. mit einem MAC versehen sein soll. Diese Aufteilung wäre aber zu fein. Zudem wäre es zu komplex für einen Programmierer oder Administrator, für jeden einzelnen Nachrichtentyp diese Einstellungen zu treffen. Deshalb wird im folgenden nur zwischen zwei Arten von Nachrichten unterschieden:

- Nachrichten, die Benutzerdaten enthalten, also Daten eines autonomen Objekts oder einer Transaktion, werden *Benutzernachrichten* genannt. Darunter fallen `OnGuardMessage`-Nachrichten, die ein autonomes Objekt oder eine Transaktion erhalten und absenden kann, sowie `OperationAckMessage`-Nachrichten, die die Antwort eines Guards enthalten und dem Aufrufer zurückliefern. Zusätzlich können auch `AtGuardMessage`-Nachrichten Benutzerdaten enthalten. Auch sie zählen deshalb zu den Benutzernachrichten.

- *Systemnachrichten* werden alle restlichen Nachrichten des AutO-Systems genannt. Auf sie hat ein Objekt bzw. der Programmierer des Objekts keinen direkten Einfluß, da sie intern für den Betrieb von AutO genutzt werden.

Der Begriff *Chiffrenstatus* wird im folgenden sowohl im Zusammenhang mit ausgehenden als auch mit eingehenden Nachrichten verwendet. Bei einer ausgehenden Nachricht legt er fest, ob sie verschlüsselt bzw. mit einem MAC versehen *werden soll*. Bei einer eingehenden Nachricht definiert er, ob sie verschlüsselt bzw. mit einem MAC versehen *sein muß*.

Systemnachrichten

Da Systemnachrichten interne Nachrichten des AutO-Systems sind, und der Programmierer eines Objekts damit keinen direkten Einfluß auf sie hat, werden die oben erwähnten Einstellungen global für alle Objekte eines Node-Managers getroffen. Dies geschieht in der Datei `AutO.config`. Dort werden die Chiffrenstati von ausgehenden und eingehenden Systemnachrichten angegeben.

Benutzernachrichten

Die verschiedenen Möglichkeiten, wie man die Chiffrenstati von Benutzernachrichten festlegen kann, werden im folgenden Abschnitt ausführlicher beschrieben.

4.5.2 Versand und Empfang von Nachrichten in Objekten

Sowohl bei autonomen Objekten als auch bei Transaktionen muß man definieren,

- ob zu sendende Benutzernachrichten
  - verschlüsselt bzw.
  - mit einem MAC versehen werden sollen, und
- ob empfangene Benutzernachrichten
  - verschlüsselt bzw.
  - mit einem MAC versehen sein müssen.

Für jeden der vier angeführten Fälle gibt es verschiedene Möglichkeiten, wie man die entsprechende Einstellung treffen kann:

- *Node-Manager-bezogen*: Die Einstellung wird global für einen kompletten AutO-Node getroffen. Alle Objekte des Rechners benutzen damit die gleiche Einstellung. Dem Programmierer eines Objekts ist es nicht möglich, diese Einstellung zu beeinflussen. Eine derartige Lösung ist relativ unflexibel, da es durchaus `OnGuardMessage`-Nachrichten geben kann, deren Parameter beim Absenden nicht verschlüsselt sein müssen, weil sie einfach zu unwichtig sind, z. B. die Telefonnummer einer Person, während dasselbe Objekt in einem anderen Guard die Kontonummer dieser Person versendet, welche auf jeden Fall verschlüsselt sein sollte.

- *Objekt-bezogen*: Hierbei wird die Einstellung global für alle entsprechenden Nachrichten eines Objekts getroffen. Auch diese Lösung ist immer noch relativ unflexibel, da ein Objekt Nachrichten sehr unterschiedlicher Wichtigkeit verschickt.

- *`sendMessage`-bezogen*: Bei jedem Aufruf der `sendMessage`-Methode - nur damit kann man Benutzernachrichten versenden - wird angegeben, welche Einstellungen für die zu sendende Nachricht und deren Antwort gelten sollen.

Bei autonomen Objekten allerdings reicht diese dritte Möglichkeit alleine nicht aus. Autonome Objekte können auch mit Hilfe von Guards Nachrichten empfangen und senden. Man muß diese dritte Möglichkeit deshalb mit einer weiteren kombinieren. Deshalb existiert bei autonomen Objekten zusätzlich eine vierte Möglichkeit, die Einstellung zu treffen:

- *Guard-bezogen*: Die Einstellung wird für jeden Guard getrennt getroffen.

Eine Kombination der einzelnen Einstellmöglichkeiten ist denkbar, allerdings erfordert dies entsprechende Prioritätsregeln, um mögliche Konflikte zu regeln.

Weiterhin muß entschieden werden, ob die Angabe jeder Einstellung immer nötig ist, oder ob gewisse Default-Einstellungen sinnvoll sind. Dabei wäre denkbar, daß diese in der `AutO.config`-Datei für einen ganzen Rechner festgelegt sind. Besser erscheint aber eine Lösung der Art, daß eine fehlende Angabe einer Einstellung bedeutet, daß eine Nachricht je nach fehlender Einstellung verschlüsselt oder mit einem MAC versehen wird bzw. verschlüsselt oder mit einem MAC versehen sein muß.

Diese zweite Möglichkeit wurde auch im Rahmen dieser Diplomarbeit implementiert, d. h. wird für eine Nachricht kein Chiffrenstatus festgelegt, wie in den folgenden beiden Abschnitten beschrieben, dann wird ein *Default-Chiffrenstatus* verwendet. Er legt fest, daß ausgehende Nachrichten immer verschlüsselt und mit einem MAC versehen werden und eingehende Nachrichten ebenfalls immer verschlüsselt und mit einem MAC versehen sein müssen.

4.5.3 Einstellmöglichkeiten bei Transaktionen

Für Transaktionen stehen folgende Einstellmöglichkeiten zur Verfügung:

- Die Einstellung, ob Nachrichten mit einem MAC versehen gesendet werden bzw. ob ankommende Nachrichten mit einem MAC versehen sein müssen, wird Objekt-bezogen getroffen.

  Um für eine Transaktion die entsprechende Einstellung zu treffen, muß bei der Transaktionsklasse, die von `Transaction` abgeleitet sein muß, der passende Konstruktor der Klasse `Transaction` aufgerufen werden:

  ```
  public Transaction(String typename, boolean outgoing_mac,
                     boolean incoming_mac)
  ```

  Der Parameter `outgoing_mac` spezifiziert, ob zu sendende Nachrichten mit einem MAC versehen werden. Analog definiert der Parameter `incoming_mac`, ob empfangene Nachrichten mit einem MAC versehen sein müssen.

- Die Einstellung, ob eine zu sendende Nachricht verschlüsselt wird bzw. eine eingetroffene Nachricht verschlüsselt sein muß, wird `sendMessage`-bezogen getroffen.

  Dies wird festgelegt, wenn im Transaktionscode die Methode `sendMessage` aufgerufen wird:

  ```
  public final boolean sendMessage(Oid receiver, String guard,
                                   ArrayList params,
                                   boolean outgoing_encryption,
                                   boolean incoming_encryption)
  ```

  Diese überladene Methode bietet unter anderem die Möglichkeit, anzugeben, ob ausgehende Nachrichten verschlüsselt werden sollen (`outgoing_encryption`) und ob empfangene Nachrichten verschlüsselt sein müssen (`incoming_encryption`).

In beiden Fällen, also sowohl bei der MAC-Berechnung als auch bei der Verschlüsselung, gilt, daß eine `security.connection.CipherStateException` geworfen wird, falls eine Nachricht, die als Antwort auf einen `Transaction.sendMessage`-Aufruf erhalten wurde, nicht den angegebenen Anforderungen entspricht. In einem solchen Fall sollte die Transaktion abgebrochen werden.

Damit ist für Transaktionen ein ausreichender Grad von Flexibilität bezüglich der Festlegung von Chiffrenstati gewährleistet.

4.5.4 Einstellmöglichkeiten bei autonomen Objekten

Bei autonomen Objekten existieren folgende Einstellmöglichkeiten:

- Die Einstellung, ob Nachrichten mit einem MAC versehen gesendet werden sollen bzw. ob ankommende Nachrichten mit einem MAC versehen sein müssen, wird Guard-bezogen getroffen.

- Die Einstellung, ob eine zu sendende Nachricht verschlüsselt wird bzw. eine eingetroffene Nachricht verschlüsselt sein muß, wird ebenso Guard-bezogen getroffen.

Für jeden Guard müssen also die entsprechenden Chiffrenstati festgelegt werden. Dies geschieht in einer Datenstruktur, die *Kryptographie-Matrix* genannt wird. Jedes autonome Objekt enthält eine solche Matrix, die festlegt, wie ankommende oder ausgehende Guard-Nachrichten behandelt werden sollen. Jede Kryptographie-Matrix enthält allerdings nur Informationen über diejenigen Guards, die auch im zur Matrix gehörenden Objekt deklariert wurden. Um also auch Zugriff auf die Informationen einer Basisklasse zu haben, ist die Kryptographie-Matrix eines autonomen Objekts von der Matrix des Basisobjekts abzuleiten. Abbildung 4.5 zeigt ein Beispiel für die Ableitung von autonomen Objekten und die analoge Ableitung der dazugehörigen Kryptographie-Matrizen.

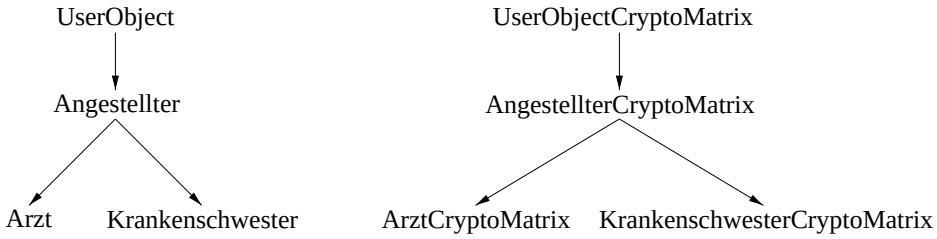


Abbildung 4.5: Zusammenhang zwischen der Ableitungshierarchie von autonomen Objekten und ihren Kryptographie-Matrizen

Jede Kryptographie-Matrix enthält für alle im dazu gehörenden autonomen Objekt definierten Guards folgende Einstellungen:

- den Chiffrenstatus der Nachricht, die einen Guard auslöst (nur bei On-Guards),
- den Chiffrenstatus der Antwortnachricht, die nach der Abarbeitung des Guards zurückgesendet wird (nur bei On-Guards),
- den Chiffrenstatus von Nachrichten, die in einer Guard-Aktion versendet werden, und
- den Chiffrenstatus von Antwortnachrichten, die in einer Guard-Aktion empfangen werden.

Ein Beispiel für eine derartige Kryptographie-Matrix zeigt Abbildung 4.6. Eine Beschreibung des Hilfsprogramms *CryptographyMatrixGenerator*, das genutzt werden kann, um für ein autonomes Objekt eine derartige individuelle Kryptographie-Matrix zu erstellen, ist in Anhang D.2 zu finden.

4.6 Die Sicherung der verbleibenden Verbindungen

Neben den Verbindungen zwischen einzelnen Node-Managern müssen auch die restlichen, in Abschnitt 4.1 angesprochenen Verbindungen gesichert werden. Die Konfiguration, wel-

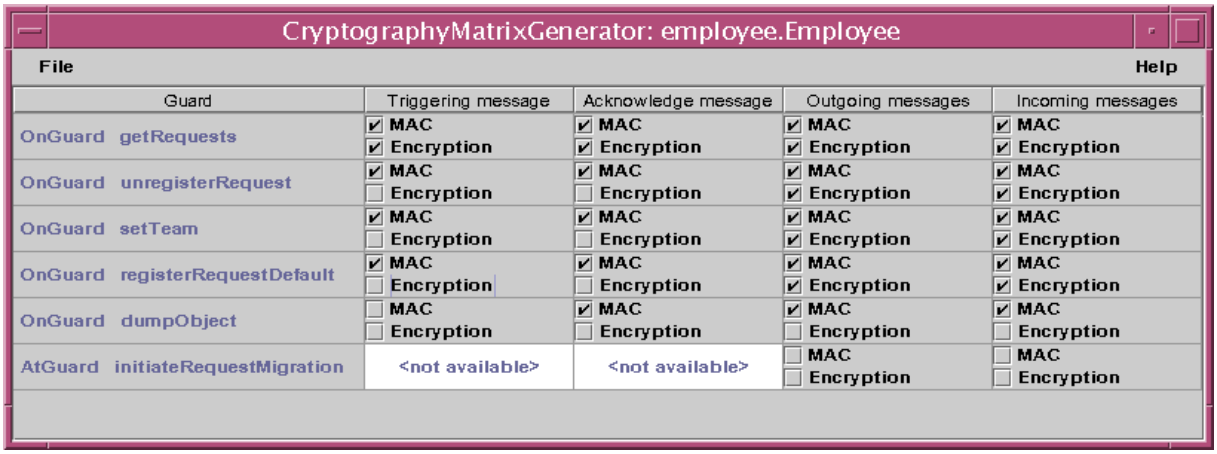


Abbildung 4.6: Ein Beispiel für eine Kryptographie-Matrix

che Nachrichten verschlüsselt bzw. mit einem MAC versehen werden oder sein müssen, wird dabei stets in der Dertei `AutO.config` (siehe Anhang E) definiert.

Node-Manager $\longleftrightarrow$ AutO-Prozeß

Auf jedem Rechner wird ein Schlüsselpaar generiert, das AutO-Prozesse verwenden, wenn sie eine Verbindung (via AutO-Socket) zu ihrem Node-Manager aufbauen. Der öffentliche Schlüssel wird dabei mit dem geheimen Schlüssel des entsprechenden Node-Managers zertifiziert. Dadurch kann beim Verbindungsaufbau garantiert werden, daß sich nur ein lokaler AutO-Prozeß mit einem Node-Manager verbinden kann.

Werden Daten über eine solche Verbindung gesendet, ist keine Verschlüsselung oder MAC-Berechnung nötig, da der Node-Manager und die mit ihm verbundenen AutO-Prozesse auf demselben Rechner laufen und Daten damit nicht über ein Netzwerk übertragen werden.

Node-Manager $\longleftrightarrow$ Name-Service

Wie auch jeder Node-Manager erhält ein Name-Service ein individuelles Schlüsselpaar, durch das er sicher mit Node-Managern kommunizieren kann.

Verbindungen zu einem DBMS

AutO kommuniziert mit DBMSen über JDBC bzw. über eine TCP/IP-Verbindung bei AutOShore. Da bei der Benutzung von JDBC kein direkter Zugriff auf die Netzwerkverbindung möglich ist, müßten Daten entweder verschlüsselt werden, bevor sie an die JDBC-Schnittstelle weitergereicht werden, oder aber JDBC selbst müßte die sichere Kommunikation gewährleisten. Im ersten Fall wäre es nötig, daß sowohl das DBMS als auch AutO dieselben Algorithmen verwenden. Damit wäre aber die JDBC-Schnittstelle von

AutO nicht mehr unabhängig vom DBMS, so daß diese Lösung entfällt. Der andere Ansatz, daß JDBC selbst die Daten ver- und entschlüsselt, ist nur von den entsprechenden Softwarefirmen, die die JDBC-Treiber für ihre Datenbanken entwickeln, sowie von Sun, die die JDBC-Schnittstelle definiert, zu implementieren.

Um die Verbindung mit AutOShore zu sichern, wäre es nötig, die auf der AutO-Seite verwendeten Kryptographiealgorithmen in AutOShore zu implementieren. Im Rahmen dieser Diplomarbeit wurde allerdings darauf verzichtet, da es sich dabei nur um reine Implementierungsarbeit handelt, die zum einen keine neuen Erkenntnisse liefert und andererseits auch unverhältnismäßig viel Zeit kosten würde.

AutO-Prozeß (Terminal-Server) $\longleftrightarrow$ Terminal-Client

Neben der Sicherung der Verbindung zwischen Terminal-Server und Terminal-Client muß auch dafür Sorge getragen werden, daß nur autorisierte Benutzer eine Verbindung zu einem Terminal-Server herstellen können.

Jeder Benutzer, dem eine Verbindung zu einem Terminal-Server auf einem AutO-Node erlaubt werden soll, erhält ein eigenes Schlüsselpaar $K_{\ddot{o}}$ und K_g, das er beim Start eines Terminal-Clients mitangeben muß. Der öffentliche Schlüssel $K_{\ddot{o}}$ wird mit dem lokalen, vom Signature-Server zertifizierten öffentlichen Node-Manager-Schlüssel K_{Node} zertifiziert:

$$K_{\ddot{o}} \text{ zertifiziert mit } K_{Node} : Z(K_{\ddot{o}}, K_{Node})$$

Dadurch umfaßt die Zertifikatkette des öffentlichen Benutzerschlüssels nun zwei Einträge: den öffentlichen Benutzerschlüssel, zertifiziert mit dem öffentlichen Node-Manager-Schlüssel, und den öffentlichen Node-Manager-Schlüssel, zertifiziert vom Signature-Server.

$$\text{Zertifikatkette} : [Z(K_{\ddot{o}}, K_{Node}); Z(K_{Node}, K_{Signature-Server})] = [Z_1; Z_2]$$

Ein Benutzer muß nur einmal ein Schlüsselpaar beantragen, auch wenn er Zugang zur mehreren Terminal-Servern auf verschiedenen AutO-Nodes erhalten will. Denn statt auf jedem Rechner ein eigenes Schlüsselpaar zu generieren, kann die Zertifikatkette des Benutzers installiert werden. Der Rechner kann die Echtheit der Zertifikatkette prüfen, indem er zuerst das Zertifikat $Z(K_{Node}, K_{Signature-Server})$ verifiziert, das den öffentlichen Schlüssel desjenigen Node-Managers enthält, der die Benutzerschlüssel generiert hat. Dazu benötigt er den öffentlichen Schlüssel des Signature-Servers. Nachdem dieser Node-Manager-Schlüssel für echt befunden wurde, kann damit das Zertifikat $Z(K_{\ddot{o}}, K_{Node})$ mit dem öffentlichen Benutzerschlüssel verifiziert werden.

$$\begin{array}{lllll} \text{Verifiziere } Z_2 : & K_{Signature-Server} & \rightarrow & Z(K_{Node}, K_{Signature-Server}) & \Rightarrow \quad K_{Node} \\ \text{Verifiziere } Z_1 : & K_{Node} & \rightarrow & Z(K_{\ddot{o}}, K_{Node}) & \Rightarrow \quad K_{\ddot{o}} \end{array}$$

Mit diesen Verfahren kann also einerseits festgestellt werden, ob ein Benutzer, der eine Verbindung zu einem Terminal-Server aufbauen will, dazu autorisiert ist - er ist es genau dann, wenn sein öffentlicher Schlüssel dem Node-Manager des Terminal-Servers bekannt ist - andererseits kann damit über einen AutO-Socket auch eine sichere Verbindung aufgebaut werden.

Kapitel 5

Autorisierung

Kapitel 4 erklärte, wie ein verteiltes System wie AutO vor illegalen Zugriffen von außen geschützt werden kann. Doch ein Sicherheitssystem kann sich nicht nur darauf beschränken, da auch der Zugriff auf Daten von innen, d. h. durch Benutzer des Systems, gewissen Einschränkungen unterliegt.

Angenommen eine Bank ermöglicht es ihren Kunden, die Konten online, etwa via Internet, zu verwalten. Ein Kunde soll die Möglichkeit besitzen, seinen Kontostand einzusehen, Geld von seinem Konto auf ein anderes zu überweisen, etc. Allerdings hat er nur die Erlaubnis, auf *sein* Konto zuzugreifen, d. h. der Zugriff auf das Konto eines anderen Kunden ist ihm verwehrt. Ansonsten könnte er beliebige Manipulationen an den Konten anderer Kunden vornehmen. Obwohl dieser Kunde also legal im System, der Online-Kontenverwaltung, angemeldet ist, hat er nicht das Recht, *alles* zu tun. Seine Möglichkeiten unterliegen gewissen Einschränkungen. Welche das sind, muß in einer unternehmensspezifischen Sicherheitspolitik festgelegt werden.

5.1 Sicherheitspolitik

Jedes Unternehmen hat bestimmte Vorstellungen davon, welche Informationen ein bestimmter Mitarbeiter lesen, verändern, kopieren, usw. darf. Darunter fallen nicht nur Informationen in Computern, sondern allgemein alle Daten, die in einem Unternehmen im Umlauf sind. Diese Menge an Richtlinien und die zugrundeliegenden Ideen werden mit dem Begriff *Sicherheitspolitik* zusammengefaßt. Eine Sicherheitspolitik definiert die Prinzipien, nach denen Zugriff auf Informationen gewährt wird [CFMS95, DTPH97, DTH97, TS94].

Die Ausprägungen einer solchen Sicherheitspolitik sind *Autorisierungsregeln*, d. h. Regeln, die genau festlegen, wann jemand, *Subjekt* genannt, autorisiert ist, auf bestimmte Daten, *Objekte*[1] genannt, zuzugreifen. Man nennt diese Regeln auch *Zugriffsregeln* und den Vorgang der Überprüfung, ob einem Subjekt der Zugriff auf ein Objekt erlaubt ist, *Zugriffskontrolle*.

[1] Diese Objekte haben nichts mit den autonomen Objekten von AutO gemein und dürfen nicht mit ihnen verwechselt werden.

5.2 Zugriffskontrolle

Spricht man im Zusammenhang mit Sicherheit, vor allem im Bereich der Informatik, von Zugriff, so wird darunter die Fähigkeit verstanden, mit einer Ressource (etwa einer Datei, einem Verzeichnis, einem Tupel aus einer Datenbank) etwas *zu tun* (z.B. lesen, modifizieren, benutzen). Zugriffskontrolle bezeichnet dann die Art und Weise wie diese Fähigkeit gewährt oder verwehrt wird. Wird sie durch einen Computer unterstützt, etwa durch eine entsprechende Softwarekomponente, kann man damit nicht nur feststellen, wer Zugriff auf welche Ressource hat, sondern auch welcher Art der Zugriff ist [NIS95].

Das Department of Defense der USA hat sich über lange Zeit mit der Entwicklung und Definition von Sicherheitskriterien beschäftigt. Zusammengefaßt wurden diese Erkenntnisse in dem Standard *Trusted Computer System Evaluation Criteria* [U.S85]. Darin werden die verschiedenen Arten von Zugriffskontrollen in zwei Hauptgruppen aufgeteilt:

5.2.1 Mandatory Access Control

Mandatory Access Control (MAC)[2] wird in [U.S85] wie folgt definiert:

> Ein Mittel, um den Zugriff auf Objekte einzuschränken, basierend auf der Sensitivität (dargestellt durch einen Label) der in den Objekten enthaltenen Informationen und der formalen Autorisation, d. h. Freigabe, von Subjekten, auf Informationen von solcher Sensitivität zuzugreifen.

Dies bedeutet, daß jedem Subjekt, einer Person, einem Programm, etc., eine bestimmte Freigabe zugeordnet wird. Jedes Objekt, z. B. eine Datei, wird mit einer bestimmten Klassifikation versehen, die die Sensitivität der Daten ausdrückt. Versucht ein Subjekt auf ein Objekt zuzugreifen, werden Freigabe und Klassifikation verglichen. Fällt der Vergleich positiv aus, kann der Zugriff gewährt werden. Ansonsten wird er abgelehnt.

Beispiel: Seien *Mickey Mouse* und *Goofy* zwei Subjekte mit den Freigaben 5 bzw. 3.

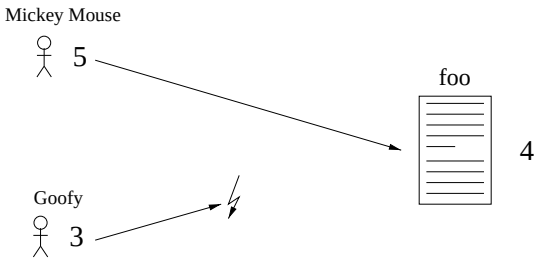


Abbildung 5.1: Beispiel: Mandatory Access Control

Beide wollen auf eine Datei mit Namen `foo` zugreifen, die die Klassifikation 4 besitzt. Die

[2] Die Abkürzung MAC wird nun für Mandatory Access Control verwendet, nicht mehr wie bisher für Message Authentication Code. Im Zweifelsfalle sollte es jedoch immer aus dem Kontext klar sein, welcher Begriff mit MAC abgekürzt wird.

Beispiel-Zugriffskontrolle gewährt Zugriff auf eine Ressource nur dann, wenn die Freigabe des Subjekts größer oder gleich der Klassifikation des Objekts ist, auf das zugegriffen werden soll. Es wird also die Freigabe von *Mickey Mouse* mit der Klassifikation von `foo` verglichen. Da 5 größer ist als 4, kann *Mickey Mouse* auf die Datei zugreifen, sie etwa lesen. Anders liegt der Fall bei *Goofy*. Seine Freigabe beträgt nur 3 und ist damit kleiner als 4. Er erhält keine Erlaubnis, auf die Datei `foo` zuzugreifen.

5.2.2 Discretionary Access Control

Discretionary Access Control, kurz DAC, bedeutet [U.S85]:

> Ein Mittel, um den Zugriff auf Objekte einzuschränken, basierend auf der Identität von Subjekten und/oder Gruppen, zu denen sie gehören. Die Kontrollen sind frei in dem Sinne, daß ein Subjekt mit einer bestimmten Zugriffserlaubnis diese an jedes andere Subjekt weitergeben kann (möglicherweise auch indirekt).

Bei DAC wird für jedes Subjekt und jedes Objekt einzeln spezifiziert, welche Zugriffsarten das Subjekt auf das Objekt ausüben darf. Darunter fällt auch die Weitergabe von Zugriffsrechten, d. h. ein Subjekt kann einem anderen Subjekt explizit Zugriff auf Objekte gewähren, für die es diese Erlaubnis (der Weitergabe von Zugriffsrechten) besitzt.

Beispiel: Wie vorher bestehe die Subjektmenge aus *Mickey Mouse* und *Goofy*. Beide

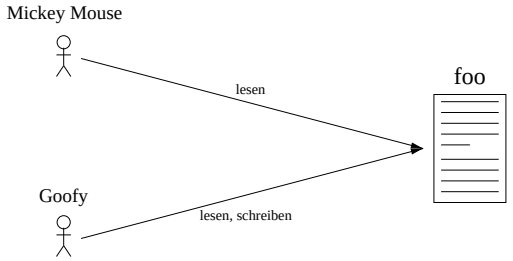


Abbildung 5.2: Beispiel: Discretionary Access Control

besitzen die in Abbildung 5.2 dargestellten Rechte, d. h. *Mickey Mouse* kann die Datei `foo` lesen, *Goofy* kann sowohl lesend als auch schreibend auf sie zugreifen. Ein Zugriff auf ein Objekt ist nur erlaubt, wenn explizit das dazu notwendige Recht existiert. Damit ist es *Mickey Mouse* verwehrt, schreibend auf die Datei `foo` zuzugreifen. *Goofy* hingegen wird die Erlaubnis dazu erhalten.

5.3 Die rollenbasierte Zugriffskontrolle

Ein Zugriffskontrollmodell, das sich nicht in die beiden oben genannten Hauptkategorien einordnen läßt, ist die *rollenbasierte Zugriffskontrolle*, auch *role-based access control*

(RBAC) genannt [FK92, SCFY96, FGK95, MD94, SM94]. Bei ihr unterscheidet man drei Mengen:

- **Users**: Diese Menge enthält die legalen Benutzer des Systems. Jedes Element aus dieser Menge, d. h. jeder Benutzer, repräsentiert eine reale Person, die Aktionen in einem System initiieren kann.

  Für die Sicherheit eines Systems ist es unabdingbar, daß die Identität eines Benutzers zweifelsfrei festgestellt werden kann. Es darf nicht möglich sein, sich unter falschem Namen im System anzumelden. Man nennt den Vorgang der Identifizierung eines Benutzers *Authentifizierung*. Bei dem Betriebssystem Unix geschieht dies etwa dadurch, daß ein Benutzer sich explizit anmelden muß, unter Angabe von Kennung und (dem nur ihm bekannten) passenden Paßwort.

- **Roles**: Eine Rolle repräsentiert eine bestimmte Funktion in einem Unternehmen, die ein Benutzer wahrnehmen kann. Damit ist die Menge der Rollen hochgradig unternehmensabhängig. Außerdem bedarf es einer genauen Analyse eines Unternehmens, um alle Rollen und die Beziehungen, die zwischen diesen bestehen, festzustellen. In einer Bank kann man unter anderem die Rollen *Filialleiter* und *Kassierer* erkennen. Auch Beziehungen zwischen einzelnen Rollen, die z. B. hierarchisch angeordnet sein können, müssen erkannt und entsprechend berücksichtigt werden.

- **Permissions**: Permissions repräsentieren die Rechte in einem System, die einzelne Rollen erhalten können. Auch sie sind, wie Rollen, in starkem Maße unternehmensspezifisch. *Betrachte die Monatsstatistiken* oder *Zahle Geld aus* wären einige Aktionen für das vorherige Bankbeispiel. Permissions können vereinfacht mit den Rechten bei einem DAC-Modell verglichen werden.

  In RBAC können sowohl positive als auch negative Permissions verwendet werden.

Die Mengen **Users** und **Roles** sowie **Roles** und **Permissions** werden durch Relationen miteinander in Beziehung gesetzt. Es gilt: $\mathbf{UR} \subseteq \mathbf{Users} \times \mathbf{Roles}$ und $\mathbf{RP} \subseteq \mathbf{Roles} \times \mathbf{Permissions}$, d. h. die Relation **UR** verbindet die Mengen **Users** und **Roles**, die Relation **RP** die Mengen **Roles** und **Permissions**.

Beide Relationen stellen n:m Beziehungen dar. Es kann also jeder Benutzer mehreren Rollen zugewiesen sein, je nachdem welche Funktion er aktuell in einem Unternehmen einnimmt. Ebenso können einer einzelnen Rolle mehrere Benutzer zugeordnet sein. Die weiter oben genannte Rolle *Kassierer* wäre ein Beispiel dafür, da in einer Bank stets mehrere Angestellte dafür eingesetzt werden. Analog kann eine bestimmte Rolle mehreren Permissions zugeordnet sein, genauso wie eine Permission mehreren Rollen zugeordnet sein kann. Abbildung 5.3 zeigt diese Zusammenhänge.

5.3.1 Unterschiede zu MAC und DAC

MAC und DAC besitzen einige Nachteile bzw. Einschränkungen:

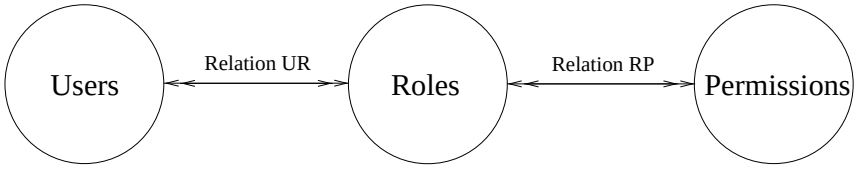


Abbildung 5.3: Die Beziehung zwischen Benutzern, Rollen und Permissions

- Bei MAC wird jedem Subjekt und jedem Objekt eine Sicherheitsstufe, Freigabe bzw. Klassifikation genannt, zugewiesen. In Unternehmen oder allgemein in nichtmilitärischen Systemen ist diese Sichtweise auf Informationen kaum zu finden. Deshalb wird MAC in solchen Systemen selten eingesetzt.
- DAC erlaubt es einem Benutzer, Rechte an Objekten, deren Besitzer er ist, weiterzugeben. Diese dezentrale Administration der Rechte ist in einem Unternehmen nicht erwünscht, denn die Möglichkeit, daß ein Benutzer Rechte an unbefugte Personen weitergibt, kann auf diese Weise nicht ausgeschlossen werden.

RBAC bietet einen Ansatz, mit dem diese Probleme umgangen werden können. Die rollenbasierte Zugriffskontrolle ist formal weder *discretionary* noch *mandatory*. Erst die konkrete Implementierung entscheidet, in welche Richtung sie tendiert. Dadurch bietet RBAC einige wesentliche Vorteile:

- Die Administrierung eines Sicherheitssystems, z. B. der Autorisierungsregeln, stellt immer eine Schwierigkeit dar, da sie grundsätzlich mit einem nicht zu unterschätzenden Aufwand verbunden ist. Oft ist es erforderlich, das System immer wieder an sich ändernde Gegebenheiten, etwa eine Änderung der Unternehmensstruktur, anzupassen. RBAC unterstützt die dynamische Administrierung eines Systems, indem je nach aktueller Aufgabe ein Benutzer zu Rollen hinzugefügt bzw. seine Zugehörigkeit rückgängig gemacht werden kann. Außerdem kann die Zuordnung von Permissions zu Rollen dynamisch an die aktuellen Unternehmensfunktionen, d. h. die verschiedenen Arbeitsgänge, die sich in einem Unternehmen unterscheiden lassen, angepaßt werden
- Bei RBAC vollzieht sich die Administrierung auf einer abstrakten Ebene, die der Sichtweise auf ein Unternehmen entspricht. Es ist möglich, beim Aufbau des Sicherheitssystems Rollen und Permissions entsprechend der Unternehmensstruktur zu definieren. Diese wird sich während der Laufzeit nur minimal ändern. Dynamisch muß zur Laufzeit nur die Zugehörigkeit einzelner Benutzer zu bestimmten Rollen angepaßt werden, um etwa die wechselnden Aufgaben- und Arbeitsbereiche eines Benutzers widerzuspiegeln.
- Gerade in einem verteilten System gestaltet sich die Verwaltung eines Sicherheitssystems oft schwierig. RBAC bietet die Möglichkeit, dies effizient zu realisieren. Rollen, die global im ganzen System verfügbar sein müssen, können zentral verwaltet werden. Die Zugehörigkeit zu diesen Rollen kann dann lokal entsprechend den Gegebenheiten geregelt werden.

5.3.2 Designentscheidungen

Bisher existiert noch keine allgemein anerkannte formale Definition von RBAC, allerdings existieren in [SCFY96, FGK95] einige Vorschläge dazu. Im folgenden wird das Modell vorgestellt, das für die Implementierung in AutO gewählt wurde. Es entspricht zu großen Teilen dem Referenzmodell $RBAC_3$ aus [SCFY96], lehnt sich aber auch an [FGK95] an.

- Die Mengen **Users**, **Roles** und **Permissions** sind so, wie am Anfang von Abschnitt 5.3 definiert.
- Die Relation $\mathbf{UR} \subseteq \mathbf{Users} \times \mathbf{Roles}$ definiert, welchen Rollen ein Benutzer $u \in \mathbf{Users}$ zugeordnet ist sowie welche Benutzer Mitglied einer bestimmten Rolle $r \in \mathbf{Roles}$ sind. Folglich gilt:

$$\begin{aligned} \text{roles-of-user}(u) &:= \{r \in \mathbf{Roles} \mid (u, r) \in \mathbf{UR}\}, \\ \text{users-of-role}(r) &:= \{u \in \mathbf{Users} \mid (u, r) \in \mathbf{UR}\}. \end{aligned}$$

- Die Relation $\mathbf{RP} \subseteq \mathbf{Roles} \times \mathbf{Permissions}$ definiert die Beziehungen zwischen den Mengen **Roles** und **Permissions**. Dabei bestimmt

$$\text{roles-of-permission}(p) := \{r \in \mathbf{Roles} \mid (r, p) \in \mathbf{RP}\}$$

  die Rollen, die einer bestimmten Permission $p \in \mathbf{Permissions}$ zugeordnet sind. Analog definiert

$$\text{permissions-of-role}(r) := \{p \in \mathbf{Permissions} \mid (r, p) \in \mathbf{RP}\}$$

  die Permissions, die in Beziehung mit einer Rolle $r \in \mathbf{Roles}$ stehen.

- Die Menge $\mathbf{Sessions} \subseteq \mathbf{Users} \times \mathcal{P}(\mathbf{Roles})$[3] enthält alle aktiven Sitzungen der Benutzer. Jede Sitzung repräsentiert einen Benutzer. Ein Benutzer kann mehrere aktive Sitzungen gestartet haben. In einer Sitzung kann ein Benutzer zur Laufzeit eine oder mehrere Rollen, denen er zugeordnet ist, aktivieren und deaktivieren. Diese Menge der aktiven Rollen bestimmt die Rechte, die ein Benutzer besitzt.
- Die Struktur eines Unternehmens ist hierarchisch aufgebaut. Da Rollen diese Struktur repräsentieren sollen, muß diese Hierarchie auch in die Menge der Rollen modelliert werden. Ähnlich wie in objektorientierten Programmiersprachen mit Klassen ist es also möglich, Rollen von anderen Rollen abzuleiten. Eine abgeleitete Rolle erbt automatisch die Permissions, die der Basisrolle zugeordnet sind.
- Es muß möglich sein, bestimmte Randbedingungen an Rollen zu knüpfen. In einem Unternehmen wird es oft der Fall sein, daß eine bestimmte Rolle, etwa *Manager*, nur einer bestimmten Zahl von Benutzern zugeordnet sein darf. Außerdem kann

[3] $\mathcal{P}(M) := \{X \mid X \subseteq M\}$ wird als *Potenzmenge von M* bezeichnet.

es vorkommen, daß ein Benutzer nicht gleichzeitig zwei bestimmte Rollen aktiviert haben darf. Man denke an ein Unternehmen, in dem derselbe Benutzer Überweisungen genehmigen und ausführen darf. Es kann also der Fall eintreten, daß sich bestimmte Rollen gegenseitig ausschließen. Die Menge der Randbedingungen wird mit **Constraints** bezeichnet.

Sitzungen

Jeder Benutzer des Systems agiert in einer Sitzung, die ihm eindeutig zugeordnet ist. Ein Benutzer kann mehrere verschiedene Sitzungen zur gleichen Zeit gestartet haben.

Ist also $s \in$ **Sessions** eine Sitzung, so kann der dazugehörige Benutzer durch die Funktion

$$\text{user-of-session}(s) \;:=\; \pi_1(s)$$

festgestellt werden[4]. In einer Sitzung kann ein Benutzer nach Belieben eine oder mehrere Rollen aktivieren und deaktivieren. Sie werden dann zur Menge der aktiven Rollen hinzugefügt. Es gilt:

$$\text{active-roles}(s) \;:=\; \pi_2(s)$$

Es muß dabei allerdings folgende Konsistenzbedingung eingehalten werden:

Regel 1. (Konsistente Sitzung)
Ein Benutzer kann nur diejenigen Rollen aktivieren, denen er auch zugeordnet ist. Eine Sitzung s $\in$ **Sessions** ist also konsistent, falls

$$\text{active-roles}(s) \subseteq \text{roles-of-user}(u), \text{ wobei } u = \text{user-of-session}(s).$$

Die Menge der Permissions einer Sitzung $s \in$ **Sessions** läßt sich damit wie folgt feststellen:

$$\text{permissions-of-session}(s) = \bigcup_{r \in active-roles(s)} \text{permissions-of-role}(r)$$

Rollenhierarchien

Jedes Unternehmen ist hierarchisch geordnet. Verantwortlichkeiten und Privilegien können sich überlappen, oder aber es kann Aufgaben im Unternehmen geben, die von jedem Angestellten ausgeführt werden können. Für RBAC, bei dem die Struktur eines Unternehmens durch die Rollen modelliert wird, ist es deshalb nötig, diese Hierarchie wiedergeben zu können. Ähnlich wie Klassen in bekannten objektorientierten Sprachen wie C++ und Java können deshalb Rollen voneinander abgeleitet werden.

Abbildung 5.4 zeigt ein Beispiel[5]. Die beiden Rollen *Internist* und *Kardiologe* sind von

[4] $\pi_i(x)$ bezeichnet die Projektion auf die i-te Koordinate von x.
[5] Ein Pfeil zeigt stets von der abgeleiteten Rolle zur Basisrolle.

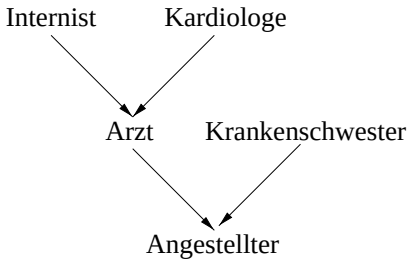


Abbildung 5.4: Ein Beispiel für hierarchische Rollen

Arzt abgeleitet und besitzen damit ebenfalls die Permissions, die der Rolle *Arzt* zugewiesen wurden.

Formal definiert die Ableitungshierarchie der Rollen eine Halbordnung $\geq$ auf der Menge der Rollen.

Definition 1. (Rollenhierarchie)
Die Menge $\mathbf{RH} \subseteq \mathbf{Roles} \times \mathbf{Roles}$ definiert eine Halbordnung auf der Menge der Rollen. Sie wird als Rollenhierarchie $\geq$ bezeichnet. Es gilt:

$$(r_1, r_2) \in \mathbf{RH} \Leftrightarrow r_1 \geq r_2.$$

Man sagt auch: Die Rolle r_1 ist abgeleitet von der Rolle r_2.

In dem vorher genannten Beispiel gilt: $\mathit{Internist} \geq \mathit{Arzt} \geq \mathit{Angestellter}$. Mit der Einführung einer Rollenhierarchie ist es allerdings nötig, die Regel der konsistenten Sitzung zu modifizieren:

Regel 2. (Erweiterte konsistente Sitzung)
Ein Benutzer kann nur diejenigen Rollen aktivieren, denen er auch zugeordnet ist oder aber solche, von denen die ihm zugeordneten Rollen abgeleitet sind. Eine Sitzung s $\in$ **Sessions** ist also konsistent, falls

$$\text{active-roles}(s) \subseteq \{r \in \mathbf{Roles} \mid (\exists r' \in \mathbf{Roles} : \quad r' \geq r \wedge (u, r') \in \mathbf{UR} \\ \text{mit } u = \text{user-of-session}(s))\}$$

Randbedingungen

In einem Unternehmen gibt es viele Möglichkeiten für einzelne Angestellte, Mißbrauch zu betreiben. Es muß deshalb Mechanismen geben, dies zu unterbinden. So kann etwa die Anzahl von Personen, die eine bestimmte Schlüsselposition (z. B. Manager) besetzen, begrenzt sein, oder der gleiche Angestellte darf nicht zwei oder mehr Positionen einnehmen, die es ihm ermöglichen würden, das Unternehmen zu betrügen.

Es lassen sich beliebig viele Randbedingungen finden, die an Benutzer, Rollen und Permissions gestellt werden können. Allerdings gibt es in der Literatur einen Konsens darüber,

welche RBAC unterstützen sollte [SCFY96, FGK95]. Es sollte bedacht werden, daß derartige Randbedingungen effizient überprüfbar sein müssen, da vor allem in einem verteilten System ansonsten Performanceeinbußen die Folge sein können.

Im folgenden werden die drei in [SCFY96] vorgestellten Randbedingungsklassen betrachtet und ihre Auswirkungen auf RBAC untersucht. Grundsätzlich können alle Randbedingungen sowohl auf Rollen als auch analog auf Permissions angewendet werden.

Sich gegenseitig ausschließende Rollen/Permissions: Wie in der Einleitung zu diesem Abschnitt erwähnt, wird es in einem Unternehmen oft der Fall sein, daß ein und derselbe Angestellte nicht gleichzeitig zwei Rollen zugewiesen sein darf. Denkbar wäre die Situation, in der ein fiktiver Benutzer A sowohl Einkäufer als auch Buchhalter eines Unternehmens ist. In einem sorgfältig geführten Unternehmen wird also dafür Sorge getragen werden, daß dieser Fall nicht eintreten kann. Die Rollen *Einkäufer* und *Buchhalter* schließen sich also gegenseitig aus. Kein Benutzer darf gleichzeitig Mitglied in beiden Rollen sein. Gleiches gilt für Permissions, d. h. eine Permission darf nicht zwei Rollen zugeordnet sein, wenn sich diese gegenseitig ausschließen.

Man kann weiterhin unterscheiden, ob der gegenseitige Ausschluß statisch oder dynamisch erfolgt. Statisch bedeutet dabei, daß ein Benutzer nicht einer Rolle zugeordnet sein darf, die sich mit einer ihm schon zugeordneten gegenseitig ausschließt. Dynamisch bedeutet, daß ein Benutzer eine Rolle nicht zu seiner Menge von aktiven Rollen in einer Sitzung hinzufügen darf, wenn sie durch eine bereits aktive Rolle ausgeschlossen wird.

Kardinalitätsbeschränkungen: Es gibt in jedem Unternehmen Positionen, die nicht beliebig oft besetzt werden können. Der Filialleiter einer Bank ist ein Beispiel dafür. In RBAC ist es möglich, dies zu modellieren. Dazu dienen Kardinalitätsbeschränkungen, die an Rollen (und analog auch an Permissions) gestellt werden können. Man unterscheidet zwischen *maximalen* und *minimalen* Kardinalitätsbeschränkungen. Maximal bedeutet, daß eine Rolle höchstens eine festgelegte Zahl von Mitgliedern haben darf. Minimal dementsprechend, daß eine Mindestzahl von Mitgliedern eingehalten werden muß. Vor allem die zweite Bedingung verursacht im allgemeinen Schwierigkeiten [SCFY96], etwa wenn ein Benutzer gelöscht werden soll und damit eine solche Minimalitätsbedingung verletzt werden würde.

Wie auch im vorhergehenden Paragraphen muß man zwischen der statischen und dynamischen Anwendung von Kardinalitätsbeschränkungen unterscheiden.

Vorausgesetzte Rollen/Permissions: Manchmal kann es sinnvoll sein, daß ein Benutzer, um einer bestimmten Rolle zugeordnet zu werden, bereits Mitglied einer anderen Rolle sein muß, d. h. die Zugehörigkeit zu dieser anderen Rolle wird vorausgesetzt.

Analog zu den beiden vorhergegenden Paragraphen gibt es auch hierzu die analoge Randbedingung bei Permissions. Auch die statische und dynamische Anwendung muß unterschieden werden.

5.3.3 Administrierung von RBAC

Die Verwaltung eines Sicherheitssystems, egal ob es sich um ein zentrales oder verteiltes handelt, stellt immer einen wichtigen Punkt dar.

In [SCFY96] wird eine Möglichkeit der Administrierung von RBAC vorgestellt. Dabei wird RBAC verwendet, um sich selbst zu verwalten (siehe dazu auch Abbildung 5.5).

In AutO werden dazu drei neue Mengen eingeführt, **AdminUsers**, **AdminRoles** und **AdminPermissions**. Durch die letzten beiden werden die administrativen Rollen und Permissions festgelegt. Die Menge **AdminUsers** enthält die Administratoren des RBAC-Systems.

Die Verwaltung dieser zweiten, administrativen Ebene könnte analog mit Hilfe von RBAC erfolgen, indem eine dritte Ebene über der zweiten errichtet wird. Allerdings müßte dann auch diese dritte Ebene verwaltet werden, usw. Da man nicht unendlich viele Ebenen zur Verwaltung der jeweils darunterliegenden RBAC-Schicht errichten kann, muß eine letzte Ebene definiert werden, die dann auf andere Art und Weise verwaltet wird.

Die Verwaltung dieser letzten Ebene könnte etwa durch eine einzige Person, eine Art *root* Administrator, analog dem Benutzer *root* in einem Unix-System, erfolgen. Dieser besäße das Recht, alle administrativen Mengen der direkt darunterliegenden Schicht zu modifizieren.

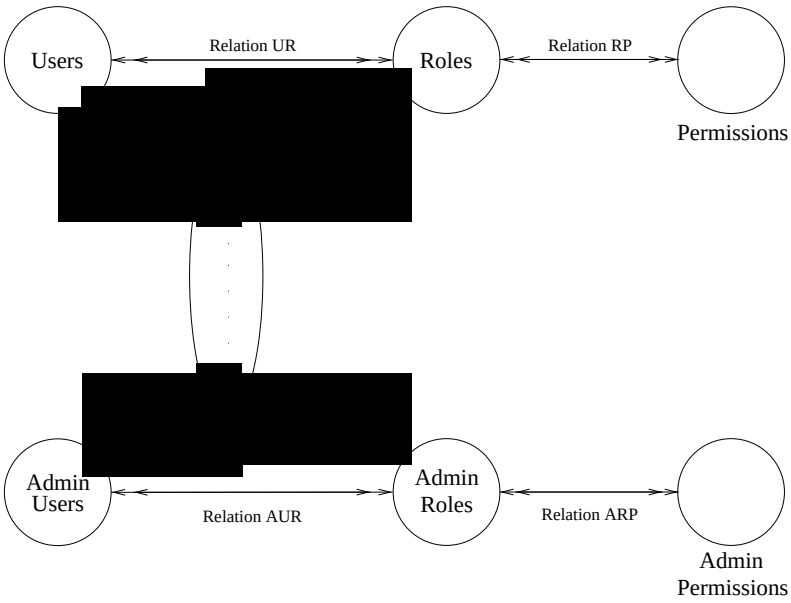


Abbildung 5.5: Ein Administrierungsmodell für RBAC

5.4 Die Implementierung von RBAC in AutO

Das in Abschnitt 5.3 vorgestellte Modell eines RBAC-Autorisierungsmodells wurde im Rahmen dieser Diplomarbeit in AutO implementiert. Der nun folgende Abschnitt erklärt, welche Komponenten implementiert wurden, wie diese in das bestehende AutO-System integriert wurden und wie sie zu benutzen sind.

5.4.1 Grundlagen der Autorisierung in AutO

Das Autorisierungssystem von AutO muß auf die verteilte Struktur des AutO-Systems abgestimmt sein. Der folgende Abschnitt beleuchtet diesen Punkt sowie andere grundsätzliche Eigenschaften von AutO, die bei der Implementierung zu berücksichtigen waren, und beschreibt, welche Lösungsmöglichkeiten entworfen und letztendlich implementiert wurden.

Authentifizierung in AutO

AutO baut bei der Erkennung eines Benutzers (Authentifizierung) notwendigerweise auf die verwendete *Java Virtual Machine* (JVM) auf, die ihre Informationen vom zugrundeliegenden Betriebssystem bezieht. Demzufolge wird ein Benutzer nur durch seine Kennung identifiziert oder aber eine entsprechende Bezeichung, die vom Betriebssystem dafür geliefert wird. Da für die Authentifizierung die JVM benutzt wird, kann jeder AutO-Prozeß genau einem Benutzer zugeordnet werden, nämlich demjenigen, der ihn gestartet hat.

Eine Ausnahme bildet ein AutO-Prozeß, in dem ein Terminal-Server ausgeführt wird. Benutzer bauen eine Verbindung zu einem solchen Prozeß mit Hilfe einer (gesicherten) Socket-Verbindung auf. Bei normalen Java-Sockets könnte nun ein Benutzer, der eine solcher Verbindung herstellt, eine beliebige Identität vortäuschen. Der Terminal-Server besäße keine Möglichkeit, den Wahrheitsgehalt der Angaben des Benutzers zu überprüfen. In diesem Fall wäre die einzige Lösung die Angabe von Kennwort und Paßwort des Benutzers. Durch die Verwendung von AutO-Sockets ist es dem Terminal-Server jedoch möglich, die Identität des Benutzers zu verifizieren, der sich mit dem Server verbunden hat, indem er das Zertifikat überprüft, das beim Verbindungsaufbau vom Client, d. h. dem Benutzer bzw. dem Terminal, das er benutzt, übermittelt werden muß (siehe Abschnitt 4.6). Dieses enthält den Namen des Benutzers, für den es ausgestellt wurde. Damit kennt der Terminal-Server die Identität des Benutzers, der eine Verbindung aufgebaut hat.

Rollen von Transaktionen und autonomen Objekten

AutO erlaubt einem Benutzer nur, Transaktionen zu starten. Erst diese können autonome Objekte kreieren und Nachrichten an Objekte senden. Demzufolge stellt sich die Frage, welche Rollen und damit Rechte Transaktionen und autonome Objekte (denn auch sie können Nachrichten senden) haben.

Transaktionen werden von einem Benutzer in einer Sitzung gestartet. Sie agieren also im Auftrag eines Benutzers im System und erben damit bei ihrer Kreierung die aktiven Rollen der Sitzung. Sendet eine Transaktion eine Nachricht, so werden in dieser Nachricht diese Rollen mitgeschickt.

Autonome Objekte erben bei ihrer Instanziierung die Rollen der Transaktion, die sie erzeugt hat. Sendet ein autonomes Objekt eine Nachricht, so wird zwischen verschiedenen Fällen unterschieden:

- Die Nachricht wird in einem On-Guard abgeschickt, der von einer Transaktion ausgelöst wurde. Die ausgehende Nachricht kann nun (prinzipiell) mit drei verschiedenen Rollenmengen gesendet werden:
  - Die Rollen der zu sendenden Nachricht sind gleich den Rollen der Nachricht, die den On-Guard ausgelöst hat.
  - Die ausgehende Nachricht erhält als Rollenmenge die Rollen, die dem autonomen Objekt bei dessen Kreierung mitgegeben wurden.
  - Die Menge der Rollen der zu sendenden Nachricht besteht aus der Vereinigung der beiden bisher genannten Mengen.

  Will man die dritte Möglichkeit nutzen, also die Vereinigung der beiden einzelnen Rollenmengen, so ergibt sich ein grundsätzliches Problem: Soll mit dieser vereinigten Rollenmenge später überprüft werden, ob sie die Ausübung eines Rechtes erlaubt, besteht die Möglichkeit, daß in einer der Mengen die entsprechende Permission enthalten ist, die die Ausübung autorisiert, und in der anderen die genau gegenteilige Permission, die die Ausübung verbietet. Für einen solchen Fall ist keine allgemein gültige Lösung erkennbar. Deshalb wurde diese dritte Möglichkeit in AutO nicht verwendet. Übrig bleiben die beiden ersten Möglichkeiten, die nochmals in Abbildung 5.6 verdeutlicht werden.

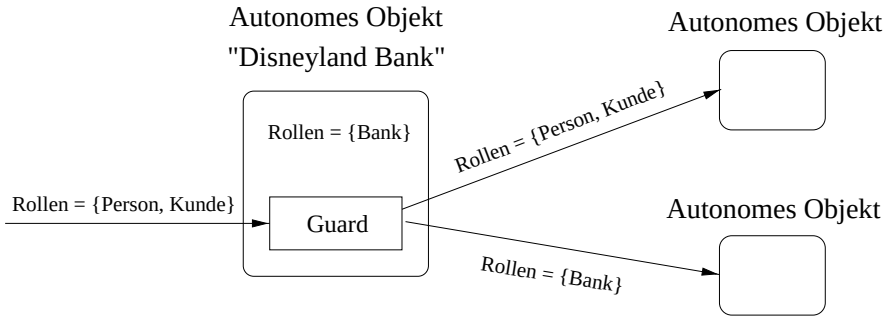


Abbildung 5.6: Mögliche Rollen einer ausgehenden Nachricht eines autonomen Objektes

- Die Nachricht wird in einem If-Guard abgeschickt. Handelt es sich dabei um einen gekoppelten Guard [Gri97, Isl97], sind wiederum die zwei Möglichkeiten denkbar, die auch im vorhergehenden Punkt angesprochen wurden. Beide werden in AutO unterstützt.

Handelt es sich allerdings um keinen gekoppelten If-Guard, so muß als Rollenmenge zwangsweise diejenige des autonomen Objekts verwendet werden.

- Ein At-Guard wird durch ein Zeitereignis ausgelöst, das von einer bestimmten Transaktion beim Timer installiert wurde. Demzufolge stehen in AutO wiederum die zwei im ersten Fall angesprochenen Möglichkeiten für die Rollenmenge einer zu sendenden Nachricht zur Verfügung.

Permissions eines Benutzers in AutO

Die Möglichkeiten, wie ein Benutzer Daten lesen, schreiben und verändern kann, sind in AutO begrenzt. Einem Benutzer stehen nur sehr wenige Aktionen zur Verfügung:

- **Ausführen einer Transaktion**: Nur Transaktionen sind in AutO in der Lage, die beiden folgenden Aktionen durchzuführen. Ein Benutzer, dem es nicht erlaubt ist, eine Transaktion auszuführen, kann den Gesamtzustand des AutO-Systems auf legale Weise nicht verändern.

  Parametrisiert werden kann diese Permission durch die Angabe der Namen der Transaktionen, die ausgeführt werden dürfen.

- **Kreieren eines autonomen Objekts:** Daten werden in AutO ausschließlich in autonomen Objekten gespeichert. Das Kreieren eines solchen autonomen Objekts stellt deshalb eine sehr wesentliche Permission dar. Sie kann ebenso wie das Ausführen einer Transaktion parametrisiert werden, indem man die Objektklassen angibt, von denen Instanzen erzeugen werden dürfen.

- **Aufrufen eines Guards:** Durch das Aufrufen von Guards autonomer Objekte ist es möglich, Daten im AutO-System zu lesen, zu ändern, etc.

  Ähnlich wie bei der Bedingung eines Guards ist es bei diesem Recht denkbar, es vom Zustand des Objekts sowie den Parametern der Nachricht, die den Guard ausgelöst hat, abhängig zu machen.

Die Granularität von Permissions

Die Permissions **Ausführen einer Transaktion** bzw. **Kreieren eines Objekts** beziehen sich immer auf eine Klasse. Diese Klasse unterliegt gewissen Einschränkungen, z. B. daß sie von `Transaction` bzw. `UserObject` abgeleitet sein muß. Die Granularität dieser Rechte ist aber damit festgelegt.

Anders bei der Permission **Aufrufen eines Guards**. Auch hier kann von einer Granularität auf Klassenebene ausgegangen werden. Doch ist eine Ausdehnung auf einzelne Instanzen einer Objektklasse vorstellbar:

- Die Granularität wird auf die Klassenebene festgelegt. Demzufolge legt man damit fest, ob das Aufrufen eines Guards für *alle* Instanzen einer Klasse erlaubt ist oder nicht.

Grundsätzlich scheint diese Lösung sehr sinnvoll zu sein, jedoch ist ihre Flexibilität nicht ausreichend. Es kann durchaus Sinn machen, nur den Zugriff auf eine Instanz einer Klasse zu erlauben. Das wäre mit dieser Lösung nicht möglich.

- Die Granularität wird auf die Instanzebene festgelegt. Für jede Instanz einer Klasse, auf die eine Rolle Zugriff haben soll, muß eine passende Permission vorhanden sein. Da man davon ausgehen muß, daß in einem AutO-System sehr viele Objekte existieren und deshalb eine Rolle meist Zugriff auf sehr viele Instanzen einer Klasse haben wird, erscheint diese Lösung aus administrativer Sicht als nicht sinnvoll.

  Ein weiteres Problem besteht darin, daß man einem Security-Representative explizit mitteilen müßte, wenn man ein Objekt kreiert hat und nun Zugriff darauf haben möchte. Denn nur er kann die Rollen und Permissions entsprechend konfigurieren.

- Eine kombinierte Lösung aus beiden Ansätzen bildet die beste Lösung. Grundsätzlich werden Permissions auf Klassenebene deklariert, allerdings besteht auch die Möglichkeit sie auf Instanzen einer Klasse neu zu definieren (sozusagen als Ausnahme). Dies wird dann in einer entsprechenden Permission vermerkt. Außerdem ist es damit möglich, für eine Rolle das Aufrufen von Guards nur für bestimmte Instanzen einer Klasse zu erlauben.

Randbedingungen

Bei Randbedingungen gilt es im Prinzip zwei Fälle zu unterscheiden: statische und dynamische Randbedingungen.

- **Statische Randbedingungen** können überprüft werden, wenn ein Security-Representative Benutzer hinzufügt oder löscht, Rollen modifiziert, etc.

- **Dynamische Randbedingungen** können erst zur Laufzeit überprüft werden, wenn beispielsweise eine Rolle zu den aktiven Rollen einer Sitzung hinzugefügt werden soll.

Sitzungen

Eine Sitzung wird in AutO durch einen AutO-Prozeß repräsentiert. Ein solcher kann eindeutig einem Benutzer zugeordnet werden. Demzufolge muß pro Prozeß eine Menge aktiver Rollen verwaltet werden. Dem Benutzer muß weiterhin die Möglichkeit gegeben werden, Rollen zu aktivieren und zu deaktivieren. Der Vorgang der Aktivierung einer Rolle für einen bestimmten Benutzer wird auch als *Rollenautorisierung* bezeichnet. Dabei muß sichergestellt werden, daß der Benutzer berechtigt ist, diese Rolle zu aktivieren.

Weiterhin muß bei Prozeßstart verifiziert werden, ob der Benutzer, der den AutO-Prozeß gestartet hat, ein gültiger Benutzer des AutO-Systems ist. Dies kann dadurch geschehen, daß überprüft wird, ob der Benutzer Element der **User**-Menge ist.

Administrierung von RBAC

Wie schon in Abschnitt 5.3.3 dargestellt geschieht die Verwaltung der Mengen **Users**, **Roles**, **Permissions** und der beiden Relationen **UR** und **RP** mit Hilfe einer zweiten Schicht von RBAC. Dazu wird ein neuer Satz von Mengen, im folgenden als *administrative RBAC-Mengen* bezeichnet, eingeführt. Dieser Satz besteht aus den einzelnen Mengen **AdminUsers**, **AdminRoles** und **AdminPermissions**.

Die Menge **AdminUsers** enthält nur Benutzer, die das Recht haben, die normalen RBAC-Mengen zu verwalten. Ein derartiger Benutzer wird im folgenden auch als *SDA-Administrator* bezeichnet. Die Menge **AdminRoles** enthält Rollen, denen nicht normale Permissions zugrunde liegen, wie sie in Abschnitt 5.4.3 erläutert werden, sondern besondere *administrative Permissions*, abgeleitet von der Klasse `AdminPermission`, die selbst wiederum von `RBACPermission` abgeleitet ist.

Natürlich können alle in Abschnitt 5.3.2 vorgestellten Randbedingungen auch auf administrative Rollen und Permissions angewandt werden. Ansonsten werden die administrativen Mengen genauso behandelt wie ihre normalen Pendants.

Administrative Permissions: Die Menge der administrativen Permissions stellt die Rechte zur Verfügung, die benötigt werden, um RBAC zu verwalten. Ein *SDA Administrator* muß jede Menge, die für RBAC benötigt wird, modifizieren können. Im folgenden werden nun die administrativen Permissions aufgeführt, die AutO bietet, um dies zu erreichen:

- Anlegen/Löschen eines Benutzers,
- Zuordnen einer Rolle zu einem Benutzer,
- Löschen der Zuordnung einer Rolle zu einem Benutzer,
- Anlegen/Löschen einer Rolle,
- Zuordnen einer Permission zu einer Rolle,
- Löschen der Zuordnung einer Permission zu einer Rolle,
- Hinzufügen/Löschen einer Permission.

Die Verwaltung der administrativen Ebene: Die administrative Ebene, d. h. die administrativen Mengen **AdminUsers**, **AdminRoles** und **AdminPermissions**, muß ebenfalls verwaltet werden. Dafür ist eine einzelne Person verantwortlich, der sogenannte *SDA-Root-Administrator*. Genauso wie normale SDA-Administrators die normalen RBAC-Mengen verwalten, verwaltet er die administrativen RBAC-Mengen. Nur ist er, im Gegensatz zu den normalen Administratoren, deren Rechte durch die administrativen RBAC-Mengen festgelegt werden, keinem RBAC-System unterworfen.

5.4.2 RBAC in einem verteilten System

Ein verteiltes System zeichnet sich unter anderem dadurch aus, daß verschiedene Systemkomponenten auf gegebenenfalls geographisch weit voneinander entfernt liegenden Rechnern ausgeführt werden, wobei dies für den Benutzer im Idealfall transparent geschieht, d. h. er erkennt keinen Unterschied zu einem nicht-verteilten System.

Die in 5.3 eingeführten Mengen müssen – zumindest teilweise – auf jedem AutO-Node vorhanden sein. Nur so ist es möglich, die Autorisierung eines Benutzers für eine bestimmte Aktion effizient zu überprüfen. Allerdings ist es unnötig und ineffizient, alle Daten auf allen Rechnern abzulegen. Diese Redundanz würde etwa Änderungen an den Mengen enorm aufwendig machen. Caching bietet eine flexible und leistungsfähige Möglichkeit, nur diejenigen Teile der verschiedenen Mengen auf einem Rechner zu speichern, die regelmäßig benötigt werden. Bei RBAC reicht es aus, nur die beiden Relationen **UR** und **RP** zu puffern, da sie alle notwendigen Informationen enthalten.

Im folgenden wird der Cache, der Teile der **UR**-Relation puffert, mit *Benutzer-Cache* bezeichnet, derjenige für die **RP**-Relation mit *Rollen-Cache*. Der Rest dieses Abschnitts beschreibt die Strategien, die angewendet werden, um die Kohärenz der beiden Puffer zu gewährleisten. Für jeden Cache wird eine eigene Strategie verwendet.

Die Teile der **UR**-Relation, die Informationen über einen bestimmten Benutzer und die ihm zugeordneten Rollen enthalten, werden nur an den Rechnern benötigt, an denen sich dieser Benutzer im System anmeldet, d. h. eine Sitzung startet. Da die Zahl dieser Rechner gering ist, kann man ein Callback-Locking-ähnliches Verfahren [Fra96] zur Erhaltung der Kohärenz des Benutzer-Caches wählen, das die Gültigkeit der Puffereinträge zu jedem Zeitpunkt garantiert. Der zweite Cache wird *lazy* verwaltet, d. h. veraltete Daten werden nur aktualisiert, wenn es sich nicht mehr vermeiden läßt. Die Strategie, die dazu verwendet wird, baut auf der Kohärenz des Benutzer-Caches auf.

Bei der Autorisierung mit RBAC kann man zwei Arten unterscheiden: *Rollenautorisierung* und *Rechteautorisierung*. Erstere bezeichnet die Überprüfung der Erlaubnis eines Benutzers, eine bestimmte Rolle zu aktivieren. Die notwendigen Informationen dazu werden dem Benutzer-Cache entnommen. Rechteautorisierung andererseits gewährleistet, daß ein Benutzer aufgrund seiner Rollen befugt ist, ein bestimmtes Recht auszuüben. Dazu wird der Rollen-Cache benötigt.

Alle RBAC-relevanten Mengen sind in der *Security-Data-Administration* (SDA) abgelegt. Benötigt ein AutO-Node eine sicherheitsrelevante Information, die nicht im entsprechenden lokalen Cache vorhanden ist, muß sie bei der Security-Data-Administration angefordert werden.

Der Benutzer-Cache

Der Benutzer-Cache puffert Teile der **UR**-Relation. Ein einzelner Eintrag wird im folgenden als *Benutzer-Information* bezeichnet und enthält folgende Daten:

- die Identität des Benutzers, z. B. dessen Namen,

- die ihm zugeordneten Rollen und
- die Versionsnummern dieser Rollen.

Die Versionsnummern sind hauptsächlich für den Rollen-Cache von Belang, genauer wird dies weiter unten erklärt. Wichtig ist, daß für die Vergabe der Versionsnummern einzig die Security-Data-Administration zuständig ist. Sie paßt diese an, wenn Änderungen an einer Rolle erfolgen – ihr etwa neue Benutzer oder Permissions zugeordnet werden.

Die Strategie, die für die Kohärenz dieses Puffers sorgt, ist an das in Client/Server-Datenbanken bekannte Callback-Locking-Verfahren [Fra96] angelehnt. Der Grundgedanke ist, daß Daten im Cache immer gültig sein sollen. Sie spiegeln damit genau den Zustand wider, der auch in der Security-Data-Administration vorliegt. Ein AutO-Node kann damit bedenkenlos diese Informationen verwenden. Sind die gewünschten Daten nicht im Puffer vorhanden, wird eine entsprechende Anfrage an die Security-Data-Administration gesendet. Diese sendet die angeforderte Informationen zurück, merkt sich aber gleichzeitig in einem Verzeichnis den Rechner, welcher sie angefordert hat. Ändern sich die Daten in der Security-Data-Administration – nur hier können sie modifiziert werden –, werden alle AutO-Nodes benachrichtigt, die nun veraltete Versionen dieser Daten gepuffert haben. Sie entfernen daraufhin die ungültigen Kopien aus ihrem Puffer.

Ein AutO-Node greift in zwei unterschiedlichen Situationen auf den Benutzer-Cache zu. Zum einen, wenn ein Benutzer versucht, eine Sitzung zu starten, zum anderen, wenn er eine Rolle aktivieren will.

Der Start einer Sitzung erfordert es, daß die Zugehörigkeit des Benutzers zur Menge **Users** überprüft wird. Dazu wird versucht, die entsprechende Benutzer-Information aus dem Benutzer-Cache zu laden. Scheitert dies, so muß sie bei der Security-Data-Administration angefordert und anschließend in den lokalen Benutzer-Cache eingefügt werden. Ist die Benutzer-Information weder im Benutzer-Cache gepuffert noch bei der Security-Data-Administration bekannt, dann handelt es sich bei dem Benutzer um keine autorisierte Person. Ihm ist es damit nicht erlaubt, eine Sitzung zu starten.

Die Rollenautorisierung stellt die zweite Situation dar, in der auf den Benutzer-Cache zugegriffen wird. Dabei muß überprüft werden, ob die Rolle dem Benutzer, der sie zu aktivieren beabsichtigt, zugeordnet ist. Diese Information ist in der entsprechenden Benutzer-Information enthalten, die beim Start der Sitzung kopiert wurde.

Der Rollen-Cache

Die Teile der **RP**-Relation, die für die Rechteautorisierung notwendig sind, werden im Rollen-Cache gepuffert. Ein Eintrag dieses Puffers wird *Rollen-Information* genannt. Er enthält im wesentlichen folgende Informationen:

- den Namen der Rolle,
- die Versionsnummer der Rolle und
- die der Rolle zugeordneten Permissions.

Das Update dieses Caches erfolgt *lazy*, d. h. erst dann, wenn es unbedingt notwendig ist. Zugriffe auf diesen Puffer erfolgen nur – wie weiter oben erwähnt – bei der Rechteautorisierung. Dabei wird überprüft, ob die aktivierten Rollen eines Benutzers es ihm erlauben, eine gewünschte Aktion auszuführen. Für jede aktivierte Rolle muß also die dazu gehörende Rollen-Information aus dem Puffer geholt werden. Ist sie dort nicht vorhanden, wird sie bei der Security-Data-Administration angefordert. Nach dem Durchsuchen aller Rollen-Informationen kann dann entschieden werden, ob der Benutzer die gewünschte Aktion ausführen darf oder nicht.

Entscheidend für die Korrektheit dieser Vorgehensweise ist, daß die Rollen-Informationen gültig sind, d. h. denen der Security-Data-Administration entsprechen. Dies kann überprüft werden, indem man die Versionsnummer der aktivierten Rolle – sie ist in der Benutzer-Information des Benutzers gespeichert – mit der Versionsnummer der gepufferten Rollen-Information vergleicht. Sind sie gleich, besitzt der Rollen-Cache eine gültige Version der Rollen-Information. Ist die Versionsnummer der Rollen-Information kleiner, sind die gepufferten Daten veraltet. Aktuelle Daten müssen in diesem Fall bei der Security-Data-Administration angefordert und der Cache damit aktualisiert werden (Abbildung 5.7). Ist die Versionsnummer der Rollen-Information größer, ist die Benutzer-Information, aus der die aktivierte Rolle stammt, veraltet. Die Aktion muß abgebrochen werden. Dieser letzte Fall kann auftreten, wenn nach dem Start einer Sitzung – dabei wird die Benutzer-Information aus dem Benutzer-Cache kopiert – die RBAC-Mengen modifiziert werden und sich damit eine veränderte Benutzer-Information ergibt. Obwohl der Benutzer-Cache nach der Änderung die aktuelle Benutzer-Information enthält, gilt dies nicht für die Sitzung. Sie besitzt damit eine veraltete Version.

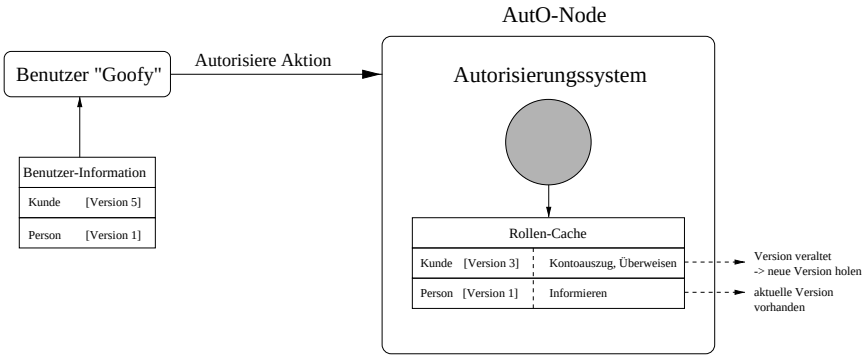


Abbildung 5.7: Caching von Rollen auf einem AutO-Node

Zum Schluß soll noch angemerkt werden, daß alle Daten durch die Übertragung über sichere AutO-Verbindungen vor unbemerkten Modifikationen geschützt sind. Die Daten in den beiden Puffern, ebenso wie die Rollenmengen, die mit Nachrichten übertragen werden, sind damit immer korrekt.

Randbedingungen

Auch Randbedingungen bedürfen im verteilten Falle einer gesonderten Betrachtung. Alle statischen Constraints können einfach geprüft werden. Änderungen müssen *immer* in der SDA durchgeführt werden und dabei ist es möglich, bei einer Modifikation alle statischen Constraints zu überprüfen.

Schwieriger wird es bei dynamischen Constraints, die erst zur Laufzeit verifiziert werden können.

Sich gegenseitig ausschließende und vorausgesetzte Rollen/Permissions: Diese beiden Bedingungen sind leicht einzuhalten, indem sie bei der Rollen- bzw. Rechteautorisierung geprüft werden.

Wird etwa eine Rolle zu der Menge der aktiven Rollen einer Sitzung hinzugefügt, kann man feststellen, ob sie durch eine bereits aktivierte ausgeschlossen wird. Ebenso leicht kann erkannt werden, ob eine vorausgesetzte Rolle bereits aktiviert wurde. Analoges gilt für die Rechteautorisierung.

Kardinalitätsbeschränkungen: Dynamische Kardinalitätsbeschränkungen in einem verteilten System sind nur mit großem Aufwand zu realisieren. Gilt etwa für eine Rolle *Manager* die Beschränkung, daß sie zu jedem Zeitpunkt immer nur genau ein Benutzer im AutO-System aktiviert haben darf, müßte das Sicherheitssystem, immer wenn ein Benutzer versucht, diese Rolle zu aktivieren, nachprüfen, ob diese Einschränkung erfüllt ist. Dazu sind verschiedene Lösungsansätze denkbar:

- Einerseits könnte eine Instanz existieren, die für alle Rollen verwaltet, wer sie wo gerade aktiviert hat. Jede Rolle, die aktiviert wird und für die eine Kardinalitätsbeschränkungen existiert, müßte bei dieser Instanz gemeldet werden. Das Laufzeitverhalten von AutO würde dabei in höchstem Maße beeinflußt, da gerade das Aktivieren und Deaktivieren von Rollen häufig geschieht.
- Andererseits könnte das Sicherheitssystem einfach alle bekannten Rechner befragen, ob auf ihnen ein Benutzer die Rolle *Manager* aktiviert hat. Gerade bei AutO, das eine große Zahl von verteilten Rechner in ein System integriert, ist diese Lösung nicht tragbar.

Festzuhalten bleibt also, daß aus Performancesicht eine Implementierung und Verwendung von dynamischen Kardinalitätsbeschränkungen nicht sinnvoll ist.

5.4.3 Die RBAC-Komponenten eines AutO-Nodes

Abbildung 5.8 zeigt den grundsätzlichen Aufbau der implementierten Komponenten für das RBAC-Autorisierungssystem. Sie, und die ihnen zugrunde liegenden Klassen, werden in den nun folgenden Abschnitten genauer erläutert.

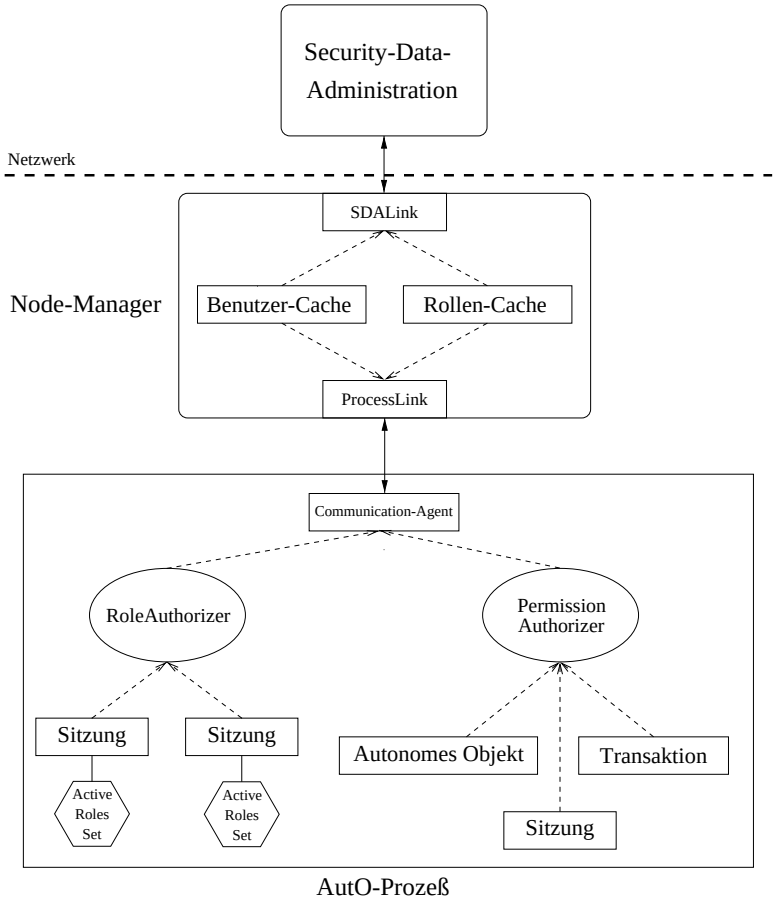


Abbildung 5.8: Aufbau des RBAC-Systems in einem AutO-Node

RBAC-Komponenten in einem AutO-Prozeß

In einem AutO-Prozeß können autonome Objekte erzeugt und gelöscht sowie Transaktionen gestartet werden. Damit ein Benutzer diese Funktionalität nutzen kann, muß er eine Sitzung starten.

Die Kommunikation eines AutO-Prozesses mit dem Rest des AutO-Systems, d. h. dem lokalen Node-Manager und anderen lokalen AutO-Prozessen sowie anderen AutO-Nodes, wird über den Communication-Agent abgewickelt (siehe dazu auch Abschnitt 2.2). Ihm werden die Nachrichten übergeben, die an ihre Empfänger zugestellt werden sollen. Auch die RBAC-Komponenten, die auf nicht im Prozeß vorhandene Informationen zugreifen wollen, nutzen diese Möglichkeit der Kommunikation.

Die Sitzung: Eine Sitzung, implementiert in der Klasse `RBACSession`, repräsentiert einen im System angemeldeten Benutzer. In AutO kann dies auf zwei verschiedene Arten geschehen:

- Der Benutzer startet einen normalen Java-Prozeß und erzeugt ein Instanz der Klasse

`RBACSession`. Beim Erzeugen bestimmt dieses Objekt die Identität des Benutzers – dies geschieht mit Hilfe der Java-System-Property *user.name* – und startet alle Komponenten, die für einen AutO-Prozeß notwendig sind.

Der Benutzer kann nur über das erzeugte `RBACSession`-Objekt bzw. dessen Methoden auf das AutO-System zugreifen.

- Der Benutzer startet ein Terminal und verbindet dieses mit einem Terminal-Server. Der Server erzeugt für den Benutzer, dessen Identität er mit Hilfe des für die Verbindung genutzten AutO-Sockets feststellen kann, ein `RBACSession`-Objekt.

Bei ihrer Instanziierung überprüft eine Sitzung, ob der Benutzer, der sie kreieren will, autorisiert ist, das AutO-System zu benutzen. Dazu befragt sie den *RoleAuthorizer* des AutO-Prozesses. Sollte dieser ablehnend antworten, kann kein `RBACSession`-Objekt erzeugt werden. Statt dessen wird eine Exception geworfen, die dem Benutzer mitteilt, daß er nicht autorisiert ist, AutO zu benutzen.

Eine Sitzung stellt einem Benutzer alle Methoden zur Verfügung, die er benötigt, um mit dem AutO-System kommunizieren zu können:

- *Aktivieren einer Rolle*: Durch das Aktivieren einer Rolle nimmt ein Benutzer eine Rolle, der er zugeordnet ist, in die Menge der aktiven Rollen, repräsentiert durch die Klasse `ActiveRolesSet`, der Sitzung auf. Die in dieser Menge enthaltenen Rollen bestimmen, zu welchen Aktionen der Benutzer autorisiert ist.

  Bevor die zu aktivierende Rolle der Menge der aktiven Rollen hinzugefügt wird, wird der RoleAuthorizer befragt, ob der Benutzer, dem die Sitzung zugeordnet ist, autorisiert ist, die Rolle zu aktivieren. Sollte dies nicht der Fall sein, wird eine Exception geworfen, die dem Benutzer mitteilt, daß die Rolle, die er zu aktivieren wünscht, unbekannt ist. Ihm wird nicht gesagt, daß er nicht autorisiert ist, sie zu aktivieren, dadurch ist es dem Benutzer unmöglich, durch gezielte Versuche festzustellen, welche Rollen dem System bekannt sind und welche nicht.

- *Deaktivieren einer Rolle*: Ein Benutzer entfernt damit eine Rolle aus der Menge der aktiven Rollen.

- *Starten einer Transaktion*: Ein Benutzer kann im Rahmen einer Sitzung eine Transaktion starten, die anschließend in seinem Namen verschiedene Aktionen ausführen kann.

  Bevor eine Transaktion mit Hilfe des Object-Managers (siehe Abschnitt 2.2) erzeugt und gestartet wird, wird zuerst der *PermissionAuthorizer* befragt, ob der Benutzer autorisiert ist, diese Aktion auszuführen.

Der RoleAuthorizer: Der RoleAuthorizer ist ein Singleton [GHJV95], d. h. in jedem AutO-Prozeß existiert nur eine Instanz dieser Klasse. Alle Sitzungen eines AutO-Prozesses greifen also auf dieselbe Instanz dieser Klasse zu.

Anmeldung eines Benutzers: Bei der Instanziierung wendet sich eine Sitzung an den RoleAuthorizer, damit dieser die Benutzer-Information des Benutzers anfordert. Eine Benutzer-Information (siehe auch Abschnitt 5.4.2), implementiert in der Klasse `UserInfo`, enthält den Benutzer sowie alle Rollen (mit Versionsnummer), denen dieser Benutzer zugeordnet ist.

Um die Benutzer-Information zu erhalten sendet der RoleAuthorizer eine `RequestUserInfoMessage`-Nachricht, die den Namen des Benutzers enthält, über den Communication-Agent des AutO-Prozesses an den lokalen Benutzer-Cache im Node-Manager. Anschließend wartet der RoleAuthorizer auf Antwort. Währenddessen kann er von anderen Sitzungen benutzt werden:

- Trifft die Antwort des Benutzer-Caches nicht innerhalb einer definierten Zeitspanne ein, wird ein Timeout ausgelöst und die Aktion abgebrochen. Die Überprüfung der Autorisierung scheitert damit und die Sitzung kann nicht erzeugt werden.

- Ist die Antwort des Benutzer-Caches vom Typ `ReturnUserInfoMessage`, dann war die Benutzer-Information des Benutzers entweder im Puffer vorhanden oder sie konnte von der SDA angefordert werden. In beiden Fällen ist der Benutzer Element der Menge **Users**. Die Sitzung kann damit gestartet werden.

- Sendet der Benutzer-Cache eine `UnknownUserMessage`-Nachricht als Antwort, ist der Security-Data-Administration der angegebene Benutzer unbekannt. Er ist nicht in der Menge **Users** enthalten und damit nicht autorisiert eine Sitzung zu erzeugen.

Wurde eine Benutzer-Information auf diese Weise beim Benutzer-Cache erfolgreich angefordert, bleibt sie unverändert, bis die Sitzung beendet wird, also auch wenn die entsprechenden Mengen des RBAC-Systems in der Security-Data-Administration geändert werden.

Autorisierung einer Rolle: Wenn ein Benutzer in einer Sitzung versucht, eine Rolle zu aktivieren, wird der RoleAuthorizer des AutO-Prozesses befragt, ob der Benutzer dazu autorisiert ist. Dies geschieht mit Hilfe der Benutzer-Information, die bei Erzeugung der Sitzung beim Benutzer-Cache angefordert wurde. Da die Benutzer-Information alle Rollen und deren Basisrollen enthält, denen der Benutzer zugeordnet ist, muß nur überprüft werden, ob die Rolle, die aktiviert werden soll, darin enthalten ist. Ist dies der Fall, so ist der Benutzer autorisiert, die Rolle zu aktivieren, anderenfalls wird eine `UnknownRoleException` geworfen.

Der PermissionAuthorizer: Die Singleton-Klasse `PermissionAuthorizer` wird in einem AutO-Prozeß dazu benutzt, die Autorisierung eines Benutzers zu überprüfen, eine bestimmte Aktion auszuführen. Immer wenn ein Benutzer oder ein von ihm beauftragtes Objekt, wie etwa eine Transaktion, eine Aktion ausführen möchte, müssen die dazu nötigen Permissions vorliegen.

Permissions in AutO: Permissions können in AutO sowohl positiv, d. h. sie erlauben dann explizit die damit verbunden Aktion, oder negativ, in diese Fall verbieten sie ausdrücklich die Aktion, sein. Folgende Permissions, jeweils abgeleitet von der Klasse `RBACPermission`, sind dem RBAC-System von AutO bekannt:

- `StartTransactionPermission`: Diese Permission erlaubt oder verbietet das Starten bzw. Kreieren einer Transaktion. Der Name der Transaktionsklasse ist in der Permission gespeichert.

- `CreateActiveObjectPermission`: Eine Permission dieser Art gibt an, ob es erlaubt oder verboten ist, eine Instanz des in der Permission genannten autonomen Objekts zu erzeugen.

- `AccessActiveObjectClassPermission`: Diese Permission gibt an, ob es erlaubt oder verboten ist, alle Guards von autonomen Objekten der in der Permission genannten Klasse aufzurufen.

- `AccessGuardClassPermission`: Diese Permission erlaubt oder verbietet es, den in der Permission angegebenen Guard bei autonomen Objekten einer bestimmten Klasse aufzurufen.

- `AccessActiveObjectInstancePermission`: Diese Permission erlaubt oder verbietet es, alle Guards eines autonomen Objekts aufzurufen, allerdings nur bei der Objektinstanz, deren OID in der Permission gespeichert ist.

- `AccessGuardInstancePermission`: Diese Permission erlaubt oder verbietet es, den in ihr angegebenen Guard desjenigen autonomen Objekts aufzurufen, dessen OID in der Permission gespeichert ist.

Autorisierung von Permissions: Immer wenn in AutO eine Aktion ausgeführt werden soll, für die eine der oben genannten Permissions notwendig ist, wird der PermissionAuthorizer befragt, ob die zur Verfügung stehenden Rollen die Aktion erlauben oder nicht.

Dazu wird für alle zur Verfügung stehenden Rollen die entsprechende Rollen-Information, implementiert in der Klasse `RoleInfo`, benötigt. Sie enthält neben der Rolle, der die Rollen-Information zugeordnet ist, die Versionsnummer der Rolle sowie alle Permissions, die der Rolle zugeordnet sind. Die Rollen-Information wird beim Rollen-Cache des lokalen Node-Managers angefordert, indem ihm eine `RequestRoleInfoMessage`-Nachricht gesendet wird, die alle Rollen (mit Versionsnummern) enthält, für die die Rollen-Information gewünscht wird. Anschließend wartet der PermissionAuthorizer auf eine Antwort des Rollen-Caches:

- Trifft die Antwort des Rollen-Caches nicht innerhalb einer vorgegebenen Zeitspanne ein, wird die Autorisierung abgebrochen. Die Aktion, die autorisiert werden sollte, kann nicht ausgeführt werden.

- Sendet der Rollen-Cache als Antwort eine `ReturnRoleInfoMessage`-Nachricht, dann stehen damit alle benötigten Rollen-Informationen zur Verfügung und die Autorisierung kann durchgeführt werden.

- Ist die Antwort vom Typ `UnknownRoleMessage`, ist mindestens eine der Rollen veraltet, da sie inzwischen in der SDA gelöscht wurde. Die Aktion wird abgebrochen.

- Erhält der PermissionAuthorizer als Antwort eine `ObsoleteRoleMessage`-Nachricht ist ebenfalls eine der Rolle veraltet, d. h. in der SDA wurden die entsprechenden Daten inzwischen modifiziert. Die Aktion wird in diesem Fall abgebrochen.

Konnten vom Rollen-Cache alle benötigten Rollen-Informationen erhalten werden, dann wird überprüft, ob die in den Rollen-Informationen enthaltenen Permissions es erlauben, die Aktion auszuführen:

- **Starten einer Transaktion**: Will ein Benutzer in einer Sitzung eine Transaktion starten, wird zuerst der PermissionAuthorizer befragt, ob die aktiven Rollen der Sitzung es dem Benutzer erlauben, die Transaktion zu starten. Wie vorher beschrieben werden aus der Menge der aktiven Rollen die dem Benutzer zur Verfügung stehenden Permissions bestimmt.

  Befindet sich darunter eine Permission, die das Starten der angegebenen Transaktion verbietet, wird die Aktion abgebrochen. Gleiches gilt, falls keine positive Permission gefunden wird, die es dem Benutzer erlaubt, die Transaktion zu erzeugen. Nur wenn eine entsprechende positive Permission gefunden wird, wird die Aktion erlaubt.

  Die Transaktion erbt in diesem Fall die gerade aktiven Rollen der Sitzung. Für alle von der Transaktion im folgenden durchgeführten Aktionen wird diese Menge an aktiven Rollen zur Autorisierung herangezogen. Darunter fallen etwa das Erzeugen eines autonomen Objekts oder das Aufrufen eines Guards eines autonomen Objekts.

- **Erzeugen eines autonomen Objekts**: Diese Aktion kann nur von einer Transaktion ausgeführt werden. Die Transaktion hat bei ihrer Erzeugung die Menge der aktiven Rollen des Benutzers geerbt, der sie kreiert hat. Mit dieser Menge wird nun die Autorisierung der Transaktion überprüft, das autonome Objekt zu erzeugen.

  Findet sich in der Menge der Permissions, die wie weiter oben beschrieben aus der geerbten Menge der aktiven Rollen bestimmt wird, eine entsprechende positive Permission, kann die Transaktion eine Instanz der angegebenen Klasse erzeugen. Ansonsten, d. h. es findet sich keine passende Permission bzw. eine negative Permission, die das Erzeugen von Instanzen dieser Klasse verbietet, wird die Aktion abgebrochen.

  Analog zur Erzeugung einer Transaktion erbt das instanziierte autonome Objekt die Rollenmenge der Transaktion.

- **Aufrufen eines Guards**: Sendet eine Transaktion oder ein autonomes Objekt eine Nachricht – gemeint ist hierbei eine `OnGuardMessage`-Nachricht – an ein autonomes Objekt, dann enthält diese Nachricht die Menge der aktiven Rollen des Absenders

(Transaktion oder autonomes Objekt). Ein autonomes Objekt, das eine solche Nachricht empfängt, bestimmt aus der Nachricht diese Menge der aktiven Rollen sowie den Guard, der ausgelöst werden soll. Mit diesen Informationen sowie dem Klassennamen und der OID des Empfängerobjekts überprüft der PermissionAuthorizer die Autorisierung des Senders.

Bei der Überprüfung der Autorisierung wird die Menge der Permissions, die aus der Menge der in der Nachricht enthaltenen aktiven Rollen bestimmt wurde, entsprechend den nachfolgend beschriebenen Schritten durchsucht: Zuerst wird nach einer passenden `AccessGuardInstancePermission` gesucht, die es erlaubt oder verbietet, den Guard des Empfängerobjekts aufzurufen. Läßt sich keine solche finden, wird nach einer geeigneten `AccessActiveObjectInstancePermission` gesucht, die es erlaubt oder verbietet, alle Guards des Empfängerobjekts aufzurufen. Existiert auch keine passende Permission dieses Typs, wird auf das Vorhandensein einer `AccessGuardClassPermission` getestet, die den Zugriff auf einen Guard aller autonomen Objekte einer bestimmten Klasse erlaubt oder verbietet. Verläuft auch diese Suche negativ, wird zuletzt nach einer `AccessActiveObjectClassPermission` gesucht, die den Zugriff auf alle Guards autonomer Objekte der genannten Klasse erlaubt oder verbietet.

Wurde bei dieser Suche keine passende Permission gefunden oder nur eine negative Permission, die es verbietet, den Guard des autonomen Objekts aufzurufen, wird die Nachricht vom Empfänger zurückgewiesen. Ansonsten wird sie in den Nachrichtenpuffer des Objekts eingereiht.

RBAC-Komponenten in einem Node-Manager

Die Aufgabe eines Node-Managers ist es, lokale Komponenten eines AutO-Systems mit anderen Komponenten zu verbinden. Dementsprechend übernimmt der Node-Manager auch die Kommunikation mit der Security-Data-Administration, die alle für die RBAC-Autorisierung notwendigen Daten bereithält. Er enthält die in Abschnitt 5.4.2 vorgestellten Puffer, also den Rollen-Cache und den Benutzer-Cache, die für eine effiziente Verwaltung der benötigten RBAC-Daten sorgen. Beide Puffer verwalten die gespeicherten Daten nach dem LRU-Prinzip in einem *LRUCache*, repräsentiert durch die Klasse `LRUCache`.

Zur Kommunikation mit der SDA wird ein *SDALink* benutzt, zur Kommunikation mit lokalen RBAC-Komponenten in einem AutO-Prozeß der entsprechende *ProcessLink*, der auch sonst für die Kommunikation zwischen Node-Manager und AutO-Prozeß verwendet wird. Im folgenden werden diese Komponenten nun ausführlicher beschrieben.

Der SDALink: Der SDALink eines Node-Managers, repräsentiert durch die Singleton-Klasse `SDALink`, wird zur Kommunikation mit der Security-Data-Administration verwendet. Er gewährleistet, daß zu jedem Zeitpunkt maximal eine Socket-Verbindung zur SDA existiert. Nachrichten, die ihm übergeben werden, leitet er an die Security-Data-Administration weiter. Nachrichten, die er über die Verbindung zur SDA erhält, übergibt er entweder dem Benutzer-Cache, falls es sich bei der Nachricht um eine `UserSDAMessage`

handelt, oder aber dem Rollen-Cache, falls die Nachricht vom Typ `RoleSDAMessage` ist.

Soll eine Nachricht an die SDA gesendet werden, wird zuerst überprüft, ob bereits eine Verbindung existiert. Wenn nicht, wird eine solche hergestellt, indem der *NodeLink*-Server der SDA kontaktiert wird. Im Anschluß kann die Nachricht an die SDA übermittelt werden. Die Verbindung zum NodeLink-Server bleibt dabei bestehen. Sie wird nur geschlossen, wenn der Verbindungs-Cache, den der NodeLink-Server verwaltet, überläuft und eine Verbindung daraus abgebrochen werden muß (siehe Abschnitt 5.4.4).

Der SDALink beinhaltet auch eine Server-Komponente, die benötigt wird, falls die SDA Kontakt zu einem Node-Manager aufnehmen möchte, etwa um ihm mitzuteilen, daß eine bestimmte Benutzer-Information ungültig geworden ist, aber keine aktive Verbindung zu diesem Node-Manager besteht. Die SDA öffnet in einem derartigen Fall eine Verbindung, indem sie den SDALink-Server kontaktiert.

Der Benutzer-Cache: Die Klasse `UserCache` implementiert den in 5.4.2 vorgestellten Benutzer-Cache. Sie ist als Singleton realisiert, da nur ein einziger Benutzer-Cache pro Node-Manager nötig und erlaubt ist.

Der Benutzer-Cache verwaltet alle gepufferten `UserInfo`-Objekte eines Auto-Nodes in einem LRUCache. Erreicht ihn eine `RequestUserInfoMessage`-Nachricht von einem RoleAuthorizer der an den Node-Manager angeschlossenen AutO-Prozesse, kann er auf zwei Arten reagieren:

- Sollte die angeforderte Benutzer-Information im Puffer enthalten sein, sendet der Benutzer-Cache dem RoleAuthorizer über den zugehörigen ProcessLink eine `ReturnUserInfoMessage`, die die gewünschte Benutzer-Information enthält.

- Ist die Benutzer-Information nicht im Puffer enthalten, muß sie bei der Security-Data-Administration angefordert werden. Der Benutzer-Cache sendet in diesem Fall mit Hilfe des lokalen SDALinks eine `RequestUserInfoSDAMessage`-Nachricht an die SDA. Die Anfrage des RoleAuthorizers wird gespeichert, so daß die Benutzer-Information an ihn weitergeleitet werden kann, wenn die Antwort der SDA eintrifft.

Erreicht eine Nachricht der SDA für den lokalen Benutzer-Cache den SDALink des Node-Managers – in diesem Falle ist sie eine Instanz der Klasse `UserSDAMessage` oder aber davon abgeleitet –, wird sie direkt an den Benutzer-Cache weitergeleitet. Abhängig von der Art der erhaltenen Nachricht wertet sie der Benutzer-Cache aus:

- `ReturnUserInfoSDAMessage`: Die SDA beantwortet damit Anfragen nach Benutzer-Informationen. Die Benutzer-Information wird zuerst in den LRUCache eingefügt, dann wird allen auf diese Benutzer-Information wartetenden RoleAuthorizern eine `ReturnUserInfoMessage` gesendet, die die gewünschte Benutzer-Information enthält.

  Muß beim Einfügen der neuen Benutzer-Information ein alter Eintrag aus dem Puffer entfernt werden, weil er überfüllt ist, wird die SDA darüber benachrichtigt. Dazu

sendet der Benutzer-Cache eine `RemovedUserInfoMessage`-Nachricht an die SDA. Dadurch muß dieser Rechner nicht mehr benachrichtigt werden, falls die nun entfernte Benutzer-Information ungültig werden sollte (siehe auch Abschnitt 5.4.4).

Der Benutzer-Cache wartet nicht auf eine Bestätigung dieser Nachricht seitens der SDA. Erhält die Security-Data-Administration die Nachricht nicht, geht sie fälschlicherweise weiterhin davon aus, daß der Benutzer-Cache die entfernte Benutzer-Information in seinem Puffer liegen hat. Tritt nun der Fall ein, daß die entsprechende Benutzer-Information ungültig wird, wird die SDA auch den Benutzer-Cache, dessen `RemovedUserInfoMessage`-Nachricht nicht angekommen ist, benachrichtigen. Dieser reagiert in einem solchen Falle so, als hätte er die Benutzer-Information noch gepuffert und würde sie erst jetzt entfernen.

- `UnknownUserSDAMessage`: Ein solche Nachricht besagt, daß der Benutzer, für den die Benutzer-Information angefordert wurde, unbekannt ist. Der Benutzer-Cache sendet daraufhin allen wartenden RoleAuthorizern eine `UnknownUserMessage`-Nachricht, um sie darüber zu informieren, daß dieser Benutzer nicht autorisiert ist, AutO zu benutzen.

- `RevokeUserInfoSDAMessage`: Sollten sich in der SDA Änderungen an den RBAC-Mengen ergeben, die Auswirkungen auf eine Benutzer-Information haben, etwa wenn dem entsprechenden Benutzer Rollen entzogen oder neue zugeordnet werden, sendet die SDA an alle Benutzer-Caches, die ein Kopie der nun ungültigen Benutzer-Information besitzen, eine `RevokeUserInfoSDAMessage`-Nachricht. Diese entfernen in einem solchen Fall die angegebene Benutzer-Information aus ihrem Puffer und bestätigen dies der SDA durch das Senden einer `RemovedUserInfoSDAMessage`-Nachricht.

Der Rollen-Cache: Analog dem Benutzer-Cache implementiert die Klasse `RoleCache` den in Abschnitt 5.4.2 vorgestellten Puffer für `RoleInfo`-Objekte. Alle `RoleInfo`-Objekte werden in einem LRUCache gespeichert. Die Klasse ist als Singleton implementiert, da nur ein Rollen-Cache pro Node-Manager-Prozeß zulässig ist. Der Rollen-Cache wird von dem PermissionAuthorizer eines lokalen AutO-Prozesses angesprochen, wenn Informationen über eine bestimmte Rolle benötigt werden. Erreicht einen Rollen-Cache die Anfrage eines PermissionAuthorizers in Form einer `RequestRoleInfoMessage`-Nachricht – womit der PermissionAuthorizer die `RoleInfo`-Objekte für mehrere in der Nachricht angegebene Rollen anfordert –, geht er für jede angeforderte Rollen-Information wie folgt vor:

- Ist die gewünschte Rollen-Information nicht im Puffer vorhanden, wird sie bei der SDA angefordert. Dazu sendet der Rollen-Cache über den lokalen SDALink eine `RequestRoleInfoSDAMessage`-Nachricht, die den Namen der Rolle enthält, deren Rollen-Information gewünscht wird.

- Enthält der LRUCache des Rollen-Caches die entsprechende Rollen-Information, so muß geprüft werden, ob sie aktuell ist. Dazu wird die Versionsnummer der in der gepufferten Rollen-Information enthaltenen Rolle mit der Versionsnummer der Rolle verglichen, die der PermissionAuthorizer mit seiner Nachricht übermittelt hat.

Stimmen die beiden Versionsnummern überein, ist die gepufferte Rollen-Information aktuell. Der Rollen-Cache merkt sich in diesem Fall, daß er eine gültige Version dieser Rollen-Information besitzt. Ist die Versionsnummer der gepufferten Rolle kleiner als die der vom PermissionAuthorizer übermittelten Rolle – die gepufferte Rolle ist also älter –, muß eine aktuelle Version von der SDA angefordert werden. Dazu sendet der Rollen-Cache der SDA eine `RequestRoleInfoSDAMessage`-Nachricht, die die Rolle enthält, deren Rollen-Information gewünscht wird. Ist die Versionsnummer der gepufferten Rolle allerdings größer als die der übermittelten, dann ist die Benutzer-Information der Sitzung, aus der die übermittelte Rolle entnommen wurde, veraltet und damit auch die Rolle. Der Rollen-Cache bricht in diesem Fall die Abarbeitung dieser Anforderung ab und benachrichtigt den entsprechenden PermissionAuthorizer mit Hilfe einer `ObsoleteRoleMessage`-Nachricht.

Hat der Rollen-Cache auf diese Weise die gesamte Anfrage bearbeitet, überprüft er, ob alle gewünschten `RoleInfo`-Objekte vorhanden sind, oder ob mindestens eines bei der SDA angefordert werden mußte. Sind alle vorhanden, stellt er sie mit Hilfe einer `ReturnRoleInfoMessage`-Nachricht an den PermissionAuthorizer über den entsprechenden ProcessLink zu. Mußten allerdings erst `RoleInfo`-Objekte bei der Security-Data-Administration angefordert werden, speichert er die Anfrage des PermissionAuthorizers, damit sie beantwortet werden kann, wenn alle `RoleInfo`-Objekte von der SDA eingetroffen sind.

Trifft im lokalen SDALink eine Nachricht vom Typ `RoleSDAMessage` oder eine davon abgeleitet Nachricht ein, wird sie direkt an den Rollen-Cache des Node-Managers weitergeleitet. Dieser unterscheidet anhand des Nachrichtentyps, welche Aktion er auszuführen hat:

- `ReturnRoleInfoSDAMessage`: Die SDA beantwortet mit dieser Nachricht die Anforderung einer Rollen-Information. Die in der Nachricht enthaltene Rollen-Information wird in den LRUCache eingefügt. Für alle Anfragen von lokalen PermissionAuthorizern, die diese Rollen-Information angefordert haben, wird nun überprüft, ob damit alle in der Anfrage geforderten `RoleInfo`-Objekte vorhanden sind. Ist dies der Fall, werden sie dem PermissionAuthorizer über den entsprechenden ProcessLink durch eine `ReturnRoleInfoMessage`-Nachricht zugestellt. Anderenfalls speichert der Rollen-Cache nur ab, daß eine weitere der gewünschten `RoleInfo`-Objekte vorhanden ist.

- `UnknownRoleSDAMessage`: Diese Nachricht wird von der SDA als Antwort gesendet, wenn die Rolle, für die eine Rollen-Information angefordert wurde, nicht bekannt ist. Dies kann allerdings nur geschehen, wenn eine Rolle gelöscht wurde. In einem solchen Fall werden alle auf diese Rollen-Information wartenden PermissionAuthorizer mit einer `UnknownRoleInfoMessage`-Nachricht darüber in Kenntnis gesetzt.

5.4.4 Die Security-Data-Administration (SDA)

Die Security-Data-Administration verwaltet alle Daten, die für die rollenbasierte Autorisierung benötigt werden. Sie beinhaltet neben den normalen RBAC-Mengen auch die

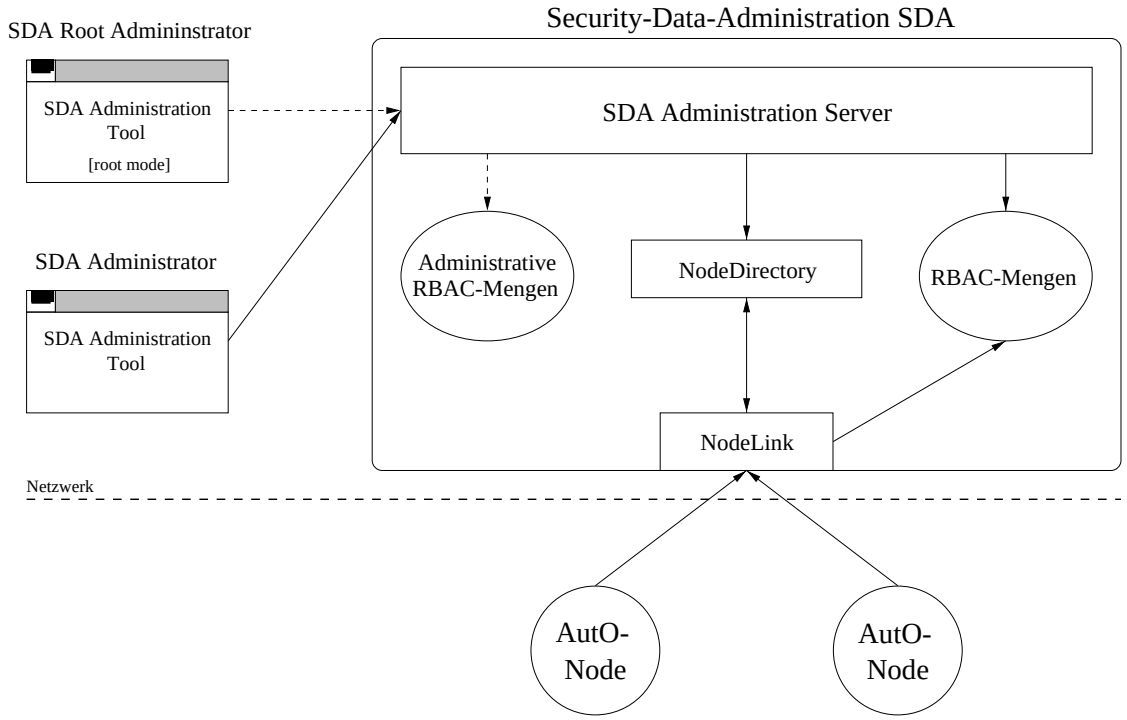


Abbildung 5.9: Der Aufbau der Security-Data-Administration SDA

administrativen RBAC-Mengen, die zur Verwaltung der normalen Mengen herangezogen werden. Abbildung 5.9, die den schematischen Aufbau der SDA wiedergibt, zeigt neben den RBAC-Mengen die weiteren Komponenten der SDA:

Der NodeLink

Der *NodeLink* ist ein multi-threaded Server, der in einem LRUCache Verbindungen zu den SDALinks einzelner Node-Manager puffert. Nimmt ein Rechner bzw. der darauf befindliche Node-Manager Verbindung mit der SDA auf, wird diese Verbindung in den Puffer eingefügt. Sollte dieser dabei eine alte Verbindung entfernen, weil er vollständig gefüllt ist, wird die entfernte Verbindung geschlossen.

Will die SDA einem Node-Manager bzw. den RBAC-Komponenten, die sich auf dessen Rechner befinden, eine Nachricht zukommen lassen, kann dies auf zwei Arten geschehen:

- Der LRUCache des NodeLinks enthält eine Verbindung zu dem gewünschten Rechner bzw. dem darauf laufenden SDALink. In diesem Fall kann die Nachricht über diese Verbindung übertragen werden.

- Es existiert keine Verbindung zu dem SDALink des gewünschten Rechners. In diesem Fall baut der NodeLink eine Verbindung auf und fügt sie anschließend in den LRUCache ein, nachdem eine gegebenenfalls daraus vorher entfernte Verbindung

geschlossen wurde. Die Nachricht kann dann über diese neu aufgebaute Verbindung gesendet werden.

Das NodeDirectory

Jeder Node-Manager enthält in seinem Benutzer-Cache die Informationen zu mehreren Benutzern, die sich auf dem entsprechenden Rechner beim AutO-System angemeldet haben. Ergeben sich nun Änderungen an den RBAC-Mengen, so können dadurch diese Benutzer-Informationen ungültig werden, d. h. sie spiegeln nicht mehr den Zustand der SDA wider.

Ein Beispiel für eine solche Änderung wäre das Löschen der Zuordnung einer Rolle zu einem Benutzer. Sollte ein Rechner die Benutzer-Information des betreffenden Benutzers gepuffert haben, ist darin gespeichert, welchen Rollen (inklusive Basisrollen) der Benutzer zugeordnet ist. Wird nun in der SDA eine solche Zuordnung gelöscht, ist die Benutzer-Information nicht mehr aktuell.

Wenn sich Änderungen an den RBAC-Mengen ergeben, werden deshalb alle Benutzer-Caches, die von der Änderung betroffene Benutzer-Informationen gepuffert haben, benachrichtigt, daß sie diese Benutzer-Informationen aus ihrem Puffer entfernen sollen. Dies geschieht mit Hilfe von `RevokeUserInfoSDAMessage`-Nachrichten, die über den NodeLink an die entsprechenden Rechner gesendet werden. Um alle Rechner zu kennen, deren Benutzer-Caches die nun ungültigen Benutzer-Informationen gepuffert haben, speichert die SDA in einem *NodeDirectory* zu jeder Benutzer-Information diejenigen Rechner, die sie gepuffert haben. Im Falle einer Änderung ist damit die Benachrichtigung dieser Rechner möglich. Nachdem die SDA alle Nachrichten abgeschickt hat, wartet sie solange, bis von jedem benachrichtigten Rechner eine Bestätigung in Form einer `RemovedUserInfoSDAMessage`-Nachricht eingetroffen ist, bevor sie die Änderung gestattet.

Sollten nicht alle Bestätigungen innerhalb einer bestimmten Zeitspanne eintreffen, können die Änderungen zum aktuellen Zeitpunkt nicht durchgeführt werden.

Der SDA-Administration-Server

Der *SDA-Administration-Server* ermöglicht die Änderung der RBAC-Mengen, sowohl der normalen als auch der administrativen, die in der Security-Data-Administration enthalten sind. Dazu notwendig ist die Benutzung des *SDA Administration Tools*, das sich mit dem Server verbindet und so Änderungen ermöglicht. Der SDA-Administration-Server erlaubt maximal eine Verbindung zu jedem Zeitpunkt. Es ist also nicht möglich, daß verschiedene Administratoren zur gleichen Zeit Änderungen an den normalen RBAC-Mengen vornehmen. Auch der SDA-Root-Administrator kann die administrativen RBAC-Mengen nicht modifizieren, wenn gerade ein SDA-Administrator eine Verbindung zum SDA-Administration-Server hält.

Wurden die Änderungen durchgeführt und wünscht der Administrator, daß diese an die RBAC-Mengen der SDA propagiert werden, sind zwei Fälle zu unterscheiden:

- Die Änderungen wurden vom SDA-Root-Administrator an den administrativen

RBAC-Mengen durchgeführt. Da nur die SDA im Besitz dieser Mengen ist, können die Änderungen ohne weitere Maßnahmen an die SDA propagiert werden.

- Hat ein SDA-Administrator Änderungen an den normalen RBAC-Mengen vorgenommen, werden zuerst die Benutzer-Informationen bestimmt, die von diesen Änderungen betroffen sind. Anschließend werden alle Rechner, deren Benutzer-Cache eine solche Benutzer-Information gepuffert hat, davon benachrichtigt, daß sie diese Benutzer-Information aus ihrem Puffer zu entfernen haben. Werden alle Benachrichtigungen innerhalb einer definierten Zeitspanne beantwortet, können die Änderungen an die SDA propagiert werden. Werden allerdings nicht alle Benachrichtigungen beantwortet, kann die Änderung nicht durchgeführt werden. Der Vorgang wird abgebrochen.

Jeder SDA-Administrator, der eine Verbindung zum SDA-Administration-Server aufbaut, muß in der administrativen **AdminUsers**-Menge enthalten sein. Ist dies nicht der Fall, kann er keine Modifikationen vornehmen. Weiterhin müssen die von ihm im Laufe der Verbindung aktivierten Rollen die Permissions zur Verfügung stellen, die ihn zu den Änderungen, die er durchführen möchte, autorisieren. Um dies zu gewährleisten, werden entsprechend den Role- und PermissionAuthorizern von AutO-Prozessen ebenfalls ein RoleAuthorizer und ein PermissionAuthorizer verwendet, die allerdings die verschiedenen Autorisierungen mit Hilfe der administrativen und nicht der normalen Mengen überprüfen.

Das SDA Administration Tool

Das Hilfsprogramm *SDA Administration Tool* wird sowohl von normalen SDA-Administrators als auch vom SDA-Root-Administrator benutzt, um Änderungen an den RBAC-Mengen der Security-Data-Administration durchzuführen. Eine genaue Beschreibung zu seiner Benutzung findet sich in Anhang D.3.

Kapitel 6

Anwendungen mit AutO

AutO stellt ein persistentes, objektorientiertes System zur Verfügung, das u. a. ein Transaktionssystem, Recovery, Migration und ausgereifte Sicherheitsmechanismen bietet, um damit schnell und effizient verteilte Anwendungen zu entwickeln. Als Beispiel für eine solche Applikation wurde im Rahmen dieser Diplomarbeit und in Zusammenarbeit mit Stefan Seltzsam (siehe [Sel99]) eine Beispielanwendung entworfen, die alle wesentlichen Merkmale von AutO benutzt. Zusätzlich wurde ein Programm entwickelt, das die Abläufe innerhalb dieser Anwendung sowie in AutO graphisch veranschaulicht.

6.1 Die Beispielanwendung - ein Supportunternehmen

Als Beispielanwendung wurde eine Applikation entwickelt, wie sie in heutiger Zeit in vielen Unternehmen im Einsatz sein könnte. Sie soll für ein Unternehmen, das professionellen Support im Bereich Informationstechnologie anbietet, die Verwaltung der einzelnen Anfragen von Personen automatisieren.

Das System soll automatisch ein Auftragsobjekt generieren, wenn eine Person bzw. ein Benutzer sich mit einem Problem bei dem Unternehmen meldet. Dies kann etwa über eine Supporthotline, eine E-Mail oder ein Java-Applet geschehen. Das Auftragsobjekt soll dann nach bestimmten Gesichtspunkten im System verteilt werden, und zwar so, daß immer eine möglichst gute Auslastung aller Angestellten des Unternehmens gewährleistet ist. Ein Auftrag soll nach seiner Erstellung verschiedene Bearbeitungsschritte durchlaufen, bis am Ende das Problem des Benutzers entweder gelöst oder aber als unlösbar erkannt worden ist. Während aller Schritte der Bearbeitung soll der Benutzer, der den Auftrag abgegeben hat, immer die Möglichkeit haben, sich über den aktuellen Bearbeitungszustand zu informieren.

6.1.1 Die Unternehmensstruktur

Zuerst wurde die interne Struktur des Unternehmens in entsprechende AutO-Objekte umgesetzt:

- Das Unternehmen besteht aus einer Reihe von Filialen, jede repräsentiert durch eine Instanz der Klasse `Branch`, die geographisch beliebig verteilt sein können. Jede Filiale enthält mehrere Teams, die für verschiedene Aufgabenbereiche zuständig sind.

  Jedes autonome Objekt vom Typ `Branch` sollte auf einem Rechner der entsprechenden Filiale beheimatet sein. Außerdem sollte es nicht migrieren können. Analoges gilt für die zu einer Filiale gehörenden Team- und Angestellten-Objekte.

- Ein Team (Klasse `Team`) ist für die Bearbeitung von Aufträgen in bestimmten Themengebieten zuständig. Themengebiete werden dabei durch Schlüsselwörter repräsentiert, wie etwa „Computer“, „Software“, etc. Jedes Team ist genau einer Filiale zugeteilt und verfügt über eine Reihe von Angestellten, denen eingehende Aufträge möglichst gleichmäßig zugeteilt werden.

- Ein Angestellter, implementiert in der Klasse `Employee`, ist für die Bearbeitung von Aufträgen zuständig. Er ist immer genau einem Team zugeteilt.

Eine beispielhafte Unternehmensstruktur ist in Abbildung 6.1 dargestellt. Sie besteht aus den drei Filialen *Passau*, *Landshut* und *Regensburg*. Jede Filiale besitzt zwei Teams. Bis auf zwei Teams besitzt auch jedes mindestens einen Angestellten, der sich um eingehende Aufträge kümmert. Die Darstellung der Struktur erledigt ein *Visualisierungstool*, das entwickelt wurde, um die internen Abläufe in der Beispielanwendung darzustellen. Eine genaue Beschreibung dieses Programms findet sich in Abschnitt 6.2.

6.1.2 Der Auftrag

Ein Auftrag, implementiert in der Klasse `Request`, wird von einem Benutzer erstellt und dann automatisch im System bzw. damit auch im Unternehmen verteilt. In der Beispielanwendung startet der Benutzer dazu das Java-Programm *RequestFactory*, das es dem Benutzer mit Hilfe einer intuitiven graphischen Oberfläche ermöglicht, einen Auftrag zu generieren, der sein Problem beschreibt (Abbildung 6.2).

Ein Auftrag enthält verschiedene Informationen, die es dem System und später dann dem Angestellten ermöglichen sollen, ihn effizient zu bearbeiten:

- *Priority*: Die Priorität des Auftrags soll für einen Angestellten einen Hinweis darauf liefern, wie dringend die Erledigung des Auftrags ist. Je kleiner dabei die gewählte Zahl ist, desto dringender ist der Auftrag. Gültige Werte liegen zwischen eins und zehn.

- *Name*: Der Name des Auftraggebers, dessen Problem durch diesen Auftrag repräsentiert wird.

- *Keywords*: Eine Reihe von Schlüsselwörtern, die die Art des Problems beschreiben. Anhand dieser Schlüsselwörter erfolgt die Verteilung des Auftrags zu geeigneten Teams, also solchen, die sich mit den angegebenen Schlüsselwörtern bzw. den dadurch repräsentierten Themengebieten beschäftigen.

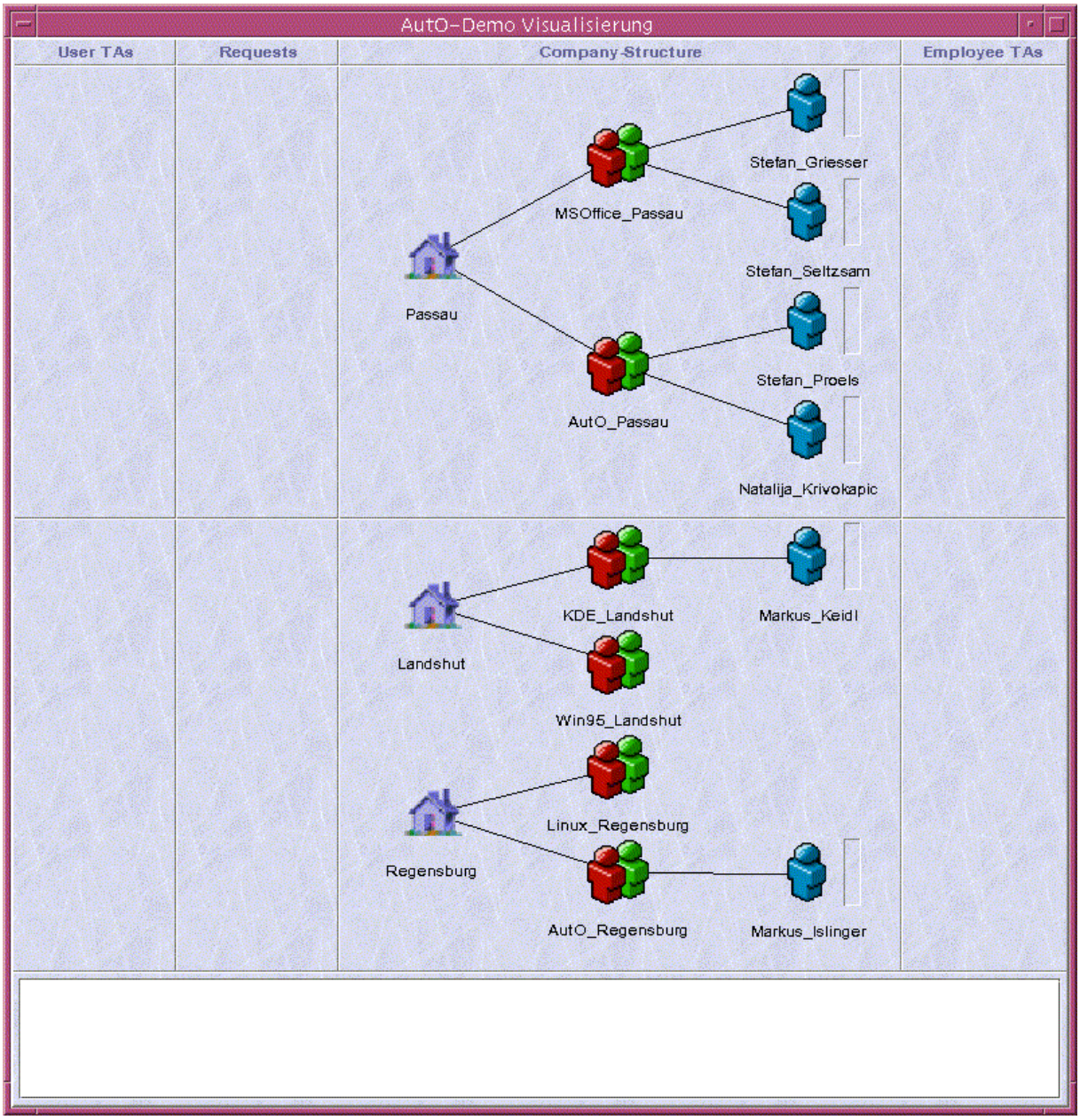


Abbildung 6.1: Die Struktur eines Beispielunternehmens

- *Problem description*: Eine Beschreibung des Problems, mit dem sich der Benutzer an das Supportunternehmen wendet.

Nachdem ein Benutzer mit Hilfe der *RequestFactory* sein Problem beschrieben hat, kann er mit einem Klick auf **ok** einen Auftrag generieren und verteilen lassen. Der Button **clear** hingegen verwirft die gemachten Eintragungen.

Nachdem der Auftrag erzeugt und im Unternehmen verteilt wurde, erhält der Benutzer eine *Tracking Number*, die eindeutig den Auftrag, den er gerade erstellt hat, identifi-

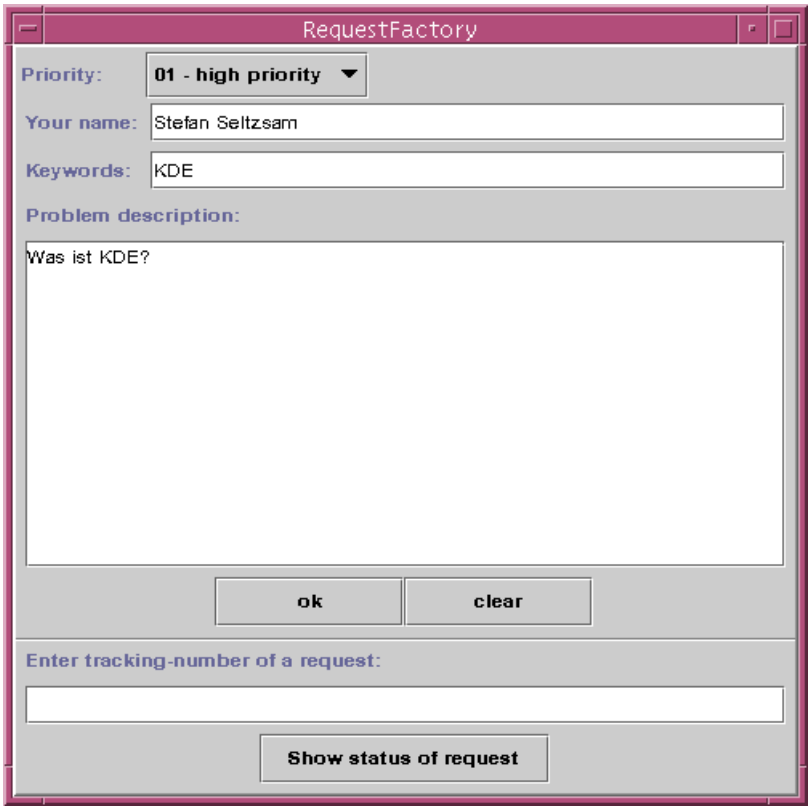


Abbildung 6.2: Die *RequestFactory* zur Generierung von Aufträgen

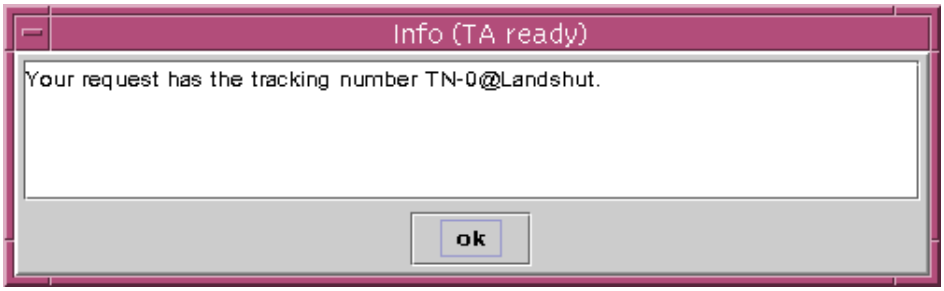


Abbildung 6.3: Die *Tracking Number* eines Auftrags

ziert (Abbildung 6.3). Mit ihr kann er jederzeit den aktuellen Bearbeitungszustand des Auftrags erfahren. Dazu trägt er die *Tracking Number* in das entsprechende Feld der *RequestFactory* ein und klickt auf **Show status of request**. Ihm wird dann der aktuelle Bearbeitungszustand des Auftrags angezeigt (siehe Abbildung 6.4).

Die Verteilung eines Auftrags

Die Verteilung eines Auftrags im System der Beispielanwendung erfolgt nach dem in Abbildung 6.5 dargestellten Schema:

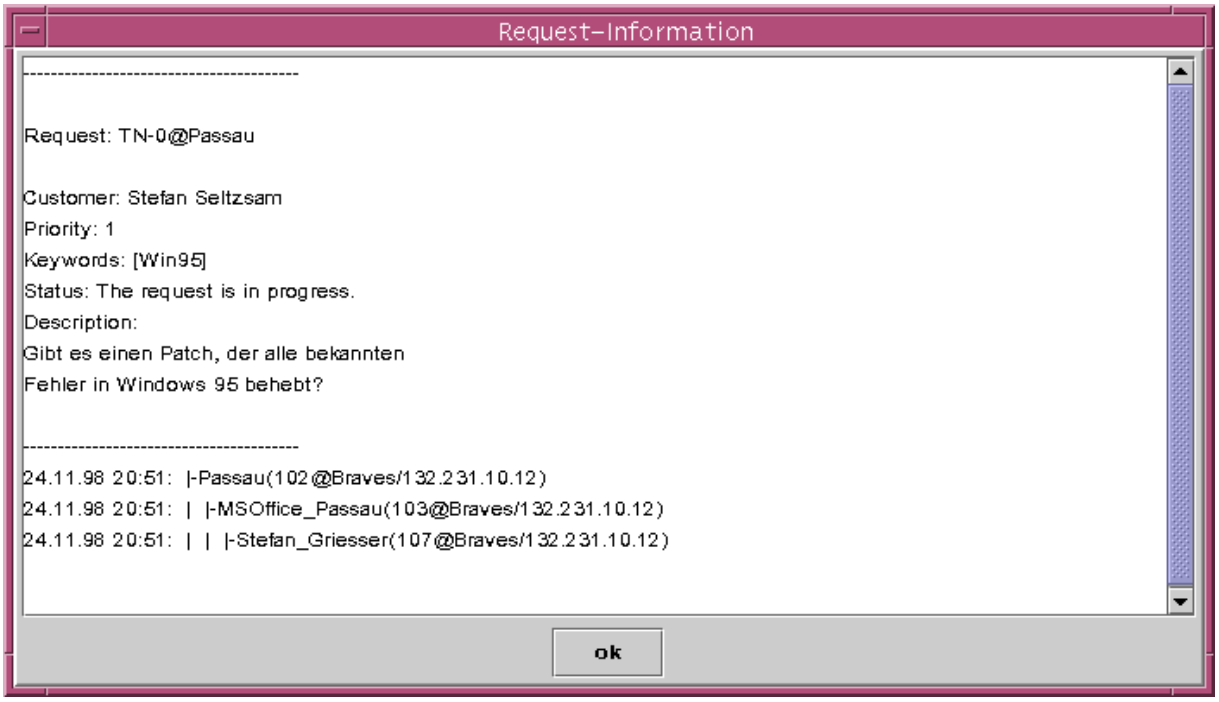


Abbildung 6.4: Anzeigen des Bearbeitungszustands eines Auftrags

Zuerst wird für einen neu generierten Auftrag eine Heimatfiliale bestimmt. Dies ist diejenige Filiale, die dem Auftrag, also dessen `Request`-Objekt, am nächsten liegt.

Beginnend mit der Heimatfiliale wird nun solange nach einer Filiale gesucht, bis eine gefunden wird, die ein Team enthält, das den Auftrag bearbeiten kann. Dies ist der Fall, wenn das Team mindestens ein Schlüsselwort des Auftrags in seiner eigenen Menge von Schlüsselwörtern enthält. Wurde eine passende Filiale gefunden, wird in dieser Filiale das geeignetste Team gesucht, also das, welches mit dem Auftrag die meisten Schlüsselwörter gemeinsam hat.

Bei diesem Team wird anschließend nach einem Angestellten gesucht, der noch nicht vollständig ausgelastet ist, also noch weitere Aufträge bearbeiten kann. Findet sich kein solcher Angestellter, wird das nächste passende Team gesucht. Findet sich in der aktuellen Filiale kein geeignetes Team mehr, wird eine neue Filiale gesucht. Scheitert dies, ist die Verteilung des Auftrags fehlgeschlagen. Da es sich bei dieser Applikation um eine Beispielanwendung handelt, wurde darauf verzichtet, etwa eine Warteschlange einzurichten, die derartige Aufträge aufnimmt, bis ein geeigneter Angestellter zur Verfügung steht.

Wurde schließlich ein geeigneter Angestellter gefunden, registriert sich der Auftrag bei diesem. Für die nun folgende Bearbeitung des Auftrags ist der entsprechende Angestellte zuständig.

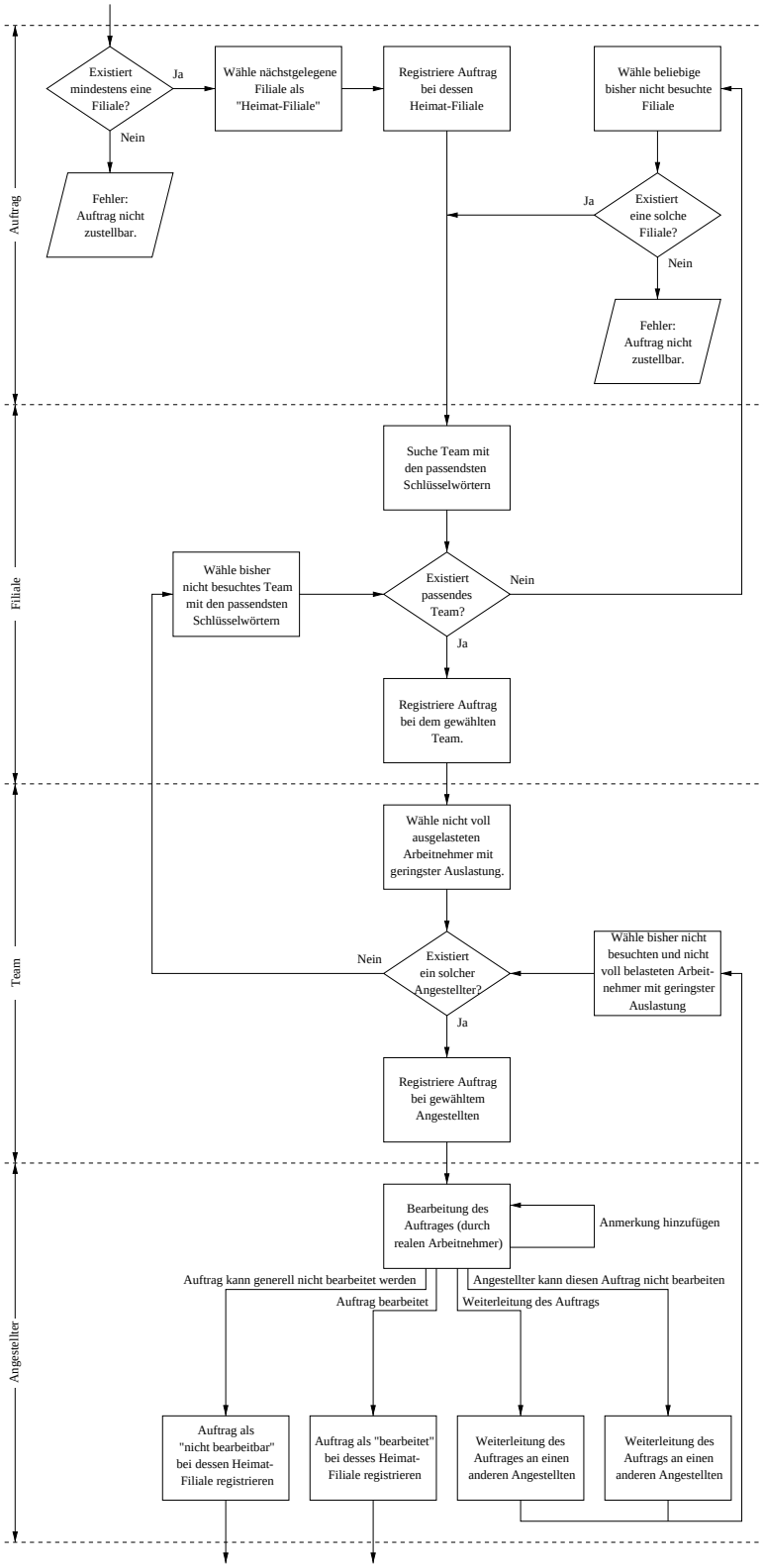


Abbildung 6.5: Die Verteilung eines Auftrags (schematisch) [Sel99]

6.1.3 Die Bearbeitung eines Auftrags

Jeder Angestellte des Supportunternehmens wird durch ein eigenes autonomes Objekt der Klasse `Employee` repräsentiert. Der Zugriff auf dieses Objekt, um etwa von den Aufträgen zu erfahren, die zu bearbeiten sind, oder um Änderungen am Bearbeitungszustand eines Auftrages durchzuführen, erfolgt mit dem Programm *EmployeeDesktop*. Dessen graphische Oberfläche zeigt Abbildung 6.6. Der *EmployeeDesktop* etabliert nach dem Start, analog

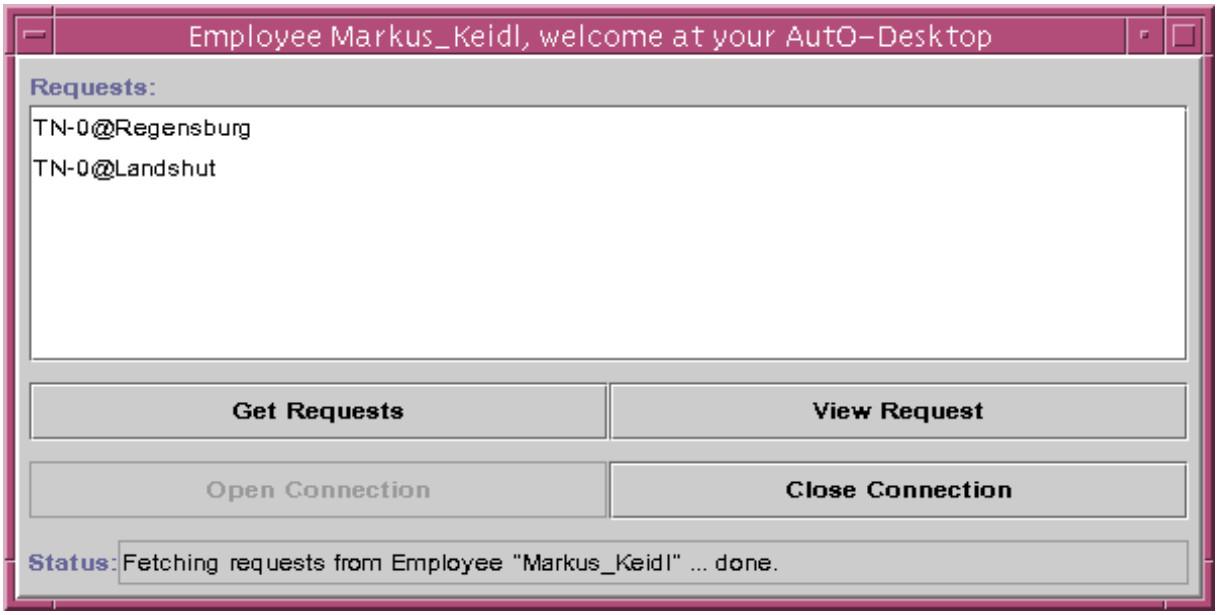


Abbildung 6.6: Der *EmployeeDesktop* eines Angestellten

einem Terminal im Client/Server-Modus (siehe dazu Abschnitt 2.2), eine Verbindung zu einem Terminal-Server. Über diese Verbindung kann dann das AutO-System und damit die Beispielanwendung angesprochen werden.

Im Bereich **Requests** werden dem Angestellten alle Aufträge angezeigt, die er zu bearbeiten hat. Mit **Open Connection** und **Close Connection** kann die Verbindung zum Terminal-Server geöffnet bzw. geschlossen werden. Mit dem Button **GetRequests** kann der Angestellte die Liste der Aufträge erneut laden, d. h. das den Angestellten repräsentierende autonome Objekt wird kontaktiert und um die aktuelle Liste der zu bearbeitenden Aufträge gebeten. Mit dem Button **ViewRequest** kann er zur Bearbeitungsansicht eines Auftrags umschalten. Diese wird in Abbildung 6.7 gezeigt. Hier bieten sich dem Angestellten nun folgende Möglichkeiten:

- **Fetch request history**: Der *EmployeeDesktop* lädt den Bearbeitungszustand des betrachteten Auftrags von neuem.
- **Add a comment**: Hiermit kann der Angestellte einen Kommentar in den Auftrag einfügen. Diese Kommentare wie auch zusätzliche Informationen werden angezeigt, wenn der Bearbeitungszustand eines Auftrags dargestellt wird.

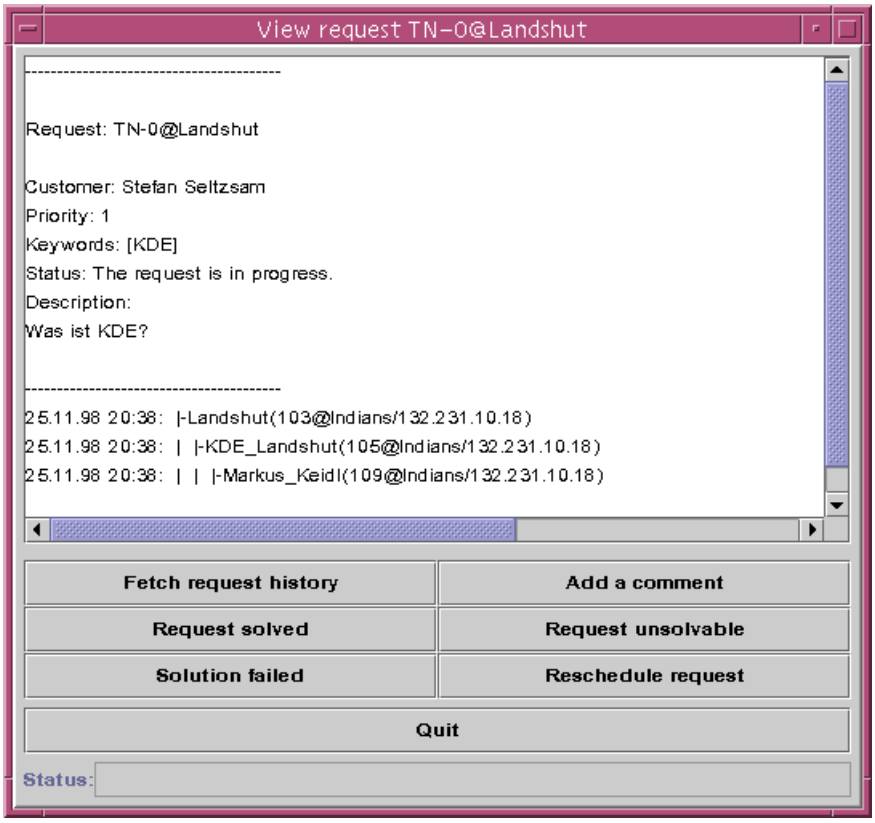


Abbildung 6.7: Die Bearbeitungsansicht eines Auftrags

- **Request solved**: Ein Auftrag wird als gelöst markiert, d. h. das Problem, das in dem Auftrag angegeben ist, wurde erfolgreich gelöst. Frägt der Auftraggeber den Bearbeitungszustand in diesem Status ab, wandert der Auftrag in das Archiv, in dem alle erledigten Aufträge gespeichert werden.

- **Request unsolvable**: Stellt sich das Problem eines Auftrags als unlösbar heraus, wird der Auftrag hiermit als unlösbar markiert. Bei der nächsten Abfrage des Bearbeitungszustands durch den Auftraggeber wandert der Auftrag in das Archiv.

- **Solution failed**: Ist der Angestellte nicht in der Lage, das Problem des Auftrags zu lösen, teilt er dies hiermit dem Auftrag mit. Dieser wird sich in einem solchen Fall einen anderen Angestellten suchen (siehe dazu auch Abbildung 6.5).

- **Reschedule request**: Ein Angestellter kann damit eine Neuverteilung des Auftrags auslösen. Dies ist etwa nötig, wenn er plötzlich krank wird und damit nicht mehr in der Lage ist, seine Aufträge in der nächsten Zeit zu bearbeiten. Gleiches gilt, wenn er etwa in Urlaub geht. Damit Aufträge in diesem Fall nicht unbearbeitet bei dem Angestellten liegen bleiben, werden sie an andere Angestellte verteilt.

- **Quit**: Ein Klick auf diesen Button beendet die Bearbeitungsansicht des Auftrags.

6.2 Das Visualisierungstool

Die internen Abläufe der Beispielanwendung, wie auch von AutO, sind ohne geeignete Hilfsmittel kaum zu erkennen. Um sie zumindest annähernd darzustellen, wurde das *Visualisierungstool* entwickelt. Es zeigt, welche autonomen Objekte und Transaktionen erzeugt wurden. Nachrichten, die zwischen Objekten ausgetauscht werden, werden ebenso angezeigt wie die Migration von Objekten von einem Rechner auf einen anderen. Die in diesem Abschnitt gezeigten Abbildungen wurden zum Großteil aus dem *Visualisierungstool* übernommen.

Abbildung 6.8 zeigt das Visualisierungstool. Das dargestellte Fenster teilt sich in verschiedene Bereiche auf, in denen jeweils unterschiedliche Informationen gezeigt werden:

- Den Bereich am unteren Ende des Fensters nimmt das *Archiv* ein. In diesem werden Aufträge angezeigt, die abgearbeitet sind. Nachdem ein Auftrag als gelöst oder aber als unlösbar markiert wurde und diese Information vom Benutzer, der den Auftrag erstellt hat, abgefragt wurde, wird ein Auftrag in dieses Archiv eingefügt.
- Die Fläche über dem *Archiv* ist in mehrere horizontale Bereiche aufgeteilt, wobei jeder dieser horizontalen Bereiche einen Rechner repräsentiert, auf dem AutO ausgeführt wird. Jeder dieser horizontalen Bereiche ist wiederum in mehrere Spalten unterteilt, die nachfolgend von links nach rechts erläutert werden:
  - Die erste Spalte zeigt Transaktionen an, die von dem Programm *RequestFactory* gestartet werden. Beispiele sind die Transaktion, die einen Auftrag bzw. das diesen repräsentierende `Request`-Objekt erstellt, oder diejenige, die den aktuellen Bearbeitungszustand eines Auftrags anzeigt.
  - In der zweiten Spalte werden die Aufträge angezeigt, die auf einem Rechner liegen. Zu jedem Auftrag wird dessen *Tracking Number* angegeben. Außerdem erhält jeder angezeigte Auftrag eine eindeutige Farbe. Dadurch kann man feststellen, welcher Angestellte ihn gerade bearbeitet. Bei jedem Angestellten werden in einer kleinen Spalte diejenigen Aufträge – repräsentiert durch ihre Farbe – angezeigt, die von diesem gerade bearbeitet werden.
  - Die dritte Spalte gibt die Struktur des Unternehmens wieder. Von links nach rechts werden zuerst die Filialen, dann die Teams und zuletzt die Angestellten dargestellt, deren Objekte auf dem Rechner liegen.
  - Die letzte Spalte wird dazu genutzt, die Transaktionen darzustellen, die ein Angestellter mit Hilfe des *EmployeeDesktops* startet. Transaktionen werden etwa erzeugt, wenn der Angestellte einen Kommentar in einen Auftrag einfügt oder wenn ein Auftrag als gelöst markiert wird.
- Nachrichten, die zwischen Objekten ausgetauscht werden, werden durch Linien dargestellt, die die graphischen Symbole der Objekte verbinden.

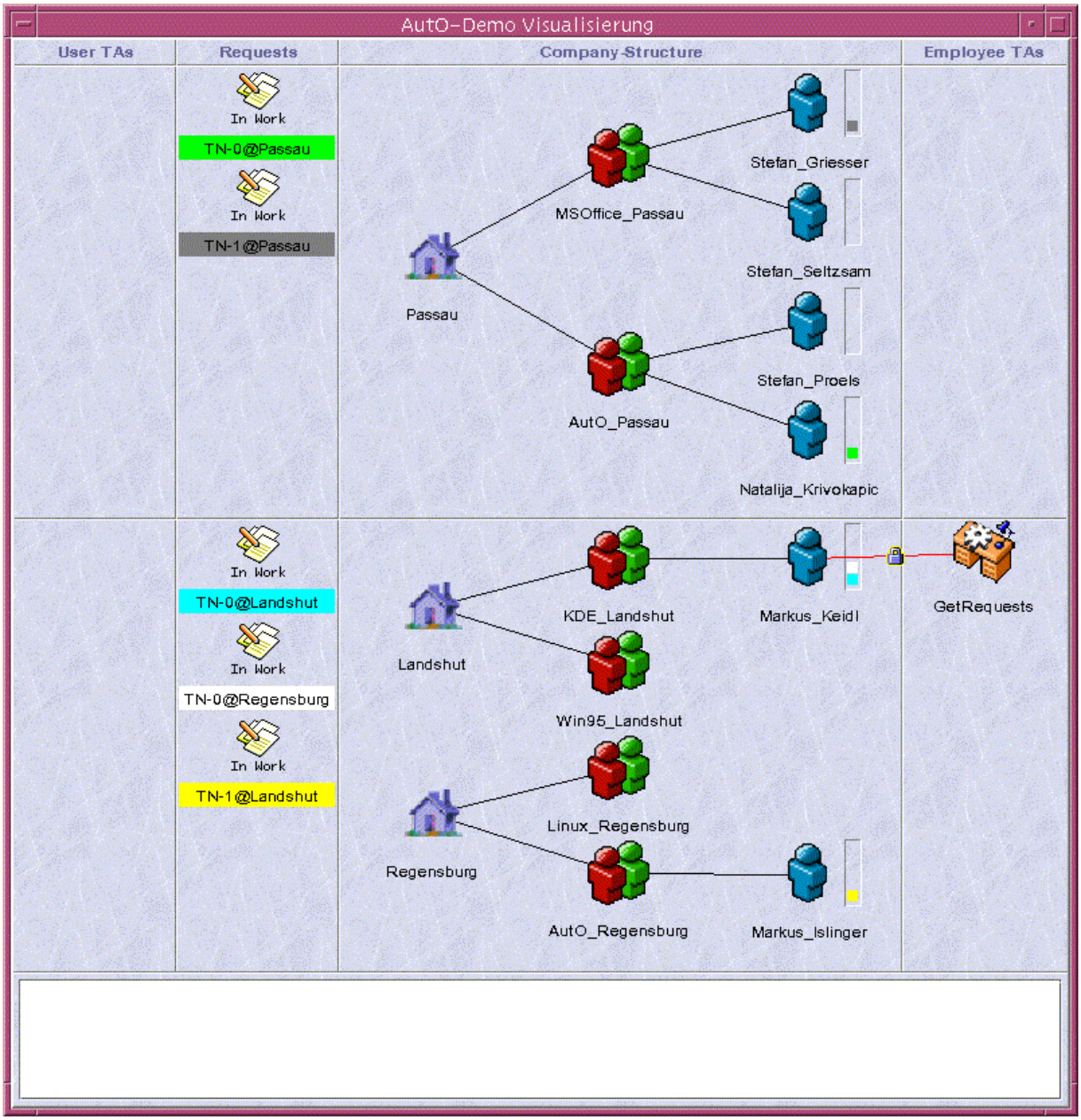


Abbildung 6.8: Das *Visualisierungstool*

Kapitel 7

Zusammenfassung und Ausblick

AutO ist ein verteiltes System autonomer Objekte. Autonome Objekte reagieren auf äußere Reize, indem sie bestimmte Aktionen ausführen. Ein Transaktionssystem sowie Persistenz und Recovery gewährleisten die Einhaltung der ACID-Eigenschaften. Migration sorgt bei Lastverschiebungen für eine dynamische Anpassung des Systems zur Laufzeit. AutO ist vollständig in der Programmiersprache Java programmiert und somit plattformunabhängig. Deshalb ist es auch für heterogene WAN-Netzwerke wie etwa das Internet geeignet.

Sicherheit ist in einem derartigen System von entscheidender Bedeutung. Es kann z. B. nicht davon ausgegangen werden, daß Daten, die über ein WAN übertragen werden, vor Manipulationen geschützt sind. Es müssen also Mechanismen existieren, die die Sicherheit in diesem System gewährleisten.

Die Zielsetzung dieser Diplomarbeit bestand darin, in AutO grundlegende und bekannte Sicherheitsmechanismen zu integrieren. Kryptographische Methoden sollten eingesetzt werden, um das System vor Angriffen von außen zu schützen. Weiterhin sollte ein geeignetes Autorisierungssystem den Zugriff auf die in AutO abgelegten Informationen regeln.

Nach einer kurzen Einführung in das Gebiet der Kryptographie wurde beschrieben, wie kryptographische Methoden in AutO eingesetzt werden, um Daten, die über möglicherweise unsichere Netzwerke übertragen werden, vor Angriffen zu schützen. Es wurde ein spezieller AutO-Socket entwickelt, der sichere Verbindungen ermöglicht.

Das rollenbasierte Autorisierungssystem, das im Rahmen dieser Diplomarbeit in AutO integriert wurde, ermöglicht eine flexible Anpassung der Zugriffsrechte auf Daten und Informationen entsprechend einer internen Unternehmensstruktur. Es wurden entsprechende Hilfsprogramme entwickelt, die die Verwaltung des Autorisierungssystems auf effiziente und einfache Weise ermöglichen.

Abschließend wurde eine Beispielanwendung vorgestellt, die entwickelt wurde, um die Fähigkeiten von AutO zu demonstrieren. Sie sollte die Auftragsannahme und -bearbeitung eines fiktiven Supportunternehmens unterstützen. Eine graphische Visualisierungskomponente gewährt Einblick in die internen Abläufe der Anwendung.

Ausblick

Im Rahmen einer Diplomarbeit ist es kaum möglich, alle Sicherheitsaspekte abzudecken, die in einem System wie AutO eine Rolle spielen. Es existieren naturgemäß Bereiche, die nicht oder nur ungenügend berücksichtigt werden können. Zum Abschluß wird nun auf solche Bereiche hingewiesen:

- Auch wenn man beim Entwurf und der Integration eines Sicherheitssystems zu beweisen versucht, daß es korrekt ist und keine Lücken enthält, zeigt sich in der Realität doch, daß dies in den wenigsten Fällen ausreichend ist. Man muß für Fälle, in denen das Sicherheitssystem umgangen oder durchbrochen wird, Mechanismen vorsehen, die dies zumindest erkennen lassen. Diese werden meistens mit dem Begriff *Auditing* bezeichnet.

- Die im Autorisierungssystem bisher verwendeten Permissions sind atomar, d. h. sie können nicht mehr weiter aufgeteilt werden. Es wäre denkbar, daß man eine Abfolge von bestimmten Permissions und den damit verbundenen Aktionen zu *Benutzer-Permissions* zusammenfaßt, die dann analog den bisherigen Permissions vergeben werden könnten. Diese *Benutzer-Permissions* würden dann komplexe Abläufe in einem Betrieb widerspiegeln. Außerdem könnten sie dafür Sorge tragen, daß die entsprechenden Aktionen in genau der definierten Reihenfolge ausgeführt würden.

- Der *Information-Service* [Gri97, Isl97] des AutO-Systems steht autonomen Objekten und Transaktionen zur Verfügung, um Metadaten auszutauschen. Die Zugriffe auf den Information-Service sind nicht in das Transaktionssystem integriert, d. h. sie werden beispielsweise nicht rückgängig gemacht, wenn eine Transaktion abgebrochen wird. Deshalb wurden keine Permissions festgelegt, die den Zugriff auf den Information-Service regeln. Allerdings existieren keine prinzipiellen Probleme, die dies verhindern, so daß der Information-Service ebenfalls in das Autorisierungssystem integriert werden könnte.

- Randbedingungen, so wie sie in Kapitel 5 eingeführt wurden, sind bisher nicht im Autorisierungssystem von AutO implementiert. Zumindest die Nutzung von statischen Randbedingungen, die einfach in das *SDA Administration Tool* und den SDA Administration Server integriert werden könnten, stellt in AutO kein Problem dar.

Anhang A

Beispiele für Kryptographiealgorithmen

Nachdem in Kapitel 3 die theoretischen Grundlagen von Kryptographiealgorithmen dargestellt wurden, folgen nun in diesem Anhang einige Beispiele für derartige Algorithmen. Weitere Informationen zu den einzelnen Algorithmen sind in [Sch96] zu finden.

A.1 Symmetrische Algorithmen

DES: DES ist eine Blockchiffre, die auf 64-Bit-Blöcken operiert und seit fast 20 Jahren der Standard bei symmetrischer Kryptographie ist. Er besitzt keine bekannten Schwächen, allerdings ist die Länge des Schlüssels mit 64 Bit (8 Bit werden als Paritybits benutzt, so daß die effektive Schlüssellänge 56 Bit beträgt) heute eigentlich nicht mehr ausreichend. Eine genaue Beschreibung der Arbeitsweise findet sich in [Sch96] oder [U.S93a]

DES-EDE3: Nachdem sich zeigte, daß die Schlüssellänge von DES mit 56 Bit nicht mehr ausreichend war, wurde DES-EDE3 entwickelt. Ein DES-EDE3-Schlüssel besteht aus drei DES-Schlüsseln und besitzt damit eine Länge von 168 Bit (192 mit Paritybits). Die Funktionsweise von DES-EDE3 besteht im Prinzip aus der Hintereinanderschaltung von drei einzelnen DES-Ver- bzw. Entschlüsselungen. Genaueres dazu ist in [Sch96] zu finden.

IDEA: IDEA ist der zur Zeit wohl beste und sicherste bekannte Blockalgorithmus [Sch96]. Er arbeitet, wie DES, mit 64-Bit-Blöcken. Ein Schlüssel besitzt die feste Länge von 128 Bit. Einen Großteil seiner Berühmtheit verdankt er seiner Verwendung in *Pretty Good Privacy* (PGP) von Philip Zimmermann [Zim95].

Blowfish: Der von Bruce Schneier entwickelte Blowfish-Algorithmus verwendet eine variable Schlüssellänge mit einer maximalen Länge von 448 Bit, um 64-Bit-Blöcke zu verschlüsseln.

CAST5: Eine weitere Blockchiffre, die auf 64-Bit-Blöcken operiert. Die Schlüssellänge kann bis zu 128 Bit betragen, wobei sie ein Vielfaches von acht sein muß.

LOKI91: Ein Blockalgorithmus, der auf 64-Bit-Blöcken operiert und einen 64-Bit-Schlüssel verwendet.

RC2: Dieser Algorithmus operiert ebenfalls auf 64-Bit-Blöcken und verwendet Schlüssel mit variabler Schlüssellänge.

RC4: RC4 ist eine Streamchiffre, die beliebig lange Schlüssel unterstützt.

SAFER: Ein bisher noch nicht genau untersuchter Algorithmus, der 64-Bit-Blöcke mit einem 64- oder 128-Bit-Schlüssel verschlüsselt.

SPEED: Eine Blockchiffre mit einstellbarer Blockgröße und variabler Schlüssellänge, die zwischen 6 und 32 Byte liegen kann.

Square: Ein Blockalgorithmus mit 128-Bit-Schlüssel und -Blöcken. Genauere Erläuterungen sind unter [DR] zu finden.

A.2 Asymmetrische Algorithmen

RSA: RSA ist wohl der bekannteste asymmetrische Algorithmus. Seine Sicherheit beruht darauf, daß es sehr lange dauert, große natürliche Zahlen zu faktorisieren.

Zuerst werden zwei ausreichend große Primzahlen p und q gewählt. Deren Produkt $n = pq$ ist eine sehr große natürliche Zahl. Weiterhin bestimmt man nun zwei Zahlen e und d so, daß

$$\mathrm{ggT}(e, (p-1)(q-1)) = 1 \text{ und}$$

$$ed \bmod ((p-1)(q-1)) \equiv 1.$$

Dabei sollte $d \geq \frac{n}{3}$ sein. Häufig wird e sehr klein gewählt, so daß die Verschlüsselungsdauer sinkt. Der Klartext wird nun so in einzelne Bitgruppen aufgeteilt, daß der Werte jeder Bitgruppe, interpretiert als Binärdarstellung einer nicht-negativen ganzen Zahl, kleiner ist als n. Jede dieser Bitgruppen kann nun verschlüsselt werden durch

$$C = E(M) = M^e \bmod n$$

und entschlüsselt werden durch

$$M = D(C) = M^d \bmod n.$$

Durch die Wahl von e (sehr klein) und d (größer oder gleich als $\frac{n}{3}$) folgt auch, daß der Entschlüsselungsvorgang im Vergleich zur Verschlüsselung mehr Zeit erfordert. Der geheime Schlüssel wird durch n und d gebildet, der öffentliche durch n und e. Weiter Informationen sind in [Sch96] zu finden.

A.3 Algorithmen für digitale Signaturen

MD2/RSA: Der Signaturalgorithmus MD2/RSA besteht aus zwei einzelnen Algorithmen, die getrennt angewendet werden. Zuerst wird mit Hilfe von MD2 ein Hashwert der zu signierenden Daten berechnet. Dieser Hashwert wird dann mit dem RSA-Algorithmus signiert (siehe dazu auch Abschnitt 3.4). MD2 ist ein relativ schneller Algorithmus, der einen 128-Bit-Hashwert berechnet [Kal92].

MD5/RSA: Genau wie MD2/RSA besteht auch dieser Algorithmus aus zwei Teilen. MD5 berechnet wie MD2 einen 128-Bit-Hashwert [Riv92], allerdings ist MD5 erkennbar langsamer als MD2.

SHA-1/RSA: SHA-1 [U.S93b] ist analog zu MD2 und MD5 eine Algorithmus zur Berechnung eines 128-Bit-Hashwertes. Dieser wird dann mit dem RSA-Verfahren signiert.

DSA: DSA [U.S94] ist ein Algorithmus der ausschließlich dazu benutzt werden kann, Daten zu signieren und Signaturen zu verifizieren. Er verwendet Schlüssellängen von 512-1024 Bit für den öffentlichen und 160 für den geheimen Schlüssel.

A.4 Algorithmen für Message Authentication Codes

Ein MAC, so wie er in der Java Cryptography Extension [Sun98] definiert wird, basiert auf einer kryptographischen Hashfunktion, genannt *HMAC* [KBC97]. Ein HMAC kann etwa eine normale Hashfunktion wie MD5 oder SHA-1 in Kombination mit einem geheimen Schlüssel benutzen. Die genaue Bezeichnung des MAC-Algorithmus ist dabei immer vom verwendeten Security Provider (siehe Anhang B) abhängig.

Anhang B

Security Provider für Java

Um Kryptographie in Java verwendet zu können, hat Sun die *Java Cryptography Extension* (JCE) eingeführt. Das Paket *java.security*, das seit einiger Zeit im *Java Development Kit* (JDK) enthalten ist [Pos98, Sun97a, SDBT], stellt nur eine minimale Untermenge der existierenden kryptographischen Algorithmen zur Verfügung. Die Spezifikationen der JCE wurden von Sun in [Sun98] festgelegt. Leider ist es in den USA und Kanada untersagt, derartiges kryptographisches Material zu exportieren. Deshalb ist es nicht möglich, die Sun-JCE nach Deutschland zu importieren. Da die Spezifikationen aber öffentlich zugänglich sind, haben verschiedene Firmen und Institutionen eigene, den Sun-Spezifikationen entsprechende Kryptographiebibliotheken entwickelt:

- **Cryptix** (*http://www.systemics.com/docs/cryptix*): Dieses Paket von der Firma *Systemics* ist momentan nur in einer Version für das JDK 1.1 erhältlich. Es kann deshalb zusammen mit AutO nicht verwendet werden.
- **IAIK-JCE** (*http://jcewww.iaik.tu-graz.ac.at*): IAIK-JCE ist konform mit den neuesten Spezifikationen JCE 1.2 von Sun. Dieses Paket von IAIK ist für den nichtkommerziellen Einsatz frei nutzbar.
- **J/CRYPTO** (*http://www.baltimore.ie/products/jcrypto/index.html*): Dieses Paket von der Firma Baltimore Technologies entspricht den neuesten JCE-Spezifikationen von Sun. Allerdings ist es nur in einer kommerziellen Version erhältlich.
- **RSA BSAFE Crypto-J** (*http://www.rsa.com/rsa/products/cryptoj*): Die Firma RSA entspricht mit ihrem Produkt RSA BSAFE Crypto-J ebenfalls den neuesten JCE-Spezifikationen von Sun.

Anhang C

Systemperformance und Kryptographie

Der Einsatz von Kryptographie garantiert in AutO die sichere Kommunikation der einzelnen Systemkomponenten (siehe dazu Kapitel 4). Dieser Aspekt bildet einen zentralen Bestandteil des in dieser Diplomarbeit entwickelten Sicherheitskonzepts. Beim Verzicht auf die in Kapitel 4 entwickelten Techniken können Angriffe auf das AutO-System nicht verhindert werden. Aber ebenso wie Persistenz und Recovery [Gri97] verlangt auch Kryptographie einen Mehraufwand, der die Systemperformance beeinflußt. Bei der Entwicklung der verschiedenen Sicherheitskonzepte wurde deshalb auf effiziente Algorithmen und Protokolle Wert gelegt. Zusätzlich stehen dem Systemadministrator viele Konfigurationsmöglichkeiten zur Verfügung, mit denen er den Einsatz von Kryptographie steuern kann, um somit zwischen Systemperformance und Sicherheit abwägen zu können.

Einen wichtigen Anhaltspunkt, um den Einfluß der gewählten Konfiguration auf das System beurteilen zu können, stellen die verschiedenen, zur Auswahl stehenden Kryptographiealgorithmen dar. Ihre Kosten, d. h. der Zeitaufwand für die Verschlüsselung, die Entschlüsselung, das Signieren und Verifizieren von Daten, waren zudem die Grundlage für viele Entscheidungen bei der Entwicklung des implementierten Sicherheitskonzepts. Sie sollten auch bei der Konfiguration des Systems berücksichtigt werden. Zur Bestimmung der Kosten wurden verschiedene Messungen durchgeführt, die im folgenden zusammen mit den Ergebnissen vorgestellt werden.

C.1 Meßgrundlagen

Alle Messungen wurden auf einer *Sun UltraSPARC 1 Creator 170E* durchgeführt. Jede einzelne Messung wurde dabei mehrmals wiederholt. Die gewonnenen Werte wurden dann zu einer Meßreihe zusammengefaßt und anschließend das arithmetische Mittel dieser Meßreihenwerte berechnet[1]. In den weiter unten angeführten Tabellen ist stets nur dieses Mittel angegeben.

[1] Genaugenommen wurde bei der Berechnung des arithmetischen Mittels der jeweils erste Werte einer Meßreihe nicht berücksichtigt, da die Java Virtual Machine bei seiner Bestimmung zuerst alle benötigten Java-Klassen laden mußte und somit das Ergebnis verfälschte.

Weiterhin sind folgende Punkte bei den in diesem Abschnitt enthaltenen Tabellen zu beachten:

- Da die in AutO verschickten Nachrichten sehr unterschiedliche Größen haben, wurde jede Messung mehrmals mit Datenblöcken verschiedener Länge durchgeführt. Die verwendeten Größen sind in den verschiedenen Tabellen angegeben. Ergebnisse, die sich auf gleiche Datenblockgrößen beziehen, sind immer in derselben Spalte aufgeführt.
- Alle hier verwendeten Kryptographiealgorithmen benötigen einen Schlüssel. Die Stärke dieses Schlüssels (genaueres dazu siehe Kapitel 3) ist oft variabel. Sie wird in den folgenden Tabellen deshalb in Klammern hinter dem Namen des verwendeten Algorithmus angegeben.
- Die gemessenen Werte werden stets in Millisekunden (ms) angegeben.
- Als Security Provider wurde Cryptix [Cry, Pas97] verwendet.

AutO verwendet Kryptographie zu mehreren unterschiedlichen Zwecken. Zum einen zur Signierung und Verifizierung von Daten, zum anderen zur Ver- und Entschlüsselung von Daten, die in Nachrichten enthalten sind. Details dazu finden sich in den Kapiteln 3 und 4.

C.2 Signierung und Verifikation

Um die Kosten für das Signieren und Verifizieren von Daten zu bestimmen, wurde die Zeit gemessen, die nötig war, um verschieden große Datenblöcke zu signieren (*Sign*) und zu verifizieren (*Verify*). Tabelle C.1 und Abbildung C.1 zeigen die Ergebnisse der einzelnen Messungen für die getesteten Algorithmen.

Algorithmus		256	512	1024	2048	4096	[Byte]
MD5/RSA (512)	Sign	25	26	30	38	56	[ms]
	Verify	7	8	13	21	37	[ms]
MD2/RSA (512)	Sign	50	72	124	221	409	[ms]
	Verify	30	54	103	199	398	[ms]
DSA (512)	Sign	26	28	33	44	67	[ms]
	Verify	44	47	52	62	86	[ms]
SHA-1/RSA (512)	Sign	25	27	32	43	65	[ms]
	Verify	7	10	16	29	47	[ms]

Tabelle C.1: Signierung und Verifikation von Datenblöcken

Man kann erkennen, daß mit Ausnahme von MD2/RSA jeder Algorithmus in der Lage ist, auch längere Datenblöcke in vertretbarer Zeit zu verarbeiten. Der lineare Anstieg des Zeitaufwands, wie ihn Abbildung C.1 zeigt, unterstützt diesen Eindruck. Es können also bis auf MD2/RSA alle Algorithmen verwendet werden.

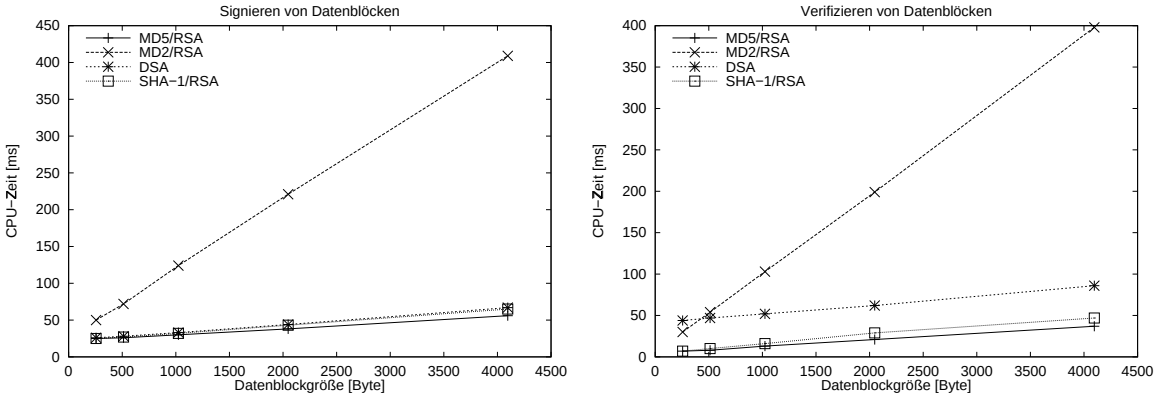


Abbildung C.1: Zeitaufwand für das Signieren und Verifizieren von Datenblöcken

C.3 Ver- und Entschlüsselung

Da AutO ein verteiltes System ist, werden oftmals Daten zwischen den einzelnen Komponenten versendet. Bei der Migration wird etwa ein komplettes Objekt mit seinen eventuell sicherheitsrelevanten Daten verschickt. Da Objekte fast immer große Datenmengen enthalten, kann die notwendige Verschlüsselung große Auswirkungen auf die Migrationsdauer haben. Deshalb muß man die möglichen Performanceeinbußen sowohl bei der Migration also auch allgemein im System abschätzen können, die möglicherweise durch Verschlüsselung entstehen.

Genau wie bei den Messungen zur Signierung und Verifikation wurden auch hier mehrere Algorithmen getestet, indem Datenblöcke verschiedener Größe ver- und entschlüsselt wurden. Die gewonnenen Ergebnisse können Tabelle C.2 entnommen werden. Zeilen, die mit *V* gekennzeichnet sind, enthalten die Verschlüsselungszeiten, Zeilen mit *E* die Entschlüsselungszeiten.

Algorithmus		256	512	1024	2048	4096	[Byte]
DES (512)	V	7	11	22	42	86	[ms]
	E	5	12	21	43	84	[ms]
Blowfish (512)	V	48	47	52	63	83	[ms]
	E	45	48	51	61	81	[ms]
CAST5 (512)	V	3	7	11	23	47	[ms]
	E	3	7	12	24	48	[ms]
DES-EDE3 (512)	V	18	34	75	144	281	[ms]
	E	18	36	68	140	282	[ms]
IDEA (512)	V	5	10	21	41	89	[ms]
	E	6	11	21	43	88	[ms]
LOKI91 (512)	V	3	11	13	27	55	[ms]
	E	3	6	13	27	53	[ms]

RC2 (512)	V	5	11	21	43	91	[ms]
	E	6	11	24	46	90	[ms]
RC4 (512)	V	2	3	8	11	23	[ms]
	E	2	4	6	12	23	[ms]
SAFER (512)	V	8	15	32	64	122	[ms]
	E	8	16	34	60	123	[ms]
SPEED (512)	V	19	39	80	155	303	[ms]
	E	18	40	82	147	292	[ms]
Square (512)	V	5	10	17	32	63	[ms]
	E	5	8	15	31	63	[ms]
RSA (512)	V	23	43	82	160	324	[ms]
	E	112	213	387	744	1478	[ms]

Tabelle C.2: Zeitaufwand für die Ver- und Entschlüsselung von Datenblöcken

Wie auch in den vorherigen Messungen ist bei der Ver- und Entschlüsselung ein linearer Anstieg der benötigten Zeit mit zunehmender Datenblockgröße zu erkennen, auch wenn er zum Teil sehr gering ausfällt. Die beiden Diagramme C.2 und C.3 verdeutlichen dies.

Deutlich erkennbar ist der höhere Aufwand für das Public-Key Verfahren RSA im Vergleich zu den restlichen, symmetrischen Verfahren. Ein Nachteil, den alle Public-Key Verfahren aufweisen ([Sch96], Seite 33). Doch unabhängig vom verwendeten Verfahren wird deutlich, daß eine komplette Verschlüsselung aller AutO-Nachrichten nicht durchführbar ist, solange es nicht wesentlich effizientere Computer gibt.

Um ebenfalls einen Eindruck davon zu bekommen, inwieweit die Stärke eines Schlüssels die Ver- und Entschlüsselungsdauer beeinflußt, wurde die vorherige Meßreihe nochmals mit verschieden starken Schlüsseln wiederholt. Dabei wurden jedoch nur zwei Algorithmen (RC4 und RSA) getestet. DES und DES-EDE3 arbeiten beide mit einer festen Schlüssellänge, so daß eine geänderte Stärke des Schlüssels keine Einfluß auf ihre Laufzeit hat. Ebenso verhält es sich mit RC2, da die verwendete Implementierung auf einen 1024-Bit-Schlüssel beschränkt ist. Die Ergebnisse in Tabelle C.3 und Abbildung C.4 demonstrieren den linearen Anstieg der Laufzeit mit steigender Schlüsselstärke bei allen gemessenen Algorithmen.

Neben dem Verschlüsseln von einzelnen Datenblöcken bietet das JCE-API auch die Möglichkeit, einen Datenstrom zu verschlüsseln. Alle Daten, die in diesen Datenstrom geschrieben werden, werden automatisch verschlüsselt, bevor sie zum Empfänger weitertransportiert werden. Dort werden sie entschlüsselt und können anschließend ganz normal aus dem Datenstrom ausgelesen werden. Dieser Ansatz bildet eine Alternative zum herkömmlichen Vorgehen, bei dem die zu sendenden Daten verschlüsselt und dann in den Datenstrom geschrieben werden. Ein Vergleich beider Alternativen zeigt Tabelle C.4. Dabei stehen in der mit C&S (*Cipher and Send*) bezeichneten Zeile die Werte, die mit dem herkömmlichen Verfahren erzielt wurden, d. h. es werden bereits verschlüsselte Datenblöcke versendet. Die zweite, mit CS (*CipherStream*) bezeichnete Zeile zeigt die Werte, die mit einem verschlüsselten Datenstrom erzielt wurden. Diagramm C.5 veranschaulicht die Ergebnisse nochmals graphisch. Die Ergebnisse zeigen, daß die Verschlüsselung eines Datenstroms Nachteile

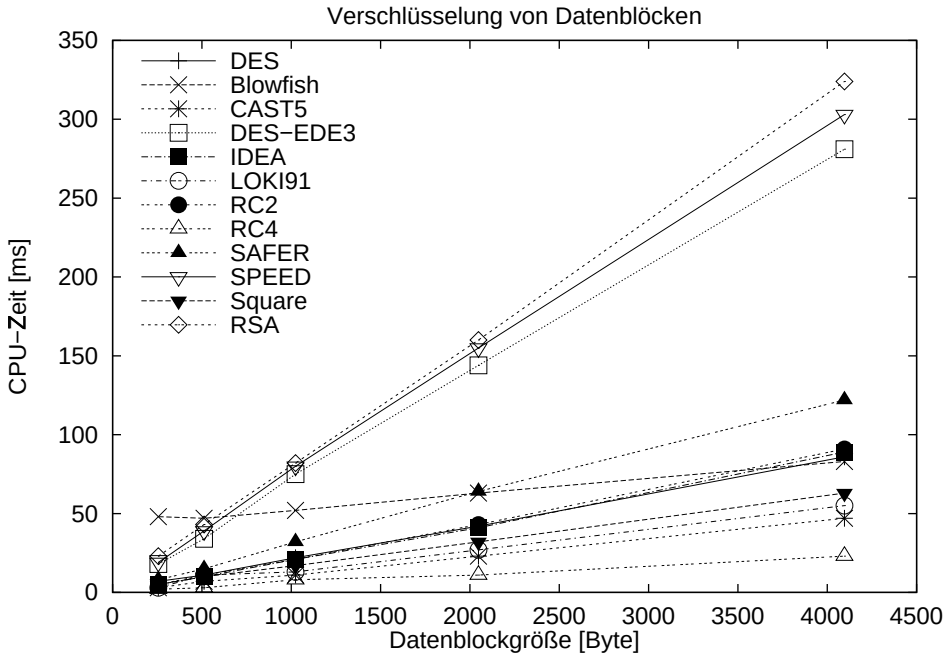


Abbildung C.2: Verschlüsselung mit verschiedenen Algorithmen

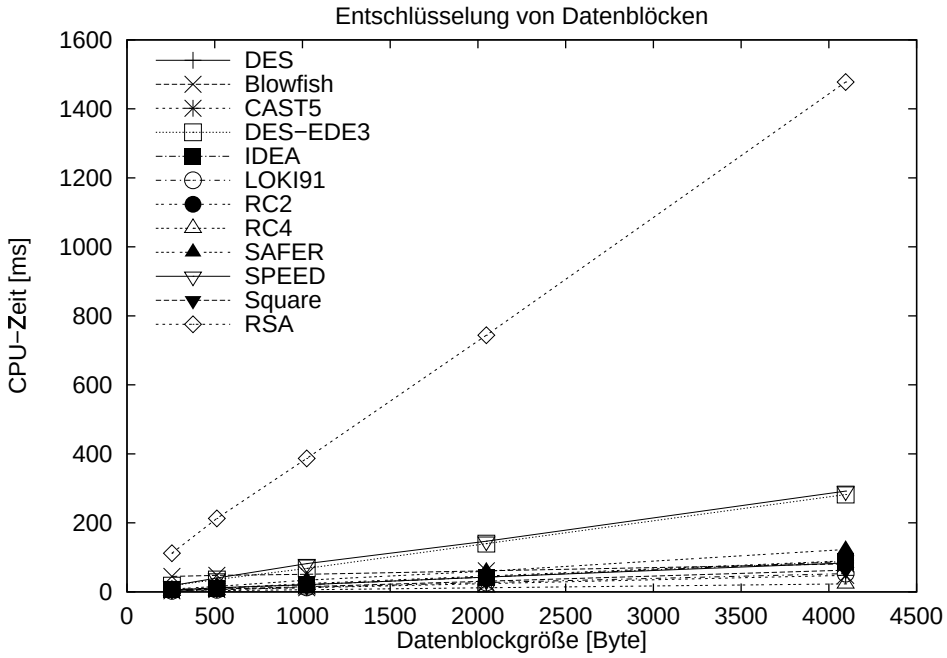


Abbildung C.3: Entschlüsselung mit verschiedenen Algorithmen

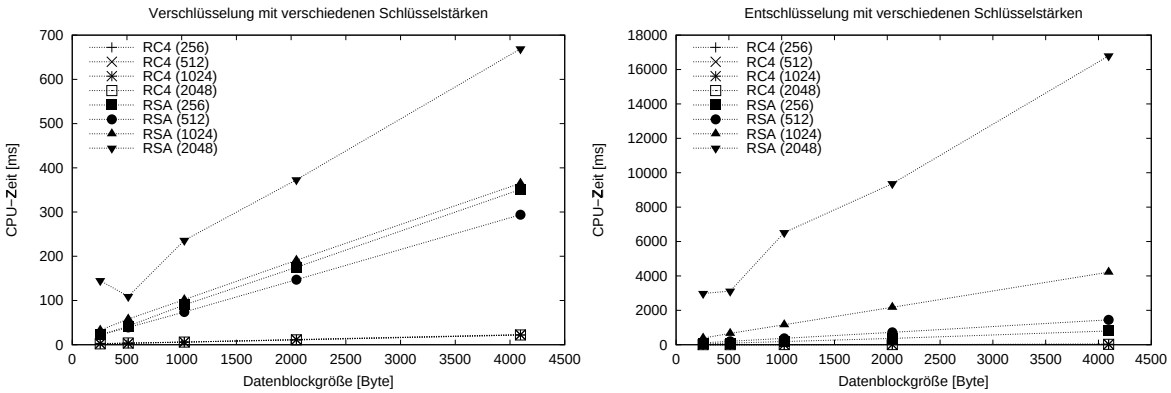


Abbildung C.4: Einfluß der Schlüsselstärke auf ausgewählte Algorithmen

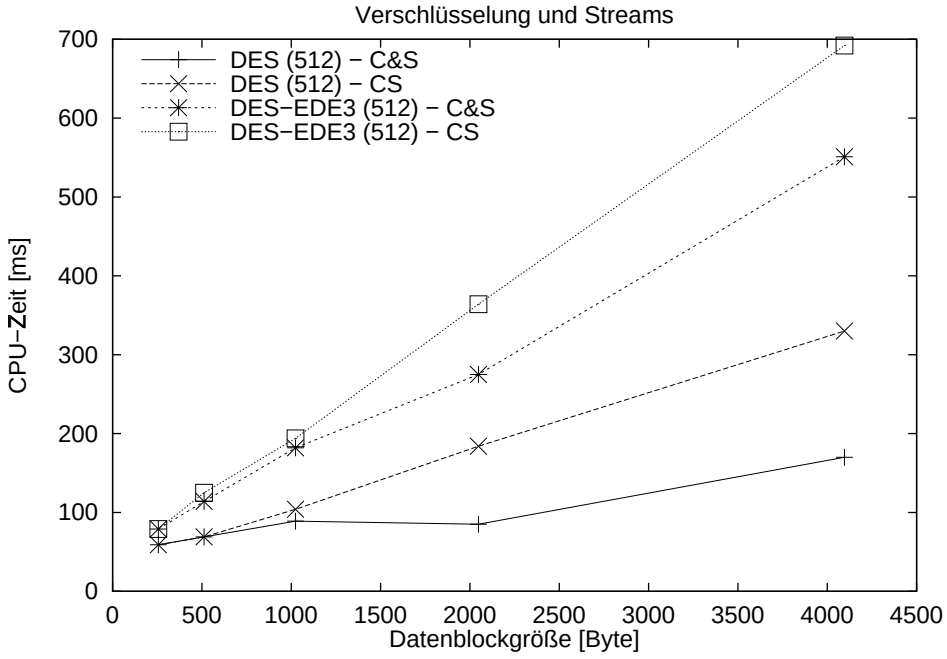


Abbildung C.5: Verschlüsselung und Datenströme

Algorithmus	256	512	1024	2048	4096	[Byte]	
RC4 (256)	V	2	4	6	11	22	[ms]
	E	2	3	6	11	23	[ms]
RC4 (512)	V	2	3	6	12	23	[ms]
	E	2	4	7	11	23	[ms]
RC4 (1024)	V	3	3	6	11	22	[ms]
	E	2	4	6	12	23	[ms]
RC4 (2048)	V	2	4	6	11	22	[ms]
	E	4	3	7	11	22	[ms]
RSA (256)	V	22	42	90	175	351	[ms]
	E	49	95	185	369	800	[ms]
RSA (512)	V	22	39	74	147	294	[ms]
	E	109	199	372	723	1445	[ms]
RSA (1024)	V	33	58	102	191	365	[ms]
	E	385	662	1162	2176	4220	[ms]
RSA (2048)	V	145	109	236	373	669	[ms]
	E	2983	3116	6523	9368	16794	[ms]

Tabelle C.3: Vergleich der Verschlüsselung mit unterschiedlich starken Schlüsseln

Algorithmus		256	512	1024	2048	4096	[Byte]
DES (512)	C&S	59	69	89	85	170	[ms]
	CS	59	69	104	184	330	[ms]
DES-EDE3 (512)	C&S	79	114	182	275	551	[ms]
	CS	79	125	194	364	692	[ms]

Tabelle C.4: Vergleich verschlüsselter und unverschlüsselter Datenströme

bezüglich der Performance besitzt. Zusätzlich lassen die doch relativ langen Verschlüsselungszeiten für größere Datenmengen die komplette Verschlüsselung eines Datenstroms - und damit in AutO auch der dahinterliegenden Kommunikationsverbindung - als nicht durchführbar erscheinen.

Anhang D

Hilfsprogramme für das AutO-Sicherheitssystem

Zur Verwaltung von AutO existiert eine Reihe von Hilfsprogrammen. Diese sind zwar alle mit einer Hilfefunktion ausgestattet, welche allerdings nur die grundsätzliche Handhabung des entsprechenden Programms beschreibt. Zum genauen Verständnis sowohl der Programme als auch der dahinterstehenden Mechanismen soll deshalb dieses Kapitel beitragen. Es beschreibt alle im Rahmen dieser Diplomarbeit entwickelten Programme.

D.1 Das *SRTool*

Der Signature-Server ist die wesentliche Komponente für alle in AutO verwendeten kryptographischen Algorithmen. Seine Aufgabe besteht darin, Informationen, die ihm von einem Security-Representative zugesendet werden, mit seinem eigenen geheimen Schlüssel zu signieren. Dadurch kann in AutO jede Komponente, die im Besitz des öffentlichen Schlüssels des Signature-Servers ist, die Authentizität und Integrität der signierten Daten verifizieren. Eine ausführlichere Beschreibung des Signature-Servers selbst ist in Abschnitt 4.4 oder [Sel99] zu finden.

Zum Betrieb eines Rechners im AutO-System sind eine Reihe von Daten nötig, die mit Hilfe des Signature-Server signiert werden müssen:

- Jeder Rechner, der in AutO integriert werden soll, benötigt ein eigenes Schlüsselpaar für den verwendeten Public-Key Algorithmus. Damit ist unter anderem der Aufbau einer sicheren Kommunikation mit anderen AutO-Rechnern möglich (Kapitel 4). Der öffentliche Schlüssel muß vom Signature-Server signiert werden.

- Jeder Rechner benötigt eine Sicherheitsklasse, die im Rahmen der sicheren Migration von Belang ist [Sel99]. Diese Sicherheitsklasse muß vom Signature-Server signiert sein, damit Objekte, denen diese Sicherheitsklasse gesendet wird, feststellen können, ob diese unverändert bei ihnen eingetroffen ist.

- Falls entsprechend konfiguriert, erlaubt AutO nur das Laden von Java-Klassen, die mit einer Signatur versehen sind. Damit ist zum einen ein gewisser Grad von Sicherheit verbunden, da nur Klassen, deren Unbedenklichkeit durch eine Untersuchung bestätigt wurde, diese Signatur erhalten. Zum anderen kann damit der Programmierer einer Klasse im Nachhinein festgestellt werden, da sein Name bei der Signierung ebenfalls gespeichert wird.

Das *SRTool* dient dazu, die Signierung bzw. Erzeugung dieser Daten zu unterstützen. Für jeden der oben aufgeführten Fälle existiert im *SRTool* ein eigener Modus, in dem Befehle zur Verfügung gestellt werden, mit denen die benötigten Daten generiert werden können. Die allgemeine Syntax für das *SRTool* lautet:

```
java [PROPERTIES] security.srtool.SRTool [OPTIONS] [GLOBAL_PARAMETERS] MODE
                              [MODE_OPTIONS] COMMAND
```

Properties

Das *SRTool* benötigt immer wieder verschiedene *Java-Keystores*. In diesen werden beliebige Schlüssel abgespeichert, unter anderem auch diejenigen, die von einem Node-Manager für sichere Kommunikation benötigt werden. Java sichert solche Keystores mit Hilfe von Paßwörtern. Das SRTool unterstützt zwei Möglichkeiten zur Eingabe dieser Paßwörter:

- Wird ein benötigtes Paßwort nicht beim Programmstart mit angegeben, wird der Benutzer an der entsprechenden Stelle im Programm gebeten, es einzugeben. Diese Methode wird empfohlen, da auf diese Weise sichergestellt ist, daß Dritte nicht in den Besitz des Paßwortes gelangen können (siehe dazu auch den nächsten Punkt).

- Das Paßwort wird mit Hilfe des entsprechenden Properties gesetzt. Properties sind in Java Umgebungsvariablen, die allen Klassen einer Java Virtual Machine zur Verfügung stehen. Auf sie kann über `System.getProperty(String)` zugegriffen werden. Beim Start eines Java-Programms besteht die Möglichkeit, solche Properties zu setzten (siehe die Syntax von `java`).

  Die einzelnen Properties zur Angabe der verschiedenen Paßwörter sind:

  `auto-keystore.password`: Das Property für das Paßwort des Keystores des lokalen Rechners. Darin müssen sich unter anderem die Schlüssel für Node-Manager und AutO-Prozeß befinden.

  `export-keystore.password`: In manchen Fällen müssen generierte Schlüssel in eigene Keystores gespeichert werden. Mit Hilfe dieses Properties kann das Paßwort für einen derartigen Export-Keystore gesetzt werden.

  `import-keystore.password`: Manche Schlüssel können nicht lokal generiert werden, etwa der öffentliche Schlüssel des Signature-Servers. Wenn sie aber trotzdem benötigt werden, müssen diese Schlüssel importiert werden. Das Paßwort für diesen Import-Keystore kann mit diesem Property gesetzt werden.

`security-representative.password`: Ein Security-Representative wird benötigt, wenn eine Verbindung mit dem Signature-Server aufgebaut werden muß. Nur er ist dazu autorisiert (Abschnitt 4.4). Mit der Property `security-representative.password` kann das Paßwort für den Keystore des Security-Representative gesetzt werden.

ACHTUNG: Man sollte sich vor Augen halten, daß Paßwörter auf diese Weise direkt beim Programmstart mitangegeben werden. Dadurch erscheinen sie *im Klartext*, wenn man sich etwa unter dem Betriebsystem Unix eine Liste der aktuell laufenden Prozesse ausgeben läßt.

Kommandozeilenoptionen

An Optionen stehen zur Verfügung:

`-v`: Wenn diese Option gewählt wird, gibt das *SRTool* nicht nur die notwendigen Statusmeldungen aus, sondern informiert den Benutzer ausführlich über gerade ausgeführte Aktionen.

`-y`: Normalerweise ergeben sich beim Betrieb des *SRTools* immer wieder Situationen, in denen der Benutzer danach gefragt wird, ob er einer Aktion zustimmt oder sie abbrechen will, etwa wenn es darum geht, eine Datei zu löschen. Mit dieser Option wird das *SRTool* dazu veranlaßt in solchen Situationen niemals nachzufragen, sondern immer von „Ja“ als Antwort auszugehen.

Globale Parameter

Globale Parameter im *SRTool* dienen dazu, Informationen für alle Modi zur Verfügung zu stellen. Mögliche Parameter sind:

`--sr-name <STRING>`: Hier kann der Name des Security-Representative angegeben werden, der das *SRTool* startet. Er wird benötigt, um bei einem Verbindungsaufbau zum Signature-Server die richtigen Schlüssel wählen zu können.

Wird der Name nicht an dieser Stelle angegeben, dann wird er aus der Datei `AutO.config` entnommen (siehe dazu auch Anhang E).

`--sr-keystore <FILE>`: Dieser Parameter dient der Angabe der Datei, in der der Keystore des Security-Representative gespeichert ist. Ist diese Datei nicht auf diese Weise spezifiziert, wird der Name aus der Datei `AutO.config` entnommen.

`--server <HOST>:<PORT>`: Spezifiziert den Rechner und den Port, an dem der Signature-Server Verbindungen akzeptiert. Wird diese Adresse nicht auf diese Weise angegeben, werden die entsprechenden Daten der Datei `AutO.config` entnommen.

Die verschiedenen Modi

Für jede der zu Beginn angeführten Aufgaben des *SRTools* steht ein eigener Modus zur Verfügung. Zusätzlich existieren noch weitere Modi, die die Arbeit mit dem Programm erleichtern.

Der Modus `help`

In diesem Modus wird ein ausführlicher Hilfetext ausgegeben, der die Aufrufsyntax des *SRTools* beschreibt sowie ausführlich auf dessen Handhabung eingeht. Es stehen keine weiteren Optionen oder Kommandos zur Verfügung.

Der Modus `key`

Das *SRTool* muß in diesem Modus gestartet werden, wenn Schüssel für den lokalen Rechner erzeugt, importiert, gelöscht oder angezeigt werden sollen. Je nach Kommando werden verschiedene Aktionen ausgeführt.

Folgende Optionen existieren:

`--keystore <FILE>`**:** Damit kann der Keystore festgelegt werden, der durch das angegebene Kommando beeinflußt werden soll. Je nach Kommando werden beispielsweise Schlüssel in ihn eingefügt oder aus ihm gelöscht.

In diesem Modus kann eine Reihe von verschiedenen Kommandos ausgeführt werden:

`create <ID>`**:** Mit diesem Kommando wird ein Schlüsselpaar für die Komponente `<ID>` in AutO erzeugt. Diese Schlüssel werden für den Aufbau sicherer Verbindungen zwischen einzelnen Teilen des AutO-System benötigt (siehe Kapitel 4).

Bei den folgenden Komponenten, für die Schlüssel erzeugt werden können, gibt `<LENGTH>` die Länge der zu erzeugenden Schlüssel an (siehe dazu auch Anhang A):

`nodemanager [ <HOST> ] <LENGTH>`**:** Erzeugt ein Schlüsselpaar für den Node-Manager des Rechners `<HOST>`, wobei der öffentliche Schlüssel vom Signature-Server zertifiziert wird. Es muß also bei der Erzeugung eine Verbindung zum Signature-Server aufgebaut werden. Deshalb kann dieses Schlüsselpaar nur von einem Security-Representative generiert werden. Sollte `<HOST>` nicht angegeben werden, wird ein Schlüsselpaar für den Node-Manager des lokalen Rechners erzeugt.

`auto-process [ <HOST> ] <LENGTH>`**:** Hiermit wird ein Schlüsselpaar für einen AutO-Prozeß des Rechners `<HOST>` erzeugt. Da alle AutO-Prozesse eines Rechners das gleiche Schlüsselpaar verwenden, muß nur ein solches Paar erzeugt werden. Der öffentliche Schlüssel wird vom lokalen Node-Manager zertifiziert. Sollte `<HOST>` nicht angegeben werden, wird ein Schlüsselpaar für den AutO-Prozeß des lokalen Rechners erzeugt.

`nameservice [ <HOST> ] <LENGTH>`: Erzeugt ein Schlüssepaar für den Name-Service des Rechners `<HOST>`, wobei der öffentliche Schlüssel vom Signature-Server zertifiziert wird. Sollte `<HOST>` nicht angegeben werden, wird ein Schlüsselpaar für den Name-Service des lokalen Rechners erzeugt.

`terminal-server [ <HOST> ':' ] <PORT> <LENGTH>` : Hiermit wird ein Schlüsselpaar für einen Terminal-Server des Rechners `<HOST>` erzeugt, der auf dem Port `<PORT>` auf Verbindungen wartet. Der öffentliche Schlüssel wird vom lokalen Node-Manager zertifiziert. Sollte `<HOST>` nicht angegeben werden, wird ein Schlüsselpaar für einen Terminal-Server des lokalen Rechners erzeugt.

`terminal-user [ <NAME> ] <LENGTH> <FILE>`: Erzeugt ein Schlüsselpaar für einen *Terminal-User* mit Namen `<NAME>`. Der generierte öffentliche Schlüssel wird vom lokalen Node-Manager zertifiziert. Die generierten Schlüssel werden in einen Keystore abgelegt, der in der mit `<FILE>` angegebenen Datei gespeichert wird. Außerdem wird der zertifizierte öffentliche Schlüssel in den Keystore des lokalen Rechners eingefügt. Damit ist dieser Terminal-User autorisiert, zu einem Terminal-Server des lokalen Rechners eine Verbindung aufzubauen.

`sda <LENGTH>` : Erzeugt ein Schlüsselpaar für die Security-Data-Administration. Der öffentliche Schlüssel wird vom Signature-Server zertifiziert.

ACHTUNG: Dieser Schlüssel darf nur auf dem Rechner erzeugt werden, auf dem dann auch die Security-Data-Administration gestartet wird.

`sda-root-admin <LENGTH> <FILE>`: Hiermit wird ein Schlüsselpaar für einen SDA-Root-Administrator erzeugt. Die Schlüssel werden in einem neuen Keystore abgelegt, der in der mit `<FILE>` angegebenen Datei abgespeichert wird. Der öffentliche Schlüssel wird von der SDA zertifiziert. Außerdem wird der zertifizierte öffentliche Schlüssel des SDA-Root-Administrators in den lokalen Keystore eingefügt, um ihm Verbindungen zum SDA-Administration-Server zu erlauben.

ACHTUNG: Dieser Schlüssel darf nur auf dem Rechner erzeugt werden, auf dem auch die Security-Data-Administration läuft.

`sda-admin <NAME> <LENGTH> <FILE>`: Hiermit wird ein Schlüsselpaar für einen SDA-Administrator namens `<NAME>` erzeugt. Die Schlüssel werden in einem neuen Keystore abgelegt, der in der mit `<FILE>` angegebenen Datei abgespeichert wird. Der öffentliche Schlüssel wird von der SDA zertifiziert. Außerdem wird der zertifizierte öffentliche Schlüssel des SDA-Administrators in den lokalen Keystore eingefügt, um ihm Verbindungen zum SDA-Administration-Server zu erlauben.

ACHTUNG: Dieser Schlüssel darf nur auf dem Rechner erzeugt werden, auf dem auch die Security-Data-Administration läuft.

`import [ <ID> ] <FILE>` : Mit diesem Kommando ist es möglich, einen zertifizierten öffentlichen Schlüssel in den lokalen Keystore zu importieren. Der Import-Keystore muß in der Datei `<FILE>` gespeichert sein. `<ID>` bedeutet:

`signature-server`: In diesem Fall wird der öffentliche Schlüssel des Signature-Servers aus dem Import-Keystore importiert.

`terminal-user [ <NAME> ]:` Importiert den zertifizierten öffentlichen Schlüssel eines Terminal-Users aus dem Import-Keystore. Wird kein Name angegeben, werden alle öffentlichen Terminal-User-Schlüssel importiert.

Wird `<ID>` nicht angegeben, werden aus dem Import-Keystore alle geeigneten öffentlichen Schlüssel übernommen.

`delete <ID>` : Alle Schlüssel der mit `<ID>` spezifizierten Komponente werden gelöscht. Mögliche Angaben für `<ID>` sind:

`nodemanager [ <HOST> ]:` Löscht alle Schlüssel des Node-Managers des Rechners `<HOST>` aus dem Keystore. Wird `<HOST>` nicht angegeben, werden die Schlüssel des lokalen Node-Managers gelöscht.

`auto-process [ <HOST> ]:` Löscht alle Schlüssel des AutO-Prozesses des Rechners `<HOST>` aus dem Keystore. Wird `<HOST>` nicht angegeben, werden die Schlüssel des lokalen AutO-Prozesses gelöscht.

`signature-server:` Löscht den öffentlichen Schlüssel des Signature-Servers aus dem Keystore.

`nameservice [ <HOST> ]:` Löscht alle Schlüssel des Name-Services des Rechners `<HOST>` aus dem Keystore. Wird `<HOST>` nicht angegeben, werden die Schlüssel eines lokalen Name-Services gelöscht.

`terminal-server [ <HOST> ':' ] <PORT> :` Löscht die Schlüssel des angegebenen Terminal-Servers aus dem Keystore. Wird `<HOST>` nicht angegeben, wird dafür der lokalen Rechner substituiert.

`terminal-user [ <NAME> ]:` Entfernt den Terminal-User mit Namen `<NAME>` aus dem Keystore. Wird kein Name angegeben, wird der Benutzername desjenigen, der das *SRTool* gestartet hat, benutzt. Damit ist dieser Terminal-User nicht mehr autorisiert, eine Verbindung zu lokalen Terminal-Servern aufzubauen.

`sda:` Löscht die Schlüssel der Security-Data-Administration aus dem Keystore.

`sda-root-admin:` Entfernt den zertifizierten öffentlichen Schlüssel des SDA-Root-Administrators aus dem Keystore. Dieser ist damit nicht mehr in der Lage, eine Verbindung zum SDA-Administration-Server aufzubauen.

`sda-admin <NAME>:` Löscht den zertifizierten öffentlichen Schlüssel des SDA-Administrators mit Namen `<NAME>` aus dem Keystore. Dieser ist damit nicht mehr in der Lage, eine Verbindung zum SDA-Administration-Server aufzubauen.

`list:` Gibt Informationen über alle Schlüssel aus, die im Keystore gespeichert sind. Es wird unter anderem angezeigt, von wem öffentliche Schlüssel zertifiziert wurden.

Der Modus `security-class`

Dieser Modus dient dazu, die Sicherheitsklasse des lokalen Rechners zu erzeugen und an den Signature-Server zu senden, damit dieser sie signieren kann. Anschließend wird sie in

einer Datei gespeichert. Diese Tätigkeit muß, da eine Verbindung zum Signature-Server etabliert wird, von einem Security-Representative ausgeführt werden. Folgende Optionen stehen für diesen Modus zur Verfügung:

`--sc-file <FILE>`**:** Hiermit kann der Name der Datei angegeben werden, in die die Sicherheitsklasse gespeichert werden soll. Damit wird der entsprechende Eintrag in der Datei `AutO.config` überschrieben.

Analog zum `key`-Modus stehen auch in diesem mehrere Kommandos zur Verfügung:

`create <INTEGER> ':' <STRING> ( ',' <STRING> )`**:** Mit diesem Kommando wird eine Sicherheitsklasse für den lokalen Rechner erzeugt. Die Sicherheitsstufe beträgt `<INTEGER>`, die Sicherheitskategorien werden nach dem Doppelpunkt, durch Kommata getrennt, angegeben. Nach dem Erzeugen der Sicherheitklasse wird eine Verbindung zum Signature-Server aufgebaut und ihm die Sicherheitsklasse gesendet. Die in der Antwort des Signature-Servers enthaltene signierte Sicherheitsklasse wird dann in der angegebenen Datei gespeichert.

`delete` : Löscht die Sicherheitsklasse des lokalen Rechners.

`list` **:** Zeigt die Sicherheitsklasse des lokalen Rechners an.

Der Modus `sign-class`

Dieser Modus dient dazu, eine oder mehrere Klassen, die im AutO-System verwendet werden sollen, aber keine Klasse des AutO-Systems sind, zu signieren (sogenannte *User-Klassen*). Dazu wird für jede zu signierende Klasse ein Hashwert berechnet, dieser Hashwert an den Signature-Server gesendet, der ihn signiert, und die zurückerhaltene Signatur in einer Datei gespeichert. Folgende Optionen stehen für diesen Modus zur Verfügung:

`--i`**:** Führt die Signierung der einzelnen Klassen interaktiv durch, d. h. der Benutzer wird bei jeder Klasse gefragt, ob sie signiert werden soll oder nicht.

`--r`**:** Mit dieser Option wird das *SRTool* dazu veranlaßt, Pakete rekursiv zu durchsuchen. Es werden also nicht nur die Klassen des angegebenen Pakets signiert, sondern auch Klassen in Unterpaketen, sofern solche existieren.

`--path`**:** Gibt das Verzeichnis an, in dem die zu signierenden Klassen liegen. Hiermit wird der Eintrag der Datei `AutO.config` überschrieben, der das Verzeichnis der User-Klassen angibt.

Für das Kommando ist folgende Syntax einzuhalten:

`<PROGRAMER> <CLASS(ES)>`

`<PROGRAMER>` spezifiziert den Programmierer der Klassen, die signiert werden sollen. Die Angabe der Klassen `<CLASS(ES)>` folgt normalen Java-Konventionen. Ausgehend von einem Klassenpfad, der entweder mit `--path` oder in der Datei `AutO.config` angegeben wird, werden Klassen mit vollständigem Paketnamen angegeben. Um die Klassen eines kompletten Paket zu signieren, kann das Zeichen `*` verwendet werden. Beispiele sind:

- `sign-class Goofy user.test.TestKlasse`
- `sign-class --r Goofy user.test.*`

Der Modus `list`

In diesem Modus werden alle Informationen ausgegeben, die auch einzeln in den anderen Modi mit dem Kommando `list`, sofern vorhanden, angezeigt werden. Es stehen keine weiteren Optionen oder Kommandos zur Verfügung.

D.2 Der *CryptographyMatrixGenerator*

Für jedes autonome Objekt existiert eine Kryptographie-Matrix, in der für jeden Guard, der in dem autonomen Objekt deklariert wurde, folgende Werte festgelegt sind (siehe auch Abschnitt 4.5.4):

- der Chiffrenstatus der Nachricht, die den Guard ausgelöst hat (nur bei On-Guards),
- der Chiffrenstatus der Antwortnachricht, die nach der Abarbeitung des Guards zurückgesendet wird (nur bei On-Guards),
- der Chiffrenstatus von Nachrichten, die in einer Guard-Aktion versendet werden, und
- der Chiffrenstatus von Antwortnachrichten, die in einer Guard-Aktion empfangen werden.

Beim Compilieren einer `<CLASS>.auto`-Datei, in der ein autonomes Objekt deklariert wird, mit dem *AutOJavaCompiler* werden automatisch zwei Dateien generiert, die für die Kryptographie-Matrix wichtig sind:

`<CLASS>CryptoMatrix.java`**:** Diese Java-Klasse enthält die Kryptographie-Matrix, die standardmäßig beim Compilieren eines autonomen Objekts für dieses generiert wird. In ihr werden alle Chiffrenstati mit dem Default-Chiffrenstatus initialisiert, d. h. alle ankommenden Nachrichten müssen verschlüsselt und mit einem MAC versehen sein, alle ausgehenden Nachrichten werden verschlüsselt und mit einem MAC versehen.

`<CLASS>.crypto`**:** Diese Datei enthält Informationen über alle deklarierten Guards des autonomen Objekts. Sie wird vom *CryptographyMatrixGenerator* benötigt, um eine individuelle Kryptographie-Matrix-Klasse zu erzeugen. Im folgenden wird sie auch als *Kryptographie-Matrix-Sourcedatei* bezeichnet.

Das Programm *CryptographyMatrixGenerator* dient nun dazu, mit Hilfe der Datei `<CLASS>.crypto` eine individuelle Version der Kryptographie-Matrix zu erzeugen und in der Java-Klasse `<CLASS>CryptoMatrix.java` zu speichern. In ihr sind dann im allgemeinen nicht mehr alle Werte auf wahr gesetzt.

Starten des *CryptographyMatrixGenerator*

Durch das Ausführen der Klasse `compiler.cryptomatrixgenerator.CMGeneratorGUI` wird der *CryptographyMatrixGenerator* gestartet. Die Aufrufsyntax lautet:

```
java compiler.cryptomatrixgenerator.CMGeneratorGUI [OPTIONS] (<CLASS>.crypto)*
```

Folgende Kommandozeilenparameter stehen zur Verfügung:

`--combo-box`**:** Die Auswahl der Chiffrenstati erfolgt mit Hilfe einer ComboBox. Ein Beispiel zeigt Abbildung D.1.

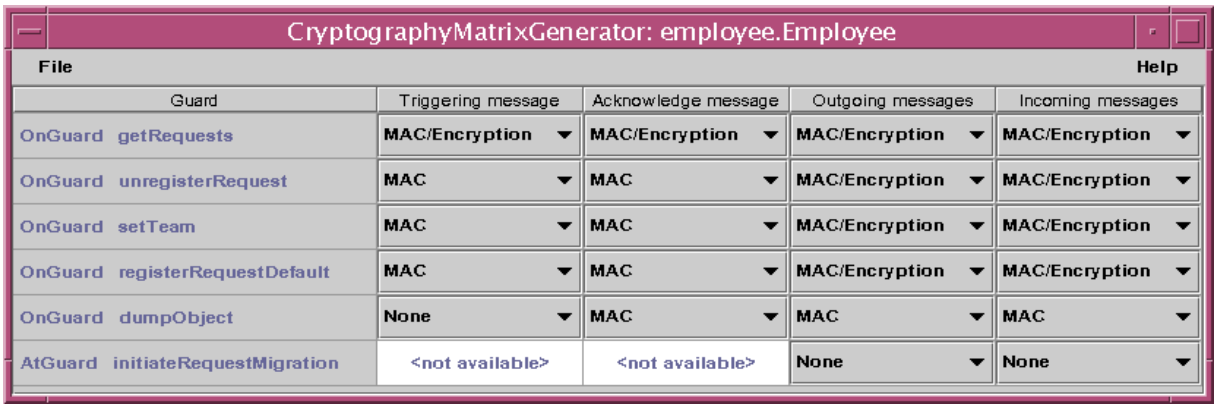


Abbildung D.1: Der *CryptographyMatrixGenerator* mit einer ComboBox-Auswahl

`--check-box`**:** Die Auswahl der Chiffrenstati erfolgt mit Hilfe einer CheckBox. Ein Beispiel zeigt Abbildung D.2.

Wird beim Start eine Kryptographie-Matrix-Sourcedatei angegeben, dann werden die darin enthaltenen Werte in einem eigenen Fenster angezeigt (siehe Abbildung D.2). Je nach gesetzten Kommandozeilenoptionen geschieht dies mit Hilfe der Combo- oder CheckBox-Darstellung.

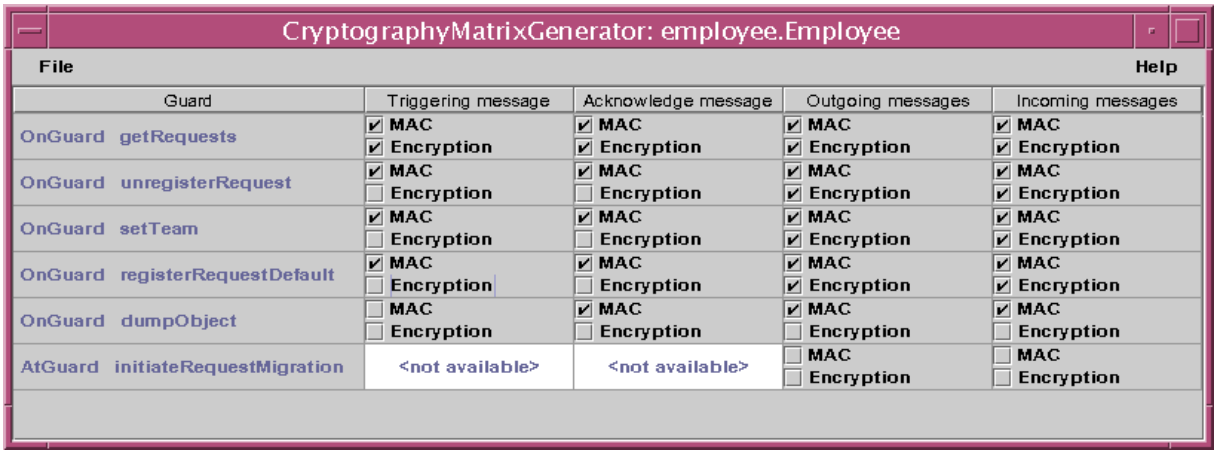


Abbildung D.2: Der *CryptographyMatrixGenerator* mit einer CheckBox-Auswahl

Das Setzen von Chiffrenstati

Für die Anzeige der Kryptographie-Matrix gilt:

- Jede Zeile stellt die Chiffrenstati eines einzelnen Guards dar, d. h. ob eingehende Nachrichten verschlüsselt bzw. mit einem MAC versehen sein müssen, und ob ausgehende Nachrichten verschlüsselt bzw. mit einem MAC versehen werden sollen.

- In der Spalte, die mit **Triggering Message** bezeichnet ist, wird definiert, ob Nachrichten, die einen Guard auslösen, verschlüsselt bzw. mit MAC versehen sein müssen.

- Die Spalte **Acknowledge Message** definiert, ob die Nachricht, die nach der Abarbeitung eines Guards als Antwort gesendet wird, verschlüsselt bzw. mit MAC versehen werden soll.

- Die **Outgoing Messages**-Spalte zeigt, ob Nachrichten, die bei der Abarbeitung eines Guards gesendet werden, verschlüsselt bzw. mit MAC versehen werden soll.

- Als letztes zeigt die Spalte **Incoming Messages**, ob Nachrichten, die bei der Abarbeitung eines Guards als Antwort auf gesendete Nachrichten empfangen werden, verschlüsselt bzw. mit MAC versehen sein müssen.

Ein Benutzer kann die einzelnen Werte für jeden Guard so setzen, wie es für das entsprechende autonome Objekt passend ist.

Das Menü des *CryptographyMatrixGenerator*

Das File-Menü: Dieses Menü enthält verschiedene allgemeine Punkte.

Load: Hiermit kann man eine Kryptographie-Matrix-Sourcedatei laden. Nach der Auswahl dieses Menüpunkts erscheint ein Dateiauswahldialog, in dem die gewünschte Sourcedatei ausgewählt werden kann.

Save: Wird dieser Menüpunkt ausgewählt, wird die angezeigte Kryptographie-Matrix im Format einer Java-Klasse abgespeichert. Der Dateiname lautet `<CLASS>CryptoMatrix.java`, wobei `<CLASS>` den Namen der Klasse des autonomen Objekts angibt, für das die Kryptographie-Matrix erstellt wurde. Die Datei wird im selben Verzeichnis gespeichert, in dem auch die Sourcedatei liegt.

Exit CMG: Mit der Auswahl dieses Menüpunkts wird der *CryptographyMatrixGenerator* beendet.

Das Help-Menü: In diesem Menüpunkt sind alle Hilfefunktionen des *CryptographyMatrixGenerator* enthalten.

About CMG: Dieser Menüpunkt zeigt ein Fenster, in dem verschiedene Informationen über den *CryptographyMatrixGenerator*, etwa seine Versionsnummer, dargestellt werden.

Content: Öffnet ein Fenster, in dem eine ausführliche Online-Hilfe zur Bedienung des *CryptographyMatrixGenerator* angezeigt wird.

D.3 Die *Security-Data-Administration*

Die Security-Data-Administration ist ein wesentlicher Bestandteil des Autorisierungssystems von AutO. Sie wird in Kapitel 5 ausführlich beschrieben. Gestartet wird die SDA mit der Klasse `security.rbac.sda.SecurityDataAdministration`. Es stehen folgende Kommandozeilenoptionen zur Verfügung:

`-h` **oder** `--help`**:** Zeigt einen kurzen Hilfetext, der die Kommandozeilenparameter der Security-Data-Administration erklärt.

`-k <FILE>` **oder** `--keystore <FILE>`**:** Legt die Datei fest, die den Keystore enthält, der von der SDA genutzt wird, um ihr eigenes Schlüsselpaar sowie die zertifizierten öffentlichen Schlüssel des SDA-Root-Administrators und der SDA-Administrators zu laden.

`-rf <FILE>` **oder** `--rbac-file <FILE>`**:** Spezifiziert eine Datei, die die normalen RBAC-Mengen enthält, die die SDA nach dem Start lädt. Der Inhalt der Datei wird mit Hilfe des `RBACFileParser` geparst.

`-af <FILE>` **oder** `--admin-rbac-file <FILE>`**:** Gibt eine Datei an, die administrative RBAC-Mengen enthält. Die Datei wird nach dem Start der SDA mit Hilfe der Klasse `AdminRBACFileParser` geparst.

D.4 Das *SDA Administration Tool*

Die Security-Data-Administration enthält Daten, die auch während der Laufzeit des Systems modifiziert werden können. Deshalb beinhaltet die SDA einen SDA-Administration-Server. Das *SDA Administration Tool* ist der Client, der die Verbindung zum SDA-Administration-Server herstellt.

Das Programm kann in zwei verschiedenen Modi betrieben werden:

Root Mode: In diesem Modus muß das *SDA Administration Tool* vom SDA-Root-Administrator gestartet werden. Nur er ist autorisiert, die administrativen RBAC-Mengen der SDA zu modifizieren. Authentifiziert wird dieser Administrator durch sein Schlüsselpaar, das mit dem *SRTool* generiert wurde. Baut er eine Verbindung zur SDA auf, kann diese anhand des Zertifikats, das bei diesem Verbindungsaufbau mitgesendet wird, die Identität des SDA-Root-Administrator verifizieren.

Common Mode: Dieser Modus wird von den normalen SDA-Administrators benutzt, um eine Verbindung zum SDA-Administration-Server aufzubauen. Sie sind in der Lage, die normalen RBAC-Mengen zu modifizieren, unterliegen dabei allerdings den Beschränkungen der administrativen RBAC-Mengen.

Ein SDA-Administrator muß in der Menge der administrativen Benutzer enthalten sein. Außerdem kann er nur Aktionen ausführen, die ihm durch seine zugeordneten administrativen Rollen erlaubt sind.

Gestartet wird das *SDA Administration Tool* mit dem Befehl:

```
java security.rbac.sda.SDAAdminTool [OPTIONS]
```

Kommandozeilenoptionen

Das *SDA Administration Tool* stellt eine Reihe von Kommandozeilenoptionen zur Verfügung, die den Ablauf des Programms beeinflussen:

`-v` **oder** `--verbose`**:** Mit Angabe dieser Option wird das *SDA Administration Tool* veranlaßt, ausführliche Statusmeldungen auszugeben.

`-h` **oder** `--help`**:** Gibt einen kurzen Text aus, der beschreibt, wie das Programm gestartet wird und welche Optionen zur Verfügung stehen.

`-k <FILE>` **oder** `--keystore <FILE>`**:** Hiermit ist der Benutzer in der Lage, die Datei anzugeben, die den Keystore enthält, in dem die Schlüssel liegen, die für einen Verbindungsaufbau zum SDA-Administration-Server benötigt werden.

Wird die Datei nicht auf diese Weise angegeben, wird zuerst im lokalen, dann im Homeverzeichnis des gegenwärtigen Benutzers nach einer Datei namens „`.SDA-Administrator-KeyStore`“ (im **Common Mode**) oder

„`.SDA-RootAdministrator-KeyStore`“ (im **Root Mode**) gesucht. Findet das Programm keine entsprechende Datei, beendet es sich mit einer Fehlermeldung.

`-s <HOST>` **oder** `--server <HOST>`**:** Gibt den Rechner an, auf dem der SDA-Administration-Server ausgeführt wird. Wird der Rechner hier nicht angegeben, wird der entsprechende Eintrag in der Datei `AutO.config` benutzt.

`-p <INTEGER>` **oder** `--port <INTEGER>`**:** Spezifiziert den Port des SDA-Administration-Servers. Wird der Port nicht auf diese Weise angegeben, wird der entsprechende Eintrag in der Datei `AutO.config` benutzt.

`-n <STRING>` **oder** `--name <STRING>`**:** Hiermit kann der Benutzer seinen Namen angeben. Dieser wird benötigt, um im Keystore, der vom *SDA Administration Tool* benutzt wird, die richtigen SDA-Administrator-Schlüssel zu finden. Wird kein Name angegeben, wird die Kennung des Benutzers als Name des SDA-Administrator verwendet.

`-r` **oder** `--root`**:** Startet das Programm im **Root Mode**. In diesem Fall muß der Keystore für dieses Programm die Schlüssel des SDA-Root-Administrators enthalten.

Die Bedienung des *SDA Administration Tools*

Nach dem Start des Programms wird zuerst eine Verbindung zum SDA-Administration-Server aufgebaut. Sollte dies fehlschlagen, beendet sich das Programm mit einer Fehlermeldung. Konnte die Verbindung etabliert werden, erstellt der SDA-Administration-Server eine Kopie der RBAC-Mengen, die der Client modifizieren will, also eine Kopie der administrativen RBAC-Mengen, falls ein SDA-Root-Administrator die Verbindung aufgebaut hat, oder eine Kopie der normalen RBAC-Mengen, wenn die Verbindung von einem SDA-Administrator hergestellt wurde. Von den im folgenden vorgestellten Kommandos wird nur diese Kopie modifiziert, dadurch ist die SDA nicht von den Änderungen betroffen. Erst wenn der Benutzer mit dem Kommando `commit` die Änderungen festschreibt, werden sie an die SDA weitergegeben.

Kommandos

Dem Benutzer stehen eine Reihen von Kommandos zur Verfügung, mit denen er das Programm bedienen und die RBAC-Mengen der SDA modifizieren kann:

`help`**:** Gibt einen Hilfetext aus, der die Bedienung des *SDA Administration Tools* beschreibt. Dieser Text enthält auch eine Erläuterung aller zur Verfügung stehenden Kommandos.

`quit`**:** Beendet das *SDA Administration Tool*. Die Verbindung zum SDA-Administration-Server wird beendet und damit alle Modifikationen verworfen, die noch nicht mit `commit` festgeschrieben wurden.

`show`: Zeigt die aktuellen RBAC-Mengen. Sollten an diesen bereits Änderungen durchgeführt worden sein, sind diese bei den angezeigten Mengen eingeschlossen. Es werden also nicht diejenigen RBAC-Mengen angezeigt, die die SDA aktuell besitzt, sondern eine möglicherweise bereits geänderte Kopie von diesen.

`save <FILE>`: Speichert die aktuellen (und möglicherweise modifizierten) RBAC-Mengen, in der Datei, die mit `<FILE>` angegeben wurde.

`open <FILE>`: Ersetzt die aktuellen RBAC-Mengen durch diejenigen, die in der angegebenen Datei enthalten sind.

`abort`: Verwirft alle durchgeführten Änderungen und lädt eine neue Kopie der RBAC-Mengen von der Security-Data-Administration.

`commit`: Propagiert die geänderten RBAC-Mengen an die SDA. Nachdem Rechner, die von der Änderung betroffene User-Informationen gepuffert haben, benachrichtigt wurden und diese nun veralteten User-Informationen aus ihrem Puffer entfernt haben, werden die entsprechenden RBAC-Mengen der SDA durch die geänderten Mengen ersetzt.

Bestätigen nicht alle Rechner mit veralteten User-Informationen die Benachrichtigung, scheitert die Änderung der RBAC-Mengen. Die geänderte Version kann nicht an die SDA propagiert werden (siehe dazu auch Abschnitt 5.4.4).

Während die bisherigen Kommandos hauptsächlich zur Steuerung des *SDA Administration Tools* dienen und damit nicht die RBAC-Mengen beeinflussen, führen die nun folgenden Kommandos gezielt Änderungen an diesen Mengen durch:

`clear`: Löscht alle RBAC-Mengen, d. h. es werden alle Elemente aus den RBAC-Mengen entfernt.

`add user <STRING>`: Fügt einen neuen Benutzer mit Namen `<STRING>` zu der aktuellen RBAC-Menge **Users** hinzu. Sollte der Benutzer bereits in dieser Menge enthalten sein, wird eine Fehlermeldung ausgegeben.

`remove user <STRING>`: Löscht den Benutzer mit Namen `<STRING>`. Der Benutzer wird aus der aktuellen **Users**-Menge entfernt. Außerdem werden alle Elemente der Relation **UR** entfernt, die diesen Benutzer mit einer Rolle verbinden.

`assign role` $\texttt{<STRING}_{\texttt{role}}\texttt{>}$ `to user` $\texttt{<STRING}_{\texttt{user}}\texttt{>}$: Ordnet die Rolle $\texttt{<STRING}_{\texttt{role}}\texttt{>}$ dem Benutzer mit Namen $\texttt{<STRING}_{\texttt{user}}\texttt{>}$ zu. Das entsprechende Tupel wird in die Relation **UR** eingefügt. Sowohl die angegebene Rolle als auch der Benutzer müssen bekannt sein, d. h. in den entsprechenden RBAC-Mengen enthalten sein.

`revoke role` $\texttt{<STRING}_{\texttt{role}}\texttt{>}$ `from user` $\texttt{<STRING}_{\texttt{user}}\texttt{>}$: Entfernt die Zuordnung des Benutzers $\texttt{<STRING}_{\texttt{user}}\texttt{>}$ zur Rolle $\texttt{<STRING}_{\texttt{role}}\texttt{>}$ aus der Relation **UR**.

`add role <STRING`$_{\texttt{role}}$`> [ extends <STRING`$_{\texttt{baserole}}$`> ]:` Fügt die Rolle `<STRING`$_{\texttt{role}}$`>` zur Menge **Roles** hinzu. Wird nach dem `extends` Schlüsselwort eine Basisrolle `<STRING`$_{\texttt{baserole}}$`>` angegeben, ist diese neue Rolle von dieser Basisrolle abgeleitet. Sollte keine Basisrolle angegeben werden, wird automatisch `RBACRole` als Basisrolle eingesetzt. Damit sind automatisch alle Rollen von `RBACRole` abgeleitet.

Wird eine unbekannte, also nicht in der Menge **Roles** enthaltene, Basisrolle angegeben, scheitert das Hinzufügen der neuen Rolle.

`remove role <STRING>:` Entfernt die Rolle `<STRING>` aus der Menge der Rollen **Roles**. Es werden auch alle entsprechenden Elemente aus der Relation **RP** entfernt, die dieser Rolle Permissions zuordnen. Ebenso werden alle Zuordnungen aus der Relation **UR** gelöscht, die einem Benutzer diese Rolle zuweisen.

`assign permission <STRING`$_{\texttt{permission}}$`> to role <STRING`$_{\texttt{role}}$`>:` Ordnet die Permission mit dem Bezeichner `<STRING`$_{\texttt{permission}}$`>` aus der Menge **Permissions** der Rolle `<STRING`$_{\texttt{role}}$`>` zu. Sowohl die angegebene Rolle als auch die Permission müssen existieren, d. h. in den entsprechenden Mengen enthalten sein.

Ein Bezeichner identifiziert eindeutig eine einzelne Permission. Bezeichner werden verwendet, da die Angabe der eigentlichen Permission an dieser Stelle zu umständlich wäre.

`revoke permission <STRING`$_{\texttt{permission}}$`> from role <STRING`$_{\texttt{role}}$`>:` Entfernt die Zuordnung der Permission mit dem Bezeichner `<STRING`$_{\texttt{permission}}$`>` zur Rolle `<STRING`$_{\texttt{role}}$`>`. Das entsprechende Element wird aus der Relation **RP** entfernt.

Die Rechte eines normalen SDA-Administrators werden durch die administrativen RBAC-Mengen festgelegt. Wie ein normaler Benutzer des AutO-Systems kann auch ein SDA-Administrator Rollen, in seinem Fall also administrative Rollen, aktivieren und deaktivieren. Je nach Menge der aktiven Rollen sind ihm bestimmte Aktionen erlaubt. Da ein SDA-Root-Administrator keinen solchen administrativen RBAC-Mengen unterworfen ist, stehen bestimmte Befehle nur im **Common Mode**, in dem ein normaler SDA-Administrator das *SDA Administration Tool* startet, zur Verfügung:

`activate role <STRING>:` Aktiviert die administrative Rolle mit Namen `<STRING>`. Der SDA-Administrator muß der Rolle zugeordnet sein, ansonsten kann er die gewünschte Rolle nicht aktivieren.

`deactivate role <STRING>:` Deaktiviert die Rolle `<STRING>`. Sie wird damit aus der Menge der aktiven Rollen entfernt.

`add permission <STRING> : <PERMISSION>:` Erzeugt eine neue Permission, die in die Menge **Permissions** eingefügt wird. Ab diesem Zeitpunkt kann diese neue Permission Rollen zugeordnet werden.

`<STRING>` stellt einen Bezeichner dar, der dieser Permission eindeutig zugeordnet wird. Sollte bereits eine Permission mit diesem Bezeichner existieren, scheitert diese

Aktion. Der Bezeichner wird benötigt, um beim Zuordnen von Permissions zu Rollen oder beim Löschen von Permissions diese eindeutig angeben zu können.

<PERMISSION> definiert eine gültige Permission des AutO-Systems. Sie werden in Abschnitt 5.4.3 genauer vorgestellt. Die Syntax von <PERMISSION> wird weiter unten beschrieben.

`remove permission <STRING>`: Entfernt die Permission mit dem Bezeichner <STRING> aus der RBAC-Menge **Permissions**. Sollte die Permission noch Rollen zugeordnet sein, so werden die entsprechenden Zuordnungen aus der Relation **RP** entfernt.

Permissions

Für jede Aktion, die ein Benutzer bzw. ein im Auftrag des Benutzers handelndes Objekt, z. B. eine Transaktion, ausführen kann, existiert eine Permission, mit der man die Ausführung der Aktion erlauben oder verbieten kann (siehe Abschnitt 5.4.3).

Mit Hilfe des Befehls `add permission <STRING> : <PERMISSION>` kann ein SDA-Administrator neue Permissions erzeugen. Jede Permission erhält einen eindeutigen Bezeichner, um sie später leichter referenzieren zu können. Die in Abschnitt 5.4.3 vorgestellten Permissions können auf folgende Weise angegeben werden:

`( '+' | '-' ) start-transaction <CLASS>`: Hiermit kann eine Permission spezifiziert werden, die das Starten einer Transaktion, implementiert in der Klasse <CLASS>, erlaubt oder verbietet.

Eine positive Permission wird mit '+' angegeben, eine negative entsprechend mit '-'.

`( '+' | '-' ) create-active-object <CLASS>`: Definiert eine Permission, die das Erzeugen eines autonomen Objekts der Klasse <CLASS> erlaubt oder verbietet.

`( '+' | '-' ) access-active-object <CLASS> [ <OID> ]`: Spezifiziert eine Permission, die den Zugriff auf alle Guards eines autonomen Objekts erlaubt oder verbietet. Wird keine OID mit Hilfe von <OID> angegeben, wird eine Klassenpermission erzeugt. Damit erlaubt oder verbietet diese Permission den Zugriff auf alle Guards aller Instanzen der Klasse <CLASS>. Wird hingegen mit Hilfe von <OID> ein einzelnes Objekt spezifiziert, wird eine Instanzpermission erzeugt, die den Zugriff auf alle Guards des autonomen Objekts mit der angegebenen OID erlaubt oder verbietet.

`( '+' | '-' ) access-guard <CLASS> [ <OID> ] <STRING>`: Definiert den Zugriff auf den Guard mit Namen <STRING> eines autonomen Objekts. Wird keine OID angegeben, wird eine Klassenpermission erzeugt, wodurch die Permission den Zugriff auf den genannten Guard für alle Instanzen der Klasse <CLASS> regelt. Anderenfalls erlaubt oder verbietet die Permission nur den Zugriff auf den Guard des mit Hilfe von <OID> angegebenen autonomen Objekts.

Administrative Permissions

Ein SDA-Root-Administrator ordnet unter anderem administrativen Rollen administrative Permissions zu. Diese administrativen Permissions regeln die Aktionen, die normale SDA-Administrators ausführen dürfen.

Die Menge der administrativen Permissions kann im Gegensatz zur Menge der normalen Permissions nicht modifiziert werden. Es existieren nur die vordefinierten administrativen Permissions.

Im folgenden werden die einzelnen administrativen Permissions, die auch in Abschnitt 5.4.1 vorgestellt werden, zusammen mit ihren eindeutigen Bezeichnern angegeben:

`add-user`: Diese Permission erlaubt es einem SDA-Administrator, neue Benutzer zur Menge **Users** hinzuzufügen.

`remove-user`: Mit dieser Permission ist ein SDA-Administrator in der Lage, Benutzer aus der Menge **Users** zu entfernen.

`assign-role-to-user`: Hiermit ist es einem SDA-Administrator erlaubt, Benutzern Rollen zuzuordnen.

`revoke-role-from-user`: Diese Permission gewährt das Recht, die Zuordnung eines Benutzers zu einer Rolle zu entfernen.

`add-role`: Ein SDA-Administrator erhält hiermit das Recht, neue Rollen in die Menge der Rollen **Roles** einzufügen.

`remove-role`: Diese Permission erlaubt es einem SDA-Administrator, Rollen zu löschen.

`assign-permission-to-role`: Hiermit besitzt ein SDA-Administrator das Recht, Permissions Rollen zuzuordnen.

`revoke-permission-from-role`: Diese Permission gewährt das Recht, die Zuordnung einer Permission zu einer Rolle zu entfernen.

`add-permission`: Ein SDA-Administrator erhält hiermit das Recht, neue Permissions in die Menge **Permissions** einzufügen.

`remove-permission`: Diese administrative Permission erlaubt es einem SDA-Administrator Permissions zu löschen.

Administrative Permissions sind niemals negativ. Darf ein SDA-Administrator eine bestimmte Aktion nicht ausführen, dann darf er keine Rolle besitzen, der die entsprechende Permission zugeordnet ist.

Anhang E

Einstellungen in der Datei `AutO.config`

Das AutO-System bietet eine Vielzahl von Möglichkeiten, es an spezielle Gegebenheiten anzupassen. Die gewünschten Einstellungen können in der Datei `AutO.config` durchgeführt werden [Gri97, Isl97, Sel99]. Diese wird beim Start von AutO gelesen. Die darin enthaltenen Werte werden dann über die Klasse `common.Configuration` den AutO-Klassen zur Verfügung gestellt.

Im Rahmen dieser Diplomarbeit wurden verschiedene Komponenten zum bestehenden AutO-System hinzugefügt. Auch sie können über die Datei `AutO.config` konfiguriert werden.

Sektion `[NodeManager]`

In dieser Sektion werden verschiedene Einstellungen getroffen, die die Arbeitsweise des Node-Managers beeinflussen.

ACHTUNG: Die folgenden Einstellungen beziehen sich nur auf Systemnachrichten, nicht auf Benutzernachrichten. Diese werden auf andere Art konfiguriert (siehe Abschnitt 4.5.1).

`messages.system.nodemanager.incoming.mac [ on | off ]:` Hiermit wird festgelegt, ob Systemnachrichten, die einem Node-Manager von einem anderen zugestellt werden, mit einem MAC versehen sein müssen. Dieser wird etwa benötigt, um die Integrität und Authentizität der empfangenen Daten zu verifizieren.

Ist dieser Schalter auf `on` gesetzt, gibt der Node-Manager eine Fehlermeldung aus, wenn Nachrichten ohne MAC empfangen werden.

Default-Einstellung: `on`

`messages.system.nodemanager.incoming.encryption [ on | off ]:` Dieser Schalter legt fest, ob Systemnachrichten, die einem Node-Manager von einem anderen zugestellt werden, verschlüsselt sein müssen.

Ist dieser Schalter auf `on` gesetzt, gibt der Node-Manager eine Fehlermeldung aus, wenn unverschlüsselte Nachrichten empfangen werden.

Default-Einstellung: `on`

`messages.system.nodemanager.outgoing.mac [ on | off ]:` Hiermit wird festgelegt, ob Systemnachrichten, die von einem Node-Manager an einen anderen gesendet werden, mit einem MAC versehen werden sollen.

Default-Einstellung: `on`

`messages.system.nodemanager.outgoing.encryption [ on | off ]:` Dieser Schalter legt fest, ob Systemnachrichten, die von einem Node-Manager an einen anderen gesendet werden, verschlüsselt werden sollen.

Default-Einstellung: `on`

Sektion `[NameService]`

In dieser Sektion werden verschiedene Einstellungen getroffen, die die Arbeitsweise des Name-Service beeinflussen.

`messages.nodemanager.incoming.mac [ on | off ]:` Hiermit wird festgelegt, ob Nachrichten, die dem Name-Service von einem Node-Manager zugestellt werden, mit einem MAC versehen sein müssen.

Ist dieser Schalter auf `on` gesetzt, gibt der Name-Service eine Fehlermeldung aus, wenn Nachrichten ohne MAC empfangen werden.

Default-Einstellung: `on`

`messages.nodemanager.incoming.encryption [ on | off ]:` Dieser Schalter legt fest, ob Nachrichten, die dem Name-Service von einem Node-Manager zugestellt werden, verschlüsselt sein müssen.

Ist dieser Schalter auf `on` gesetzt, gibt der Name-Service eine Fehlermeldung aus, wenn unverschlüsselte Nachrichten empfangen werden.

Default-Einstellung: `on`

`messages.nodemanager.outgoing.mac [ on | off ]:` Hiermit wird festgelegt, ob Nachrichten, die vom Name-Service an einen Node-Manager gesendet werden, mit einem MAC versehen sein müssen.

Default-Einstellung: `on`

`messages.nodemanager.outgoing.encryption [ on | off ]:` Dieser Schalter legt fest, ob Nachrichten, die vom Name-Service an einen Node-Manager gesendet werden, verschlüsselt sein müssen.

Default-Einstellung: `on`

`shore-link.port <INTEGER>:` Wird der *AutOShore-Server* [Gri97] als DBMS für das AutO-System verwendet, greift dieser in bestimmten Fällen auf den Name-Service zu, um von diesem benötigte Daten über den Aufenthaltsort von autonomen Objekten zu erhalten. Der Port, auf dem der Name-Service auf solche Anfragen reagiert, wird mit diesem Wert festgelegt.

Default-Einstellung: `20020`

Sektion `[Security]`

Diese Sektion dient zur Konfiguration des gesamten Sicherheitsystems von AutO. Zusätzlich existieren noch spezifische Sektionen wie `Cryptography` und `Authorization`, in denen speziell für diese Bereiche zuständige Werte und Schalter gesetzt werden können. Die Sektion `Security` enthält hingegen allgemeine Schalter, die auf alle Teilkomponenten des Sicherheitssystems Auswirkungen haben.

`security [ on | off ]:` Schaltet das gesamte Sicherheitssystem von AutO aus oder ein. Es wird dringend empfohlen, die Sicherheit einzuschalten. Ansonsten können Angriffe gegen ein laufendes System nicht erkannt und abgewehrt werden.

Default-Einstellung: `on`

Sektion `[SignatureServer]`

Diese Sektion dient zur Konfiguration des Signature-Servers. Weiterhin können auch Programme und Komponenten konfiguriert werden, die mit dem Signature-Server in Beziehung stehen, also etwa das *SRTool* (siehe Anhang D) oder das *SATool* [Sel99].

`signatureserver-host <STRING>:` Spezifiziert die Adresse des Rechners, auf dem der Signature-Server ausgeführt wird. Dieser Eintrag wird benötigt, wenn mit dem *SRTool* eine Verbindung zum Signature-Server aufgebaut werden soll, um etwa Klassen zu signieren.

`signatureserver-port <INTEGER>:` Gibt den Port an, auf dem der Signature-Server Verbindungen entgegennimmt.

Default-Einstellung: `30001`

`security-representative-name <STRING>:` Hiermit kann der Name des Security-Representatives gesetzt werden, der etwa bei der Benutzung des *SRTools* benötigt wird.

`security-representative-keystore <STRING>:` Spezifiziert den Namen der Datei, in dem der Keystore für den Security-Representative abgelegt ist.

Default-Einstellung: `{home}/.Security-Representative-Keystore`[1]

Sektion `[Cryptography]`

In dieser Sektion werden verschiedene Algorithmen und Provider für die verwendeten Kryptographiealgorithmen festgelegt. Diese werden unter anderem für den Aufbau sicherer Verbindungen zwischen den verschiedenen Rechnern des AutO-Systems benötigt.

Die Namen der einzelnen Kryptographiealgorithmen müssen der Dokumentation des verwendeten Security Providers, entnommen werden. Verschiedene Provider werden in Anhang B vorgestellt.

`cryptography [ on | off ]:` Wenn dieser Wert auf `on` gesetzt wird, werden kryptographische Methoden genutzt, um das AutO-System vor Angriffen zu schützen. AutO-Sockets nützen in diesem Fall Kryptographie, um Nachrichten auf sichere Art zu übertragen. Wird diese Option auf `off` gesetzt, ist keine Nutzung von AutO-Sockets möglich.

Default-Einstellung: `on`

`message_digest_algorithm <STRING>:` Legt den Algorithmus fest, der zur Berechnung von kryptographischen Hashwerten, auch *Message Digests* genannt, verwendet wird. Er wird von AutO-Sockets sowie bei der Signierung von Klassen mit Hilfe des *SRTools* benötigt.

Default-Einstellung: `SHA`

`message_digest_algorithm_provider <STRING>:` Definiert den Provider, der den Algorithmus zur Berechnung von *Message Digests* zur Verfügung stellt. Wird kein solcher angegeben, wird der erste bekannte Provider verwendet, der den gewünschten Algorithmus zur Verfügung stellt.

`mac_algorithm <STRING>:` Legt den Algorithmus fest, der zur Berechnung von MAC-Werten verwendet wird.

Default-Einstellung: `HMAC/SHA`

`mac_algorithm_provider <STRING>:` Spezifiziert den Provider, der den Algorithmus zur Berechnung von MAC-Werten zur Verfügung stellt. Wird kein solcher angegeben, wird der erste bekannte Provider verwendet, der den gewünschten Algorithmus zur Verfügung stellt.

`cipher_algorithm <STRING>`: Legt den symmetrischen Algorithmus fest, der zur Verschlüsselung von Daten in AutO verwendet wird.

Default-Einstellung: `DES/CBC/PKCS5Padding`

[1] Die Variable `{home}` enthält das Home-Verzeichnis des Benutzers, der den aktuellen Prozeß gestartet hat.

`cipher_algorithm_provider <STRING>`: Definiert den Provider, der den mit `cipher_algorithm` angegebenen Algorithmus zur Verfügung stellt. Wird kein Provider angegeben, wird der erste bekannte Provider verwendet, der den gewünschten Algorithmus zur Verfügung stellt.

`cipher_algorithm_strength <INTEGER>`: Legt die Stärke des Schlüssels fest, der bei der symmetrischen Verschlüsselung in AutO verwendet wird. Genaueres zur Stärke von Schlüsseln kann in [Sun98, Sun97b] nachgelesen werden.

`public_key_algorithm <STRING>`: Legt den Algorithmus fest, der für asymmetrische Verschlüsselung benutzt wird. Dieser wird beim Schlüsselaustausch beim Verbindungsaufbau mit AutO-Sockets benötigt.

Default-Einstellung: `RSA`

`public_key_algorithm_provider <STRING>`: Definiert den Provider, der den mit `public_key_algorithm` angegebenen Algorithmus zur Verfügung stellt. Wird kein solcher angegeben, wird der erste bekannte Provider verwendet, der den gewünschten Algorithmus zur Verfügung stellt.

`public_key_algorithm_strength <INTEGER>`: Legt die Stärke der Schlüssel fest, die bei den asymmetrischen Algorithmen verwendet werden.

`auto-keystore <STRING>`: Gibt den Namen der Datei an, die den Keystore für den lokalen Rechner enthält. In diesem müssen alle Schlüssel gespeichert sein, die ein AutO-Node benötigt, also etwa Schlüssel für Node-Manager, AutO-Prozeß, usw.

Default-Einstellung: `{AUTO_HOME}/.autohost_keystore`[2]

`cache_check_interval <TIME>`: Gibt das Zeitintervall an, das zwischen zwei Überprüfungen des Session-Key-Caches der lokalen AutO-Sockets liegt. Bei einer Überprüfung werden diejenigen gepufferten Session-Keys als ungültig markiert, die veraltet sind. Erkennt ein AutO-Socket, daß der verwendete Session-Key veraltet ist, generiert er einen neuen und trägt diesen in den Cache ein.

Das Format von `<TIME>` kann in der Online-Dokumentation zur Klasse `common.Configuration` nachgelesen werden.

Default-Einstellung: `1d`

`max_session_key_age <TIME>`: Legt das maximale Alter eines Session-Keys fest. Überschreitet ein derartiger Schlüssel dieses Alter, ist er ungültig und ein neuer Session-Key muß erzeugt werden.

Default-Einstellung: `7d`

`auto-socket.usage [ on | off ]`: Schaltet die Benutzung von AutO-Sockets zur sicheren Kommunikation in AutO ein oder aus. Ist die Benutzung von AutO-Sockets deaktiviert, werden normale Java-Sockets zur Kommunikation benützt. Sicherheit vor Angriffen Dritter ist in diesem Fall nicht gewährleistet.

[2] Die Variable `{AUTO_HOME}` enthält das Verzeichnis, in dem AutO installiert wurde.

Default-Einstellung: `on`

Sektion `[Authorization]`

In dieser Sektion kann das Autorisierungssystem von AutO konfiguriert werden.

`authorization [ on | off ]:` Schaltet das Autorisierungssystem von AutO ein oder aus. Sollte es deaktiviert werden, hat keine der folgenden Schalter Einfluß auf das AutO-System.

Default-Einstellung: `on`

`sda.host <STRING>:` Spezifiziert die Adresse des Rechners, auf dem die Security-Data-Administration ausgeführt wird. Diese ist für den Betrieb des RBAC-Autorisierungssystems unbedingt notwendig.

`sda.revoke-timeout <TIME>:` Setzt das Timeout-Intervall für die Benachrichtigung von Rechnern des AutO-Systems, wenn sich in der Security-Data-Administration Änderungen ergeben haben und deshalb lokale Informationen aus den Puffern entfernt werden müssen (siehe dazu auch Abschnitt 5.4.4).

Default-Einstellung: `1h`

`sda.messages.incoming.mac [ on | off ]:` Hiermit wird festgelegt, ob Nachrichten, die der Security-Data-Administration von einem Node-Manager zugestellt werden, mit einem MAC versehen sein müssen.

Ist dieser Schalter auf `on` gesetzt, gibt die Security-Data-Administration eine Fehlermeldung aus, wenn Nachrichten ohne MAC empfangen werden.

Default-Einstellung: `on`

`sda.messages.incoming.encryption [ on | off ]:` Dieser Schalter legt fest, ob Nachrichten, die der Security-Data-Administration von einem Node-Manager zugestellt werden, verschlüsselt sein müssen.

Ist dieser Schalter auf `on` gesetzt, gibt die Security-Data-Administration eine Fehlermeldung aus, wenn unverschlüsselte Nachrichten empfangen werden.

Default-Einstellung: `on`

`sda.messages.outgoing.mac [ on | off ]:` Hiermit wird festgelegt, ob Nachrichten, die die Security-Data-Administration an einen Node-Manager sendet, mit einem MAC versehen werden sollen.

Default-Einstellung: `on`

`sda.messages.outgoing.encryption [ on | off ]:` Dieser Schalter legt fest, ob Nachrichten, die die Security-Data-Administration an einen Node-Manager sendet, verschlüsselt werden sollen.

Default-Einstellung: `on`

`sda.administration.port <INTEGER>`: Definiert den Port, auf dem der SDA-Administration-Server Verbindungen akzeptiert. Dieser Server läuft auf demselben Rechner, auf dem auch die Security-Data-Administration ausgeführt wird. Das *SDA Administration Tool* benötigt diesen Port, um mit dem SDA-Administration-Server kommunizieren zu können.

Default-Einstellung: `22010`

`sda.node-link.port <INTEGER>`: Definiert den Port, auf dem der NodeLink der Security-Data-Administration Verbindungen entgegennimmt. Node-Manager des AutO-Systems nutzen diese Verbindungen, um Daten des RBAC-Systems anzufordern.

Default-Einstellung: `22009`

`sda.node-link.connection-cache-size <INTEGER>`: Legt die maximale Zahl von Verbindungen fest, die ein NodeLink in seinem LRUCache puffert.

Default-Einstellung: `200`

`sda-link.port <INTEGER>`: Definiert den Port, auf dem der SDALink eines Node-Managers Verbindungen der Security-Data-Administration entgegen nimmt.

Default-Einstellung: `22008`

`sda-link.retries <INTEGER>`: Legt die Anzahl von Versuchen fest, eine Verbindung zur Security-Data-Administration aufzubauen. Scheiterte der Verbindungsaufbau so oft, wie mit diesem Wert festgelegt wurde, wird eine Fehlermeldung ausgegeben.

Default-Einstellung: `5`

`sda-link.wait-time <TIME>`: Setzt das Zeitintervall, das gewartet wird, wenn ein Verbindungsaufbau zur Security-Data-Administration scheitert, bevor ein neuer Versuch unternommen wird.

Default-Einstellung: `10s`

`user-cache.size <INTEGER>`: Gibt die Größe des Benutzer-Caches eines Node-Managers an, d. h. die maximale Anzahl von Benutzer-Informationen, die in diesem gepuffert werden können.

Default-Einstellung: `100`

`role-cache.size <INTEGER>`: Gibt die Größe des Rollen-Caches eines Node-Managers an, d. h. die maximale Anzahl von Rollen-Informationen, die in diesem gepuffert werden können.

Default-Einstellung: `100`

`role-authorizer.timeout <TIME>`: Definiert die maximale Zeitspanne, die der RoleAuthorizer eines AutO-Prozesses wartet, wenn er eine Benutzer-Information beim lokalen Benutzer-Cache angefordert hat. Trifft innerhalb dieser Zeitspanne keine Antwort ein, wird ein Fehler gemeldet.

Default-Einstellung: `10min`

`permission-authorizer.timeout <TIME>`: Definiert die maximale Zeitspanne, die der PermissionAuthorizer eines AutO-Prozesses wartet, wenn er eine Rollen-Information vom lokalen Rollen-Cache angefordert hat. Trifft innerhalb dieser Zeitspanne keine Antwort ein, wird ein Fehler gemeldet.

Default-Einstellung: `10min`

Sektion `[Terminal]`

In dieser Sektion kann das *Terminal*, ein spezieller interaktiver AutO-Prozeß, der in Abschnitt 2.2 beschrieben wird, konfiguriert werden.

`server.messages.incoming.mac [ on | off ]`: Hiermit wird festgelegt, ob Nachrichten, die einem Terminal-Server von einem Terminal-Client zugestellt werden, mit einem MAC versehen sein müssen.

Ist dieser Schalter auf `on` gesetzt, gibt der Terminal-Server eine Fehlermeldung aus, wenn Nachrichten ohne MAC empfangen werden.

Default-Einstellung: `on`

`server.messages.incoming.encryption [ on | off ]`: Dieser Schalter legt fest, ob Nachrichten, die einem Terminal-Server von einem Terminal-Client zugestellt werden, verschlüsselt sein müssen.

Ist dieser Schalter auf `on` gesetzt, gibt der Terminal-Server eine Fehlermeldung aus, wenn unverschlüsselte Nachrichten empfangen werden.

Default-Einstellung: `on`

`server.messages.outgoing.mac [ on | off ]`: Hiermit wird festgelegt, ob Nachrichten, die von einem Terminal-Server an einen Terminal-Client gesendet werden, mit einem MAC versehen sein müssen.

Default-Einstellung: `on`

`server.messages.outgoing.encryption [ on | off ]`: Dieser Schalter legt fest, ob Nachrichten, die von einem Terminal-Server an einen Terminal-Client zugestellt werden, verschlüsselt sein müssen.

Default-Einstellung: `on`

Literaturverzeichnis

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

[CDF 94] Carey, M.; DeWitt, D.; Franklin, M.; Hall, N.; McAuliffe, M.; Naughton, J.; Schuh, D.; Solomon, M.; Tan, C.; Tsatalos, O.; White, S.; Zwilling, M.: Shoring Up Persistent Applications. **In:** *Proc. of the ACM SIGMOD Conf. on Management of Data*. Minneapolis, MI, USA, Mai 1994, S. 383–394

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

[Cry] Cryptix-JCE. `http://www.systemics.com/docs/cryptix`

[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

[DR] Daemen, J.; Rijmen, V. The Block Cipher SQUARE. `http://www.esat.kuleuven.ac.be/%7Erijmen/downloadable/square/fse.pdf`

[DTH97] Demurjian, S. A.; Ting, T. C.; Hu, M.-Y.: Security for Object-Oriented Databases, Systems, and Applications. **In:** Prater, J. (Hrsg.): *Progress in Object-Oriented Databases*, Ablex, 1997

[DTPH97] Demurjian, S. A.; Ting, T. C.; Price, M.; Hu, M.-Y.: Extensible and Reusable Role-Based Object-Oriented Security. **In:** Samarati, P. (Hrsg.); Sandhu, R. S. (Hrsg.): *Database Security, X: Status and Prospects*, Chapman Hall, 1997

[FGK95] Ferraiolo, David F.; Gugini, Janet A.; Kuhn, C. R. Role-Based Access Control (RBAC): Features and Motivations. Computer Security Applications Conference. 1995

[FK92] Ferraiolo, David F.; Kuhn, Richard: Role-Based Access Control. Erschienen in: *15th NIST-NCSC National Computer Security Conference* (1992), Okt, S. 554–563

[Fra96] FRANKLIN, M.: *Client Data Caching: A Foundation*. Kluwer Academic Press, 1996

[GHJV95] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.: *Design Patterns – Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1995

[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

[Kal92] KALISKI, Burton S. The MD2 Message-Digest Algorithm. `http://rfc.fh-koeln.de/rfc/html/rfc1319.html`. April 1992

[KBC97] KRAWCZYK, H.; BELLARE, M.; CANETTI, R. HMAC: Keyed-Hashing for Message Authentication. `ftp://ftp.informatik.uni-hamburg.de/pub/doc/rfc/rfc-2100-2199/rfc2104.txt.gz`. Februar 1997

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

[KIK98] KRIVOKAPIĆ, N.; ISLINGER, M.; KEMPER, A.: Migrating Autonomous Objects in a WAN Environment / Universität Passau. 94030 Passau, Germany, 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

[Kor83] KORTH, H. F.: Locking Primitives in a database system. Erschienen in: *Journal of the ACM* 30 (1983), Januar, Nr. 1, S. 55–79

[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

[KS91] KORTH, H. F.; SILBERSCHATZ, A.: *Database System Concepts*. Singapur: McGraw–Hill, 1991. – ISBN 0–07–100804–7

[MD94] MOHAMMED, Imtiaz; DILTS, David M.: Design for dynamic user-role-based security. Erschienen in: *Computers & Security* 13 (1994), S. 661–671

[NIS95] NIST CSL BULLETIN ON RBAC. An Introduction to Role-based Access Control. `http://www.nist.gov/itl/lab/bulletns/rbac`. Dez 1995

[Pas97] PASSON, Hajo: Auf Nummber sicher. Erschienen in: *iX* 10 (1997), S. 160

[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

[Riv92] RIVEST, Ronald L. The MD5 Message-Digest Algorithm. `http://www.roxen.com/rfc/rfc1321.html`. April 1992

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

[SCFY96] SANDHU, Ravi S.; COYNE, Edward J.; FEINSTEIN, Hal L.; YOUMAN, Charles E.: Role-Based Access Control Models. Erschienen in: *IEEE Computer* 29 (1996), Nr. 2, S. 38–47

[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

[Sel99] SELTZSAM, S.: *Migration und Sicherheit in AutO, einem verteilten System autonomer Objekte*. D-94030 Passau, Universität Passau, Diplomarbeit, Januar 1999

[SM94] SOLMS, S. H.; MERWE, Isak van d.: The management of computer security profiles using a role-oriented approach. Erschienen in: *Computers & Security* 13 (1994), S. 673–680

[SS84] SCHWARZ, P. M.; SPECTOR, A. Z.: Synchronizing Shared Abstract Types. Erschienen in: *ACM Trans. Computer Systems* 2 (1984), Nr. 3, S. 223–250

[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

[Sun98] Sun Microsystems. `http://java.sun.com/security/JCE1.2/early-access/index.html`: *JCE API Specification Version 1.2 (Draft)*. 1998

[TS94] THOMAS, Roshan K.; SANDHU, Ravi S.: Conceptual Foundations for a Model of Task-based Authorizations. Erschienen in: *IEEE Computer Security Foundations Workshop* 7 (1994), Juni, S. 66–79

[U.S85] U.S. DEPARTMENT OF DEFENSE. Trusted Computer Security Evaluation Criteria. DOD 5200.28-STD. 1985

[U.S93a] U.S. Department of Commerce. `http://www.itl.nist.gov/div897/pubs/fip46-2.htm`: *FIPS PUB 46-2: Data Encryption Standard*. Dezember 1993

[U.S93b] U.S. Department of Commerce. `http://www.itl.nist.gov/div897/pubs/fip180-1.html`: *NIST FIPS PUB 180-1: Secure Hash Standard*. Mai 1993

[U.S94] U.S. Department of Commerce. `http://www.itl.nist.gov/div897/pubs/fip186.htm`: *FIPS PUB 186: Digital Signature Standard*. Mai 1994

[Zim95] ZIMMERMANN, Philip: *The Official PGP User's Guide*. Boston: MIT Press, 1995

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 .....

Markus Keidl