Sicherheit in einem Java-basierten verteilten System autonomer Objekte*

Markus Keidl, Stefan Seltzsam, Alfons Kemper und Natalija Krivokapić

Universität Passau
Lehrstuhl für Informatik
94030 Passau, Germany
<*Nachname*>`@db.fmi.uni-passau.de`

Kurzfassung Diese Arbeit beschreibt die Integration verschiedener Sicherheitsmechanismen in ein persistentes, verteiltes System autonomer Objekte. Dieses System wurde für den Einsatz in einer heterogenen WAN-Umgebung konzipiert und deshalb vollständig in Java implementiert, um Plattformunabhängigkeit zu erreichen. Um den Zugriff der Benutzer auf die Objekte im System kontrollieren zu können, wird ein rollenbasiertes Autorisierungsverfahren integriert, das sowohl skalierbar als auch leicht administrierbar ist. Da das System Migration von Objekten unterstützt, müssen sowohl Objekte vor unsicheren Rechnern, als auch Rechner vor feindlichen Objekten geschützt werden. Ersteres erreicht man durch eine geeignete Kategorisierung der Rechner und Objekte, letzteres durch Anwendung des in Java integrierten Sandkasten-Sicherheitsmodells.

1 Einleitung

Die zunehmende weltweite Vernetzung von Computersystemen verändert die Anforderungen an verteilte Datenbanken. Diese werden in Zukunft Hunderte von Sites unterstützen müssen, an denen eine Vielzahl von Benutzern arbeiten werden. Verbunden werden die einzelnen Sites durch ein Wide-Area-Network (WAN), z. B. dem Internet, und nicht, wie bisher üblich, durch ein Local-Area-Network (LAN) [18, 19]. Dadurch können Verbindungen zwischen den einzelnen Sites nicht nur unterschiedliche Geschwindigkeiten aufweisen, sondern möglicherweise auch weniger zuverlässig und sicher sein. Solche Systeme verlangen ein hohes Maß an Skalierbarkeit bezüglich Leistung, Verfügbarkeit und Administration.

In derartigen verteilten Datenbanken sollten dezentrale Kooperationskonzepte zum Einsatz kommen, da sie die Skalierbarkeit des Systems im Gegensatz zu zentralen Kontrollinstanzen unterstützen. Sie können unter anderem durch einen hohen Grad an Autonomie der Objekte oder Sites, auf denen die Objekte liegen, realisiert werden. Autonome Objekte, welche auf natürliche Weise um Kooperationskonzepte, wie z. B. [15, 16, 17], erweitert werden können, bieten sich an.

* Diese Arbeit wurde gefördert von der German Israel Foundation (GIF)

Im Rahmen von AutO, einem persistenten, verteilten System autonomer Objekte, wurde ein derartiges Modell [11] realisiert. AutO ist skalierbar, plattformunabhängig, da vollständig in Java implementiert, und darauf ausgelegt, innerhalb eines WANs zu arbeiten und damit beispielsweise in einem weltweit operierenden Unternehmen eingesetzt zu werden. Migration wird unterstützt, um die Kommunikationskosten, die die Leistungsfähigkeit eines weit verteilten Systems maßgeblich beeinflussen, zu reduzieren. Dies ist vor allem im Hinblick auf die zu erwartenden Lastveränderungen in einem derartigen System von Bedeutung.

Ein ebenfalls für WANs konzipiertes System ist *Mariposa* [24], das einen mikroökonomischen Ansatz verfolgt und Autonomie auf Site-Ebene bietet. Der Schwerpunkt liegt dabei auf Optimierung der Anfrageverarbeitung und auf Replikation. *JavaSpaces* [26] ist eine Plattform zur Entwicklung von verteilten Anwendungen im Internet/Intranet, die verteilte Persistenz unterstützt, jedoch weniger mächtig ist als AutO. Sie stellt einen Mechanismus zum Austausch von Java-Objekten zwischen verschiedenen *Spaces* zur Verfügung. Objekte können in einen *Space* geschrieben, daraus gelesen oder entfernt werden. JavaSpaces bietet einen Benachrichtigungsmechanismus, mit dem Objekte informiert werden können, wenn ein Objekt, das einen bestimmten Typ oder Inhalt hat, in einen *Space* geschrieben wird.

Ein System wie AutO erfordert effiziente Sicherheitsmechanismen, um unautorisierte Manipulationen durch Benutzer zu verhindern. Bei der Migration von Objekten müssen nicht nur Rechner vor möglicherweise gefährlichen Objekten, sondern auch Objekte vor unsicheren Rechnern geschützt werden.[1]

In dieser Arbeit werden dafür geeignete Sicherheitskomponenten für verteilte Objektbanksysteme am Beispiel von AutO vorgestellt. Dabei werden keine neuen Sicherheitskonzepte entwickelt, sondern aus der Vielzahl existierender Konzepte diejenigen gewählt, die in AutO sinnvoll integriert werden können. Insbesondere werden die von Java zur Verfügung gestellten Konzepte analysiert und eingesetzt. Obwohl die folgenden Kapitel teilweise sehr AutO-bezogen sind, können die vorgestellten Methoden einfach auf andere verteilte Objektsysteme übertragen werden.

Der Rest der Arbeit ist wie folgt organisiert. In Kapitel 2 wird das Modell autonomer Objekte kurz beschrieben und AutO vorgestellt. Kapitel 3 gibt einen Überblick über die in Java vorhandenen Sicherheitskonzepte, die in dieser Arbeit eingesetzt wurden. Das Autorisierungsmodell wird in Kapitel 4 vorgestellt. Kapitel 5 beschreibt den Schutz von Objekten vor unsicheren Sites, Kapitel 6 den Schutz der Sites vor Angriffen durch Objekte. Kapitel 7 faßt den Inhalt dieser Arbeit nochmals kurz zusammen.

[1] Es wird davon ausgegangen, daß die restlichen Komponenten des Systems sicher sind. Kommunikationsverbindungen müssen etwa, wie es in AutO durch den Einsatz von kryptographischen Verfahren der Fall ist, vor Angriffen Dritter gesichert werden.

2 Einführung in AutO

AutO ist ein Java-basiertes, persistentes und verteiltes System autonomer Objekte, das auf dem in [11] vorgestellten Modell basiert. Objekte empfangen Nachrichten[2] und reagieren auf diese. Liegen mehrere Nachrichten vor, werden diese gepuffert. Das Objekt entscheidet nach einem vorgegebenen Algorithmus, der die Prioritäten der Nachrichten berücksichtigt, über die Reihenfolge der Bearbeitung. Der folgende Abschnitt soll einen kurzen Einblick in dieses System geben.

2.1 Systemarchitektur

AutO wurde im Hinblick auf den Einsatz in einem Wide-Area-Network (WAN) konzipiert. Die Struktur eines solchen Netzwerkes ist selten homogen, deshalb wurde bei der Entwicklung auf Plattformunabhängigkeit Wert gelegt, und AutO vollständig in Java [1] implementiert.

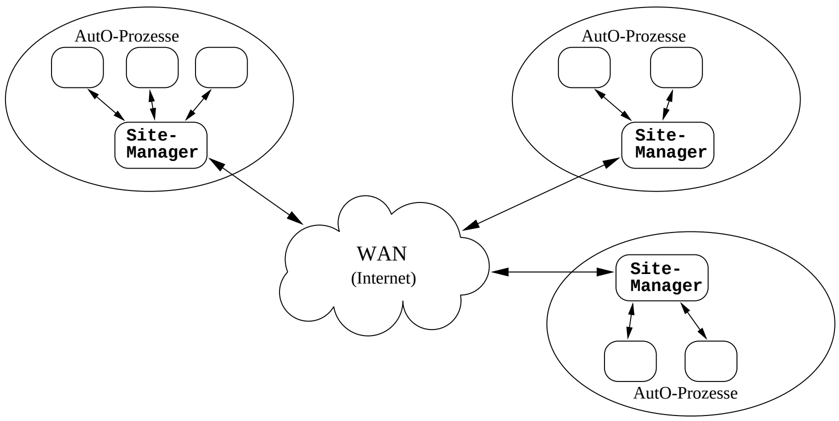

Abbildung 1. Die Architektur des AutO-Systems

Abbildung 1 zeigt vereinfacht den Aufbau eines AutO-Systems, bestehend aus mehreren vernetzten Rechnern. Jeder daran beteiligte Rechner, im folgenden auch *(AutO-)Site* genannt, zeichnet sich durch zwei Arten von Prozessen aus, welche auf ihm ausgeführt werden:

– je einem *Sitemanager*, dessen Hauptaufgabe die Verbindung lokaler AutO-Komponenten mit Teilsystemen auf anderen AutO-Sites ist. Er leitet etwa Nachrichten von lokalen Objekten an andere Sitemanager weiter oder nimmt

[2] Nachrichten in AutO sind Instanzen einer speziellen Klasse `Message`. Sie werden zum Versenden über das Netzwerk serialisiert.

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.

Zur Kommunikation über das verbindende Netzwerk werden Java-Sockets verwendet, die das Standardprotokoll TCP/IP benutzen.

2.2 Autonome Objekte

Autonome Objekte besitzen eine Nachrichten-Queue, in der eintreffende Nachrichten verwaltet werden. Ein autonomes Objekt, das mit einem eigenen Thread versehen ist, um so unabhängig vom Restsystem agieren zu können, wählt sich nach einem festgelegten Algorithmus eine dieser Nachrichten aus und führt dann die damit verbundene Operation aus. Dabei können auch Nachrichten an andere autonome Objekte versendet werden.

Eine Transaktion besteht in AutO aus einer Folge von Zugriffen auf autonome Objekte und möglicherweise Berechnungen, die zwischen diesen durchgeführt werden. Transaktionen greifen auf autonome Objekte zu, indem sie ihnen Nachrichten senden, die die Ausführung der gewünschten Operation veranlassen. Neben der Kommunikation mit Objekten, können Transaktionen auch Objekte erzeugen. Das Transaktionskonzept von AutO [13] erfüllt die ACID-Eigenschaften und basiert auf semantischem Locking [12, 23].

Persistenz und Recovery sorgen dafür, daß AutO nach einem Systemabsturz in einem transaktionskonsistenten Zustand weiterarbeiten kann. Zu diesem Zweck kommt ein beliebiges kommerzielles Datenbanksystem, das eine JDBC-Schnittstelle [8] bereitstellt, als Objektspeicher zum Einsatz. Zusätzlich wurde eine Anbindung an SHORE [2] implementiert, das alternativ zur Speicherung von Objekten eingesetzt werden kann. In einem AutO-System können gleichzeitig verschiedene Datenbanksysteme eingesetzt werden. Jede AutO-Site kann eine eigene Datenbank verwenden, mehrere Sites können sich aber auch eine solche teilen. Aus Effizienzgründen sollte jedoch mindestens eine Datenbank pro LAN existieren.

Zur dynamischen Optimierung des Systems – genauer, zur Reduzierung von Kommunikationskosten – wird in AutO *Migration* unterstützt. Autonome Objekte bestimmen selbständig, ob und wohin sie migrieren. Dazu werden Statistiken über das Zugriffsverhalten der AutO-Sites erstellt, von denen Nachrichten empfangen werden. Eine für jedes Objekt individuell festlegbare Strategie, die auch zur Laufzeit austauschbar ist, bestimmt dann mit Hilfe dieser Statistiken das Migrationsziel. Ausführlicher beschreibt dies [14].

3 Sicherheitsmechanismen von Java

Die Programmiersprache Java wird unter anderem als Internet-Sprache propagiert. Der Grund dafür sind vor allem die bekannten Applets, die von einem

Browser aus dem Internet geladen und dann ausgeführt werden können. Dieser Vorgang ist nicht ohne Risiko für den Rechner, auf dem das Applet ausgeführt wird, da es ohne spezielle Vorkehrungen sehr einfach wäre, ein Applet in Java zu implementieren, das z. B. unerlaubt Daten von der Festplatte des Anwenders liest oder Viren in das System einschleust. Wegen dieser Sicherheitsrisiken wurde von den Entwicklern von Java ein Sicherheitsmodell fest in der Sprache verankert[3], das sogenannte Sandkastenmodell [20]. Dieses Modell besteht aus fünf Schichten (siehe Abbildung 2), die zusammen den Sandkasten bilden, der einen geschützten Bereich darstellt. Ausbruchsversuche aus diesem Bereich werden von keiner der fünf Schichten, die jede für spezielle Aufgaben zuständig ist, toleriert:

Abbildung 2. Die fünf Schichten des Sandkastenmodells.

Compiler: Der Compiler ist für die grundsätzliche Einhaltung der Sprachsemantik von Java zuständig. Er stellt sicher, daß die Zugriffsrechte (z.B. `public` oder `private`) von Klassen, Methoden und Variablen strikt eingehalten werden, daß Programme keinen Zugriff auf uninitialisierte Speicherzellen erlangen und daß der `final`-Modifier[4] bei Klassen, Methoden und Variablen beachtet wird. Außerdem verhindert der Compiler ungültige Typkonvertierungen (Casts), soweit diese statisch feststellbar sind.
Da der vom Compiler erzeugte Code nachträglich modifiziert werden oder von einem manipulierten Compiler stammen kann, müssen die vom Compiler vorgenommenen Prüfungen in den anderen Schichten des Modells nochmals vorgenommen werden.

Class Loader: Der Class Loader ist für das Laden von Klassen in die Java Virtual Machine (JVM) zuständig. Jeder Class Loader definiert dabei seinen eigenen Namensraum. Dadurch ist es möglich, verschiedene Komponenten einer Anwendung durch verschiedene Class Loader in die JVM laden zu lassen, ohne daß eine Komponente Zugriff auf die Klassen der jeweils anderen Komponenten hat.

Class-/Bytecode Verifier: Aufgabe des Verifiers ist es, das Format der Klassendatei zu überprüfen. Da der Java-Code von einem nicht vertrauenswürdigen Compiler stammen könnte, werden einige Prüfungen, die schon der Compiler vorgenommen hat, an dieser Stelle nochmals durchgeführt. Außerdem wird sichergestellt, daß keine Über- oder Unterläufe im Operandenstack vor-

[3] In diesem Artikel wird die Version 1.2 von Java zugrunde gelegt.

[4] Der `final`-Modifier kennzeichnet in Java Klassen, Methoden und Variablen, die nicht mehr veränderbar sind. Eine `final`-Klasse kann z. B. nicht mehr abgeleitet werden.

kommen können. Diese Überprüfungen sind nötig, da es einem Programm sonst möglich wäre, auf fremde Speicherbereiche zuzugreifen.

Security Manager: Der Security Manager legt die Grenzen des Sandkastens fest, in dem die Anwendungen laufen sollen. Dazu wird der Security Manager vor jedem Zugriff auf lokale Ressourcen des Rechners (z. B. Dateisystem und Netzwerk) befragt, ob diese Aktion erlaubt ist. Ist das nicht der Fall, verhindert der Security Manager die Aktion.

Interpreter/JIT: Die Laufzeitumgebung von Java übernimmt die letzten nötigen Überprüfungen. Sie überprüft Typkonvertierungen, deren Gültigkeit statisch nicht feststellbar war. Außerdem überwacht sie, daß Arrayzugriffe stets innerhalb der Arraygrenzen stattfinden. Dies ist nötig, um Anwendungen am Zugriff auf fremde oder uninitialisierte Speicherbereiche zu hindern.

Dieses Sicherheitsmodell gestattet es, Objekte in einer Laufzeitumgebung mit eingeschränkten Privilegien ablaufen zu lassen und so unerwünschte, u. U. sicherheitsgefährdende Aktionen zu verhindern.

Darüberhinaus bietet Java dem Programmierer einige kryptographische Hilfsmittel an, wie z. B. Algorithmen für digitale Fingerabdrücke, digitale Signaturen und diverse Verschlüsselungsverfahren. Diese sind als *Java Cryptography Extension* (JCE) [25] standardisiert und sollen den Entwickler dabei unterstützen, eigene sichere Anwendungen zu entwickeln. Der Benutzer von Java ist dabei nicht auf die Implementierung der benötigten Klassen von SUN angewiesen, sondern kann sich beliebige *Security Provider* installieren, die die benötigte Funktionalität bereitstellen. Da der Export des von SUN angebotenen Security Providers aus den USA nicht erlaubt ist, wurde für diese Arbeit auf IAIK [9] zurückgegriffen, das alle benötigten Algorithmen bietet.

In AutO werden kryptographische Verfahren unter anderem verwendet, um eine sichere Verbindungen zwischen den verschiedenen Komponenten des Systems zu ermöglichen. Diese Verbindungen erlauben es, jede einzelne Nachricht zu signieren und/oder zu verschlüsseln, um den jeweiligen Anforderungen möglichst flexibel und ressourcensparend gerecht werden zu können.

4 Rollenbasierte Autorisierung

Sicherheit in Datenbanksystemen [7, 10, 21] beinhaltet unter anderem den Schutz von gespeicherten Informationen vor unautorisiertem Zugriff. *Autorisierung* ist dabei die Umsetzung einer gewählten Sicherheitsstrategie in Regeln, die genau festlegen, wer auf welche Informationen in welcher Art und Weise zugreifen darf. Ein Zugriff ist autorisiert, wenn er sich mit diesen *Zugriffsregeln* verträgt. Stellt man sicher, daß auf Daten nur auf autorisiertem Wege zugegriffen werden kann, bezeichnet man dies als *Zugriffskontrolle* [4].

4.1 Zugriffskontrollmodelle

In den letzten Jahrzehnten wurden viele Modelle für die Zugriffskontrolle in Datenbanksystemen entworfen. Die meisten lassen sich grundsätzlich in zwei

Gruppen einordnen [28]: *Mandatory Access Control (MAC)* und *Discretionary Access Control (DAC)*.

In Umgebungen, in denen Daten entsprechend ihrer Geheimhaltungsstufe klassifiziert werden, wird oft ein MAC-Modell verwendet. Jedem Objekt, wie etwa einer Datei oder einem Verzeichnis, wird eine Klassifikationsstufe zugeordnet. Ebenso erhält jedes Subjekt – z. B. Personen oder Prozesse – eine Freigabestufe. Zugriffsregeln legen dann fest, in welcher Relation Klassifikation und Freigabe stehen müssen, damit ein Zugriff autorisiert ist.

Bei der Discretionary Access Control wird für jedes Subjekt/Objekt-Paar die Art des Zugriffs angegeben, für die das Subjekt autorisiert ist. Außerdem erlaubt DAC dem Besitzer eines Objekts, im Allgemeinen dessen Erzeuger, anderen Subjekten darauf Zugriff zu gewähren.

In den letzten Jahren hat sich gezeigt, daß diese beiden Modelle vor allem in objektorientierten Systemen mehrere Nachteile besitzen [5]. Verstärktes Interesse gilt deshalb der *rollenbasierten Zugriffskontrolle*, auch *Role-based Access Control (RBAC)* genannt, die gerade bei objektorientierten Datenbanksystemen mehrere Vorteile bietet. Man muß etwa nicht, wie bei DAC, für jedes Subjekt/Objekt-Paar Zugriffsmöglichkeiten spezifizieren. In einem flexibel skalierbaren System wie AutO würde dies durch die schiere Menge von Paaren zu administrativen Problemen führen.

4.2 RBAC in AutO

Ein formales Modell für rollenbasierte Zugriffskontrolle wird in [22] vorgestellt. Drei Grundmengen werden unterschieden:

- Die Menge **User** repräsentiert die erlaubten Benutzer eines Systems, etwa die Mitarbeiter eines Unternehmens.
- Die verschiedenen Aufgaben und Funktionen der Benutzer werden durch die Menge **Rollen** dargestellt.
- In der Menge **Rechte** werden alle möglichen Zugriffsrechte definiert.

Zwischen den drei Mengen existieren Beziehungen wie sie in Abbildung 3 dargestellt werden. Die Relation $\mathbf{UR} \subseteq \mathbf{User} \times \mathbf{Rollen}$ definiert, welche Rollen einem Benutzer zugeordnet sind. Analog definiert die Relation $\mathbf{RR} \subseteq \mathbf{Rollen} \times \mathbf{Rechte}$, welche Rechte einer Rolle zugeordnet sind.

Abbildung 3 zeigt noch zwei weitere Mengen: **Sessions** und **Constraints**. Eine *Session* repräsentiert einen im System aktiven (eingeloggten) Benutzer. Jede Session kann eindeutig einem Benutzer zugeordnet werden. Umgekehrt ist die Zahl der zur gleichen Zeit aktiven Sessions eines Benutzers nicht begrenzt. In einer Session kann ein Benutzer eine beliebige Zahl von Rollen aktivieren und deaktivieren, sofern er ihnen gemäß der UR-Relation zugeordnet ist. Dies ist nötig, wenn etwa Einschränkungen verhindern, daß ein Benutzer gleichzeitig zwei bestimmte Rollen verwendet. Es kann ihm auch helfen, Fehler zu vermeiden,

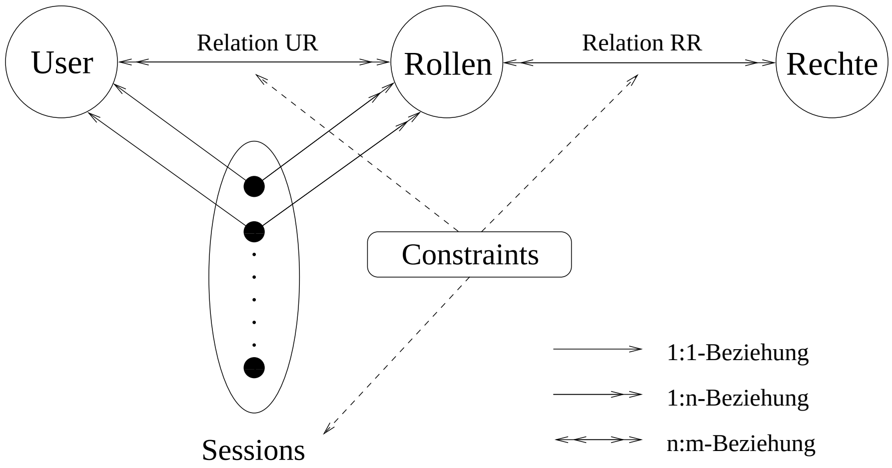

Abbildung 3. Ein Modell für rollenbasierte Zugriffkontrolle [22]

indem er nur diejenigen Rollen aktiviert, die er für die Erledigung seiner Aufgabe benötigt[5].

Einschränkungen, definiert in der Menge **Constraints**, ermöglichen es, die Zuordnung von Benutzern zu Rollen gewissen Randbedingungen zu unterwerfen, indem z. B. ein Benutzer nicht gleichzeitig zwei bestimmten Rollen zugeordnet sein darf oder aber die Zahl der Benutzer einer Rolle beschränkt ist. Analog existieren auch Einschränkungen, die auf Rollen und Rechte einwirken.

AutO ist ein verteiltes System, in dem Transaktionen und autonome Objekte im Auftrag von verschiedenen Benutzern durch den Austausch von Nachrichten agieren. Dies verlangt einige Modifikationen bzw. Erweiterungen des in [22] definierten RBAC-Modells, die in den folgenden beiden Abschnitten genauer beleuchtet werden.

4.3 Propagierung von Rollen in AutO

Neben Benutzern existieren in AutO weitere aktive Komponenten: Transaktionen und autonome Objekte. Auch für sie muß festgelegt werden, welche Rollen und damit Rechte sie besitzen.

Startet ein Benutzer in AutO eine Transaktion, so erbt diese die gerade aktiven Rollen des Benutzers, da sie alle weiteren Aktionen *in dessen Auftrag* ausführt. Genauso erbt ein autonomes Objekt die Rollen der erzeugenden Transaktion.

Nachrichten, die eine Transaktion absendet, enthalten auch die Rollen dieser Transaktion. Dadurch kann der Empfänger der Nachricht überprüfen, ob die

[5] Auch in Unix-Systemen wird der Systemverwalter `root` sich nur dann mit dieser Kennung einloggen, wenn es unbedingt nötig ist. Dadurch können überflüssige Fehler vermieden werden.

Transaktion autorisiert ist, die mit der Nachricht verbundene Operation auszuführen. Soll dann im Laufe dieser Operation eine Nachricht an ein anderes autonomes Objekt abgeschickt werden, so muß auch diese mit entsprechenden Rollen versehen werden. Dazu existieren zwei Möglichkeiten, wie in Abbildung 4 dargestellt:

- Die Rollen der zu sendenden Nachricht stimmen mit den Rollen der auslösenden Nachricht überein.
- Die ausgehende Nachricht wird mit den Rollen versehen, die dem autonomen Objekt zugeordnet sind.

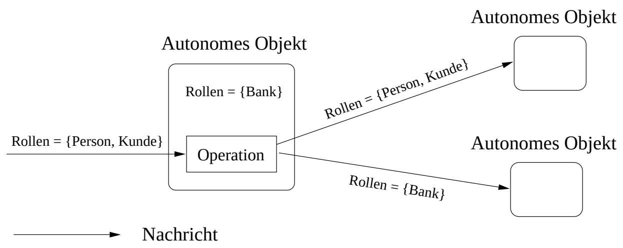

Abbildung 4. Propagierung von Rollen in autonomen Objekten

Da je nach Art des Objekts und der ausgeführten Operation beide genannten Möglichkeiten sinnvoll sein können, kann der Programmierer eines autonomen Objekts für jede Operation festlegen, welche der Alternativen gewählt werden soll.

4.4 RBAC in einem verteilten System wie AutO

Die in 4.2 eingeführten Mengen müssen – zumindest teilweise – auf jeder AutO-Site vorhanden sein. Caching bietet eine flexible und leistungsfähige Möglichkeit, nur diejenigen Teile der verschiedenen Mengen auf einer Site zu speichern, die regelmäßig benötigt werden. Bei RBAC reicht es aus, nur die beiden Relationen UR und RR zu puffern, da sie alle notwendigen Informationen enthalten.

Im folgenden wird der Cache, der Teile der UR-Relation puffert, mit *User-Cache* bezeichnet, derjenige für die RR-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 Session startet. Da die Zahl dieser Rechner gering ist, kann man ein Callback-Locking-ähnliches Verfahren zur Erhaltung der Kohärenz des User-Caches wählen, das die

Gültigkeit der Puffereinträge zu jedem Zeitpunkt garantiert. Der zweite Cache wird *lazy* verwaltet, d. h. Daten werden nur aktualisiert, wenn es unbedingt notwendig ist, weil etwa auf sie zugegriffen wird. Die Strategie, die dazu verwendet wird, baut auf der Kohärenz des User-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 User-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* abgelegt. Benötigt eine AutO-Site eine sicherheitsrelevante Information, die nicht im entsprechenden lokalen Cache vorhanden ist, so muß sie bei der Security Data Administration angefordert werden.

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

- die Identität des Benutzers, d. h. unter anderem dessen Namen,
- die ihm zugeordneten Rollen und
- die Versionsnummern dieser Rollen.

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

Die Strategie, die für die Kohärenz des Caches sorgt, ist an das in Client/Server-Datenbanken bekannte Callback-Locking-Verfahren [6] 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. Sind die gewünschten Daten nicht im Puffer vorhanden, so wird eine entsprechende Anfrage an die Security Data Administration gesendet. Diese sendet die Informationen zurück, merkt sich aber gleichzeitig in einem Verzeichnis die AutO-Site, welche sie angefordert hat. Ändern sich die Daten in der Security Data Administration – nur hier können sie modifiziert werden – werden alle AutO-Sites, die nun veraltete Versionen dieser Daten gepuffert haben, benachrichtigt. Sie entfernen daraufhin die ungültigen Kopien aus ihrem Puffer.

Eine AutO-Site greift in zwei unterschiedlichen Situationen auf den User-Cache zu: zum einen, wenn ein Benutzer versucht, eine Session zu starten, zum anderen, wenn er eine Rolle aktivieren will.

Der Start einer Session fordert von einer AutO-Site, daß sie die Zugehörigkeit des Benutzers zur Menge **User** überprüft. Dazu versucht sie, die entsprechende User-Information aus dem User-Cache zu laden. Scheitert dies, werden sie sie bei der Security Data Administration angefordert. Im Erfolgsfalle werden die Daten anschließend in den lokalen User-Cache eingefügt. Ansonsten ist der Benutzer keine autorisierte Person. Ihm ist es damit nicht erlaubt, eine Session zu starten.

Die Rollenautorisierung stellt die zweite Situation dar, in der auf den User-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 User-Information enthalten, die nach dem Start der Session gepuffert ist.

Der Rollen-Cache Die Teile der RR-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 Rechte.

Das Update dieses Caches erfolgt *lazy*, d. h. erst dann, wenn es unbedingt nötig ist. Zugriffe auf auf diesen Puffer erfolgen nur bei der Rechteautorisierung. Dabei wird überprüft, ob die aktivierten Rollen eines Benutzers ihm erlauben, ein bestimmtes Recht auszuüben. 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 das gewünschte Recht ausüben darf oder nicht.

Entscheidend für die Korrektheit dieser Vorgehensweise ist, daß die Rollen-Informationen gültig sind, d. h. denen der Security Data Administration entsprechen. Diese Gültigkeit kann festgestellt werden, indem man die Versionsnummer der aktivierten Rolle – sie ist in der User-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. Eine aktuelle Version muß in diesem Fall bei der Security Data Administration angefordert und der Cache aktualisiert werden (Abbildung 5).

Wichtig ist die Erkenntnis, daß die Versionsnummern der aktivierten Rollen bzw. die vollständige User-Information *immer* gültig sind. Dies wird durch den Cachingalgorithmus des User-Caches garantiert.

5 Schutz autonomer Objekte vor unsicheren Sites

Ein weiterer Aspekt der Sicherheit ergibt sich in verteilten Systemen, wenn sie mobile Objekte unterstützen. Migration bietet viele Vorteile, birgt aber auch einige Gefahren, da zum einen nicht alle Sites als Aufenthaltsort für ein Objekt gleich geeignet sind, und zum anderen die Objekte während des Versendens

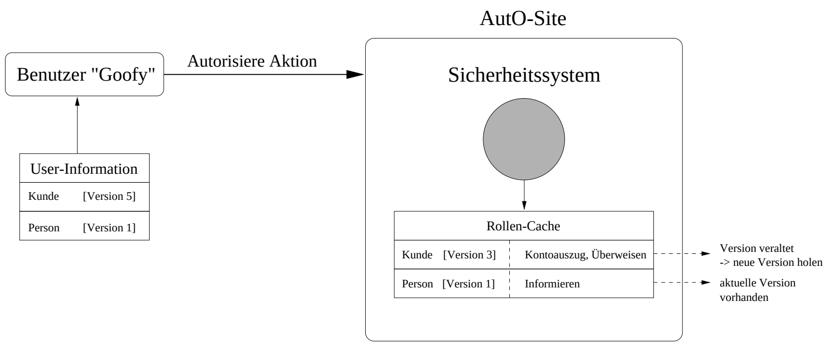

Abbildung 5. Caching von Rollen auf einer AutO-Site

über das Netzwerk angreifbar sind. Durch Verwendung sicherer Kommunikationskanäle kann das Objekt während der Migration geschützt werden, das Problem der verschieden sicheren Sites läßt sich dadurch nicht beheben.

Die Sicherheit, die ein Rechner einem Objekt bieten kann, hängt von verschiedensten Faktoren ab, wie z. B. dem zugrundeliegenden Betriebssystem, dem Standort des Rechners und dem verwendeten Datenbanksystem, das AutO für die Persistenz seiner Objekte benutzt. Dem Schutz der Objekte liegt in AutO ein einfacher Gedanke zugrunde: es soll für Objekte, die wichtige Informationen enthalten, nicht möglich sein, auf Rechner zu migrieren, bei denen nicht garantiert werden kann, daß das Objekt dort sicher ist. Auf diese Weise kann man sicherstellen, daß ein Objekt mit geheimen Informationen nicht auf einen Rechner migriert, auf dem normale Benutzer Zugriff auf beliebige Speicherbereiche oder die Datenbank haben, da dort die geheimen Informationen des Objektes kompromittiert werden können.

Um zu erreichen, daß Objekte nur auf Rechner migrieren können, die für sie ausreichend sicher sind, stellt AutO ein Konzept zur Kategorisierung von Rechnern und autonomen Objekten zur Verfügung, das es ermöglicht, die erlaubten Migrationsziele eines Objekts zu spezifizieren. Angelehnt ist dieses Konzept an das aus der Autorisierung bekannte Mandatory Access Control Modell [3].

Bei der Kategorisierung von Rechnern und Objekten kommen Sicherheitsklassen zum Einsatz. Dabei handelt es sich um Tupel (S, K), die aus einer Sicherheitsstufe S und einer Menge von Kategorien K bestehen und folgende Bedeutung haben:

Sicherheitsstufe: Diese Zahl zwischen 0 und 10 beschreibt die Sicherheitsanforderungen eines Objekts ($0 \equiv$ Unklassifiziert, $10 \equiv$ Höchste Sicherheitsanforderungen) bzw. für welche Objekte ein Rechner geeignet ist.

Kategorien: Diese beschreiben die Zugehörigkeiten der Objekte bzw. der Rechner zu verschiedenen, anwendungsspezifischen Bereichen. Im Falle einer Firma könnten geeignete Kategorien z. B. „Produktion“, „Verwaltung“ und „Personal“ sein.

Um diese Sicherheitsklassen verwenden zu können, muß außerden eine partielle Ordnung definiert werden, die Sicherheitsklassen in Relation zueinander setzt: Seien $SK_1 = (S_1, K_1)$ und $SK_2 = (S_2, K_2)$ Sicherheitsklassen. Dann gilt:

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

Eine Migration ist immer dann erlaubt, wenn die Sicherheitsklasse des Objekts kleiner oder gleich der Sicherheitsklasse des Zielrechners ist, also der Rechner eine genügend hohe Sicherheitsstufe bietet und mindestens für die Kategorien zugelassen ist, die das Objekt fordert.
Der allgemeine Verlauf einer Migration sieht nun folgendermaßen aus:

- Objekt O wertet seine Migrationsstrategie aus und stellt fest, daß es auf den Rechner R migrieren will.
- Das Objekt O schickt eine Nachricht an den Rechner R, um diesem den Migrationswunsch mitzuteilen.
- Falls Rechner R das Objekt akzeptiert – also nicht wegen Überlast oder aus anderen Gründen ablehnt – sendet er als Antwort seine signierte Sicherheitsstufe S_R, wie unten erklärt, an O zurück.
- Falls $S_O \leq S_R$ ist, migriert O auf den Rechner R, ansonsten wird die Migration abgebrochen.

Eine effizientere Lösung erhält man, wenn zusätzlich eine Strategie implementiert wird, der Wissen über die Netztopologie und die Sicherheitsklassen der Rechner im System zur Verfügung gestellt wird. Dadurch ist es leicht möglich, sich zuerst den Rechner auszusuchen, der am geeignetsten für das Objekt ist. Besitzt dieser Rechner keine passende Sicherheitsklasse, muß man lediglich einen Rechner in der Umgebung[6] des eigentlich gewünschten Zielrechners suchen, der den Sicherheitsanforderungen des Objekts genügt, um eine erfolgreiche Migration durchführen zu können.

Um zu verhindern, daß die Sicherheitsklasse eines Rechners unautorisiert verändert wird, wird diese von einer zentralen Zertifizierungsinstanz signiert. Der öffentliche Schlüssel, der nötig ist, um die Signaturen der Sicherheitsklassen zu überprüfen, muß dabei jedem Rechner zur Verfügung stehen. Die Sicherheitsklassen der Objekte werden dagegen vom Programmierer des Objekts festgelegt und können nachträglich vom Sicherheitsadministrator des Systems verändert werden. Da hierzu das im Kapitel 4.2 beschriebene Autorisierungsverfahren verwendet wird, ist sichergestellt, daß Änderungen der Sicherheitsklasse eines Objekts immer autorisiert sind. Aus diesem Grunde ist es nicht nötig, die Sicherheitsklassen der Objekte zu signieren.

6 Schutz der Sites vor Angriffen durch Objekte

In einem verteilten System, das Migration unterstützt, müssen nicht nur Objekte vor unsicheren Sites geschützt werden, auch Rechner müssen vor eventuellen

[6] Zum Beispiel Rechner im selben LAN oder in einem LAN mit einer guten Netzverbindung zu dem eigentlich gewünschten LAN

Attacken autonomer Objekte sicher sein. So darf es einem Objekt weder möglich sein, das laufende AutO-System einer Site zu manipulieren, noch unautorisierten Zugriff auf das lokale Dateisystem oder andere lokale Ressourcen zu besitzen. Da Java, wie z. B. auch Telescript [27], für derartige Probleme, die durch mobilen Code[7] entstehen, ein geeignetes Sicherheitskonzept zur Verfügung stellt (siehe [20]), bietet es sich an, dieses an die Bedürfnisse von AutO anzupassen.

6.1 Anforderungen an ein Sicherheitskonzept zum Schutz von AutO-Sites

Ein Sicherheitskonzept für AutO muß in der Lage sein, alle schädlichen Aktionen eines autonomen Objektes zu verhindern. Diese Aufgabe gliedert sich dabei in zwei Teile: Kontrolle der Zugriffe auf lokale Ressourcen und Kontrolle der Zugriffe autonomer Objekte auf Systemkomponenten von AutO.

Der Zugriff auf lokale Ressourcen wird in Java durch einen Security Manager kontrolliert, der vor jedem Zugriff auf das Dateisystem, auf das Netzwerk oder andere Systemressourcen befragt wird, ob die Aktion erlaubt ist. Die Aufrufe des Security Managers sind dabei in den Java-Systemklassen eingebaut, können also nicht umgangen werden. Die Verwendung von lokalen Ressourcen ist schon im Hinblick auf die Migrationsfähigkeit der Objekte bedenklich, wird aber noch kritischer, wenn man betrachtet, welchen Schaden Objekte durch derartige Aktionen auslösen können: Ein Objekt mit Zugriff auf das Dateisystem könnte von Rechner zu Rechner migrieren und vertrauliche Dateien lesen oder löschen. Aus diesen Gründen ist autonomen Objekten normalerweise jeglicher Zugriff auf Ressourcen des Rechners, auf dem sie ausgeführt werden, untersagt. Um dies zu erreichen, wurde ein AutO Security Manager entwickelt, der auf den Möglichkeiten von Java 1.2 aufbaut. Auf die Realisierung dieses Security Managers unter Verwendung der von Java bereitgestellten Hilfsmittel wird im Kapitel 6.2 näher eingegangen.

Trotz der Verwendung eines Security Managers ist das System nicht vor Attacken sicher, die die ordnungsgemäße Arbeit des Systems behindern. So ist es für ein autonomes Objekt durchaus möglich, Unmengen von Speicherplatz für sich zu reservieren und dadurch den regulären Betrieb von AutO zu stören oder sogar ganz zu verhindern. Da es keine allgemeine und praktikable Lösung dieses Problems gibt, werden die Klassendateien der benutzerdefinierten Objekte von einer Zertifizierungsstelle signiert. Dadurch lassen sich derartige Attacken gegen das System zwar nicht verhindern, aber der Verursacher kann – wenn das problemverursachende Objekt identifiziert ist – leicht festgestellt werden.

Eine weitere Anforderung an das Sicherheitskonzept von AutO ist, daß es autonomen Objekten den Zugriff auf das AutO-System nur über sichere Schnittstellen erlauben darf. Dies ist erforderlich, weil in jedem AutO-Prozeß verschiedene Verwaltungs- und Dienstleistungsobjekte existieren, darunter z. B. eine Objektverwaltung und eine Kommunikationsinstanz, die das Versenden von Nachrichten im System übernimmt. Verhindert man den unkontrollierten Zugriff auf

[7] Eine sehr bekannte Form von mobilem Code sind Applets.

diese Objekte bzw. deren Klassen nicht, könnte sich ein benutzerdefiniertes Objekt eigene Instanzen dieser Klassen erzeugen oder die Daten einer laufenden Instanz manipulieren. Ein weiteres Problem wäre, daß die Methoden der Dienstleistungsobjekte mit falschen Parametern aufgerufen werden könnten. Damit wäre es möglich, z. B. über die Kommunikationsinstanz Nachrichten mit einer gefälschten Absenderadresse zu verschicken. Um dies zu verhindern, wird sowohl für Transaktionen als auch für autonome Objekte jeweils eine Basisklasse zur Verfügung gestellt, die alle notwendigen Verbindungen zu den AutO-Systemkomponenten implementiert. Benutzerdefinierten autonomen Objekten ist es lediglich erlaubt, über diese Methoden auf verschiedene kritische Komponenten des AutO-Systems zuzugreifen. Dies kann man beispielsweise durch einen eigenen Class Loader für AutO sicherstellen, der das Laden benutzerdefinierter Objekte übernimmt. Dadurch kann man genau kontrollieren, welche Klassen von autonomen Objekten geladen werden. Verhindert man dabei das Laden aller kritischen AutO-Klassen außer der Basisklassen von Transaktionen und autonomen Objekten (`Transaction` und `UserObject`), dienen diese als sichere Schnittstelle zwischen den benutzerdefinierten Objekten und dem AutO-System. Die Arbeitsweise des AutO Class Loaders wird in Kapitel 6.3 genauer erklärt.

6.2 Schutz der lokalen Ressourcen eines Rechners

Wie bereits angesprochen, wurde in AutO ein eigener Security Manager implementiert, um die Ressourcen eines Rechners vor unerlaubten Zugriffen autonomer Objekte zu schützen. Der Security Manager von Java ist prinzipiell in der Lage, folgende Aktionen im Bedarfsfall zu unterbinden:

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

In Java 1.2 trifft der Security Manager seine Entscheidungen unter Verwendung eines Access Controllers, der basierend auf einer konfigurierbaren Sicherheitspolitik bestimmt, ob eine der obigen Aktionen von einer bestimmten Klasse ausgeführt werden darf. Dabei können Rechte in Abhängigkeit des Herkunftsortes[8] und/oder der Signatur einer Klasse vergeben werden.

[8] angegeben als URL

Bei der Rechtevergabe können die AutO-Systemklassen als vertrauenswürdig eingestuft werden und unterliegen deshalb keinerlei Restriktionen. Den Klassen aus dem Verzeichnis für benutzerdefinierte Objekte werden normalerweise keine Rechte gegeben, wodurch alle oben aufgezählten Aktionen verhindert werden, wenn sie von autonomen Objekten oder Transaktionen angestoßen werden.

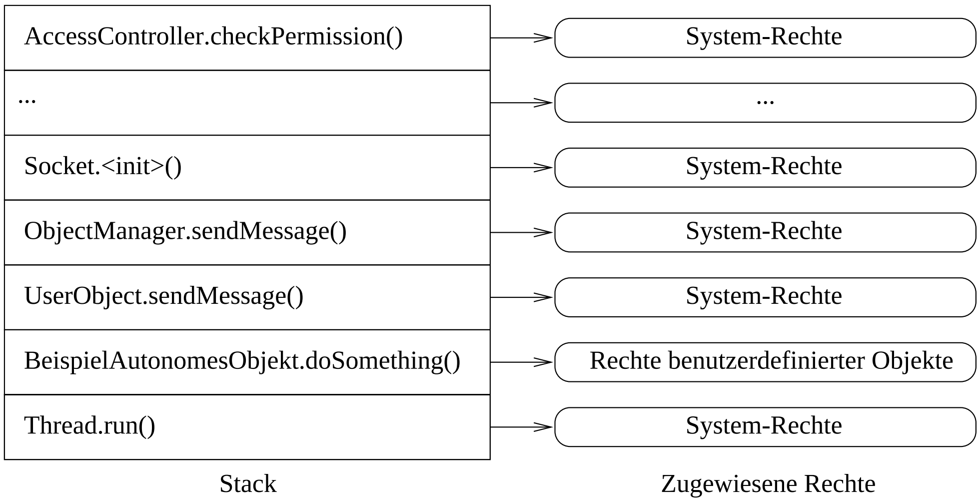

Abbildung 6. Beispielstack eines autonomen Objekts

Aufgrund der Arbeitsweise des Access Controllers sind allerdings Änderungen am System nötig, da dieser Aktionen normalerweise nur dann erlaubt, wenn alle Klassen, die auf dem Stack[9] liegen, das Recht haben, diese auszuführen (einen Beispielstack zeigt Abbildung 6). Aus diesem Grund muß man Aktionen im AutO-System, die von benutzerdefinierten Objekten angestoßen werden können, aber für diese nicht erlaubt sind, speziell behandeln. Der Access Controller kontrolliert alle Klassen auf dem Stack von oben nach unten, endet aber bei der ersten Klasse, die ihre gerade auf dem Stack liegende Methode als *privilegierten Code* markiert hat. Nötig ist das z. B. bei Socket-Verbindungen, die AutO nur im Bedarfsfall herstellt. Greift ein benutzerdefiniertes Objekt auf die Methode einer Systemkomponente (im Beispiel ist das die Objektverwaltung `ObjectManager`) von AutO zu, die einen dieser On-Demand-Sockets verwendet, muß u. U. von dieser zuerst die Verbindung hergestellt werden. Da das benutzerdefinierte Objekt als Aufrufer der Methode natürlich auf dem Stack liegt, würde der Access Controller dies verhindern. Wird das Öffnen des Sockets innerhalb

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

der AutO-Systemkomponente als privilegierter Code gekennzeichnet, wird diese Aktion vom Access Controller gestattet.

6.3 Schutz des AutO-Systems

Um das AutO-System vor Angriffen durch benutzerdefinierte Objekte zu schützen, wurde ein spezieller Class Loader für AutO implementiert, der dazu verwendet wird, die Klassen von Transaktionen und autonomen Objekten in die JVM zu laden. Versucht ein autonomes Objekt, andere Klassen – entweder explizit z. B. durch Aufruf einer Methode des Class Loaders oder implizit durch Java veranlaßt (z. B. Basisklassen, Klassen von Membervariablen oder von Parametern) – nachzuladen, verwendet Java dazu automatisch wieder den AutO Class Loader, da immer der Class Loader mit dem Nachladen von Klassen beauftragt wird, der die Klasse des anfordernden Objektes geladen hat. Außerdem definiert jeder Class Loader in AutO seinen eigenen Namensraum. Dadurch sind die Klassen, die dem AutO-System bekannt sind, und die, die den autonomen Objekten zur Verfügung stehen, vollständig unabhängig voneinander.

Der AutO Class Loader besteht aus mehreren verschiedenen Class Loadern (siehe Abbildung 7), an die nacheinander das Laden einer Klasse delegiert wird, bis die angeforderte Klasse gefunden wurde, oder bis feststeht, daß die Klasse entweder nicht geladen werden darf oder nicht existiert. Dabei wird nach folgendem Schema vorgegangen:

- Zuerst wird versucht, die Klasse innerhalb der Java-Systemklassen zu lokalisieren. Da diese Klassen bei allen eventuell gefährlichen Aktionen den Security Manager befragen, stellen sie kein Sicherheitsrisiko dar und dürfen von autonomen Objekten geladen werden. Der AutO Class Loader verwendet zum Laden der Klasse einen `URLClassLoader`, der die geforderte Klasse in den Verzeichnissen sucht, die die Java-Systemklassen beinhalten.
- Handelt es sich nicht um eine Systemklasse von Java, wird überprüft, ob die Klasse in einer Liste enthalten ist, die alle diejenigen Klassen des AutO-Systems beinhaltet, die von autonomen Objekten geladen werden dürfen. Die wichtigsten Klassen sind dabei `UserObject` und `Transaction`, die die Basisklassen aller autonomen Objekte bzw. Transaktionen in AutO bilden. Diese spezifizieren alle Methoden, die autonomen Objekten bzw. Transaktionen zur Verfügung stehen, um mit AutO zu kommunizieren. So beinhaltet `UserObject` z. B. eine Methode, die es ermöglicht, anderen Objekten Nachrichten zu senden; `Transaction` beinhaltet zusätzlich Methoden, um autonome Objekte zu erzeugen. Gehört die angeforderte Klasse zu dieser Art von AutO-Klassen, wird sie von dem Class Loader geladen, der auch alle anderen Klassen des AutO-Systems geladen hat.
- Ist die gesuchte Klasse auch keine der zugänglichen AutO-Klassen, wird als nächstes überprüft, ob das Package, in das die Klasse geladen werden soll, von anderen Klassen im CLASSPATH belegt ist. Sollte das der Fall sein, wird das Laden der Klasse verweigert. Diese Sicherheitsmaßnahme ist nötig, um zu verhindern, daß ein benutzerdefiniertes Objekt z. B. in eines der `java.*`

oder der Packages von AutO eingeschleust wird und dadurch privilegierte Zugriffsrechte auf andere Klassen in diesem Package erlangt.

– Anschließend wird versucht, die Klasse in dem Verzeichnis zu lokalisieren, in dem die benutzerdefinierten Objekte abgelegt sind. Wird die Klasse dort gefunden, wird sie vom AutO Class Loader selbst geladen.

  Ist sie dort nicht zu finden, wird bei einem zentralen Informationsserver nachgefragt, ob die gesuchte Klasse bei ihm registriert ist. Ist das der Fall, wird die Adresse einer Site geliefert, die die Klasse lokal installiert hat. Von dieser wird die Klasse angefordert, in dem Verzeichnis für benutzerdefinierte Klassen installiert und dann vom AutO Class Loader geladen.
– Schlug auch dieser letzte Versuch fehl, die Klasse zu laden, ist diese weder auf dem lokalen Rechner vorhanden noch registriert. Das Laden der Klasse scheitert also.

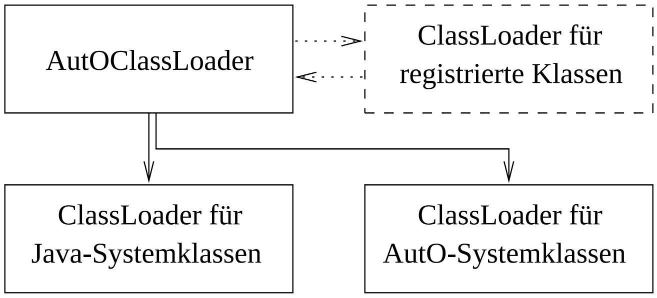

Abbildung 7. Delegationsmodell des AutO Class Loaders

Durch das eben beschriebene Vorgehen beim Laden einer Klasse erreicht man, daß benutzerdefinierten Klassen nur sichere Einstiegspunkte in das AutO-System offen stehen. Beispielsweise stehen einer Transaktion dazu lediglich die Methoden zur Verfügung, die in `Transaction` implementiert wurden (siehe Abbildung 8). Da `Transaction` selbst und alle zugehörigen Basisklassen durch den normalen Class Loader geladen werden, hat `Transaction` Zugriff auf alle Klassen von AutO, unterliegt selbst also keinen Beschränkungen und kann dadurch ihren Aufgaben im System ungehindert nachkommen. Der AutO Class Loader vergibt lediglich den Auftrag, `Transaction` zu laden, deshalb ist ihm nur diese Klasse bekannt. Alle weiteren Basisklassen sind dem AutO Class Loader verborgen. Versucht eine benutzerdefinierte Transaktion, eine Methode einer Basisklasse von `Transaction` aufzurufen, versucht der AutO Class Loader, die entsprechende Basisklasse nachzuladen und scheitert. Auch die Methoden und Variablen der Basisklassen von `Transaction`, die zur internen Verwaltung und Steuerung von Transaktionen gedacht sind, sind damit vor unbefugten Zugriffen sicher.

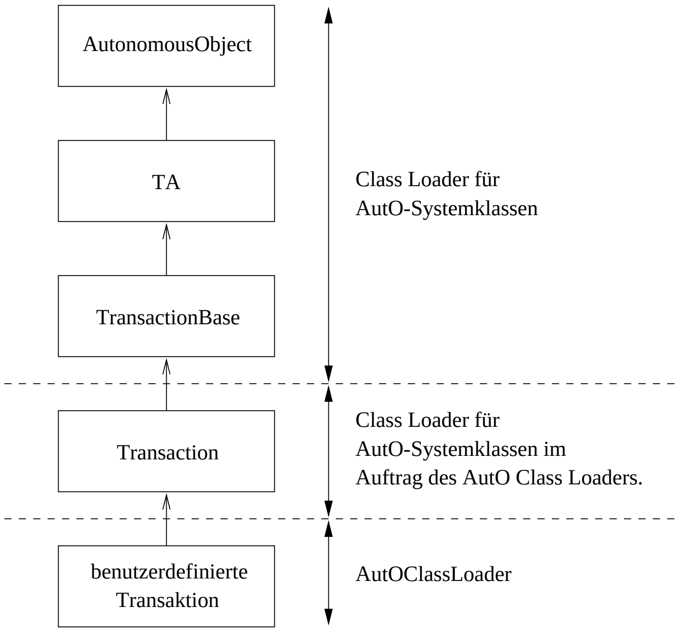

Abbildung 8. Aufgabenverteilung beim Laden einer Transaktion

7 Zusammenfassung

In dieser Arbeit wurde gezeigt, wie verschiedene Sicherheitsmechanismen in ein persistentes, verteiltes System (autonomer) Objekte integriert werden können. Als erstes wurde das System um ein rollenbasiertes Autorisierungsverfahren erweitert, das es ermöglicht, Benutzern verschiedene Rollen, und damit Rechte, zuzuordnen. Außerdem wurden Sicherheitsprobleme behandelt, die durch die Migration von Objekten entstehen. Dazu wurde ein MAC-ähnlicher Mechanismus in das System integriert, der sicherstellt, daß Objekte nur auf Sites migrieren, die ihren Sicherheitsansprüchen genügen. Zum Schutz der Sites vor feindlichen Objekten, wurden die von Java angebotenen Sicherheitskonzepte genutzt, um zu verhindern, daß die Objekte unerlaubte Aktionen ausführen.

8 Danksagungen

Wir danken den restlichen Mitgliedern des AutO-Teams (Stefan Grießer, Markus Islinger und Stefan Pröls). Unser Dank gilt außerdem den anonymen Gutachtern für die hilfreichen Kommentare und Professor Gudes von der Ben Gurion Universität (Israel) für die anregenden Diskussionen.

Literatur

[1] ARNOLD, K. und J. GOSLING: *The Java Programming Language.* Addison-Wesley, Reading, MA, USA, 1998.

[2] CAREY, M. J., D. J. DEWITT, M. J. FRANKLIN, N. E. HALL, M. L. MCAULIFFE, J. F. NAUGHTON, D. T. SCHUH, M. H. SOLOMON, C. K. TAN, O. G. TSATALOS, S. J. WHITE und M. J. ZWILLING: *Shoring Up Persistent Applications.* In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seiten 383–394, Minneapolis, MI, USA, Mai 1994.

[3] CASTANO, S., M. G. FUGINI, G. MARTELLA und P. SAMARATI: *Database Security.* ACM Press. Addison-Wesley, Reading, MA, USA, 1995.

[4] FERNANDEZ, E. B., R. C. SUMMERS und C. WOOD: *Database Security and Integrity.* Addison-Wesley, 1981.

[5] FERRAIOLO, DAVID F. und RICHARD KUHN: *Role-Based Access Control.* 15th NIST-NCSC National Computer Security Conference, Seiten 554–563, Oktober 1992.

[6] FRANKLIN, M.: *Client Data Caching: A Foundation.* Kluwer Academic Press, 1996.

[7] GAL-OZ, N., E. GUDES und E. B. FERNANDEZ: *A Model of Methods Access Authorization in Object-Oriented Databases.* In: *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Seiten 52–61, Dublin, Ireland, August 1993.

[8] HAMILTON, G., R. CATTELL und M. FISHER: *JDBC database access with Java: A Tutorial and Annotated Reference.* Addison-Wesley, Reading, MA, USA, 1997.

[9] *IAIK-JCE.* `http://jcewww.iaik.tu-graz.ac.at/IAIK-JCE/jce.htm`.

[10] JONSCHER, D. und K. R. DITTRICH: *An Approach for Building Secure Database Federations.* In: *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Seiten 24–35, Santiago, Chile, September 1994.

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

[12] KORTH, H. F.: *Locking Primitives in a database system.* Journal of the ACM, 30(1):55–79, Januar 1983.

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

[14] KRIVOKAPIĆ, N., M. ISLINGER und A. KEMPER: *Migrating Autonomous Objects in a WAN Environment.* Technical Report MIP-9811, Universität Passau, 94030 Passau, Germany, August 1998.

[15] LIU, L. und R. MEERSMAN: *Activity Model: A Declarative Approach for Capturing Communication Behaviour in Object-Oriented Databases.* In: *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Seiten 481–493, Vancouver, Canada, August 1992.

[16] LIU, L. und R. MEERSMAN: *The Building Blocks for Specifying Communication Behavior of Complex Objects: An Activity-Driven Approach.* ACM Transactions on Database Systems, 21(2):157–207, Juni 1996.

[17] LOCKEMANN, P. C. und H.-D. WALTER: *Activities in Object Bases.* In: PATON, N. W. und M. H. WILLIAMS (Herausgeber): *Proc. of the First Intl. Workshop on Rules in Database Systems*, Workshops in Computing, Seiten 3–22, Edinburgh, United Kingdom, September 1993. Springer-Verlag.

[18] Merz, M., K. Müller-Jones und W. Lamersdorf: *Mobile Klienten: Ortsübergreifender Zugang zu Diensten in offenen verteilten Informationssystemen.* In: *Proc. der 25. GI-Jahrestagung*, Seiten 423–430, 1995.

[19] Müller, G., U. Kohl und R. Strauss (Herausgeber): *Zukunftsperspektiven der digitalen Vernetzung.* dpunkt Verlag, Heidelberg, 1996.

[20] Oaks, Scott: *Java Security.* O'Reilly & Associates, Sebastopol, CA, 1998.

[21] Pernul, G.: *Database Security.* Advances in Computers, 38:1–72, 1994.

[22] Sandhu, Ravi S., Edward J. Coyne, Hal L. Feinstein und Charles E. Youman: *Role-Based Access Control Models.* IEEE Computer, 29(2):38–47, 1996.

[23] Schwarz, P. M. und A. Z. Spector: *Synchronizing Shared Abstract Types.* ACM Trans. Computer Systems, 2(3):223–250, 1984.

[24] Stonebraker, M., P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin und A. Yu: *Mariposa: A wide-area distributed database system.* The VLDB Journal, 5(1):48–63, Januar 1996.

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

[26] Sun Microsystems Inc.: *JavaSpaces: Innovative Java Technology that Simplifies Distributed Application Development.* http://java.sun.com/products/javaspaces/whitepapers/, Juli 1998.

[27] Tardo, J. und L. Valente: *Mobile Agent Security and Telescript.* In: *In Proc. COMPCON*, Seiten 58–63, Februar 1996.

[28] U.S. Department of Defense: *Trusted Computer Security Evaluation Criteria.* DOD 5200.28-STD, 1985.