Eberhard Karls
Universität
Tübingen

Fakultät für Informations- und Kognitionswissenschaften
Wilhelm-Schickard-Institut für Informatik

Anwendungs- und Transaktionsisolation unter Java

Diplomarbeit

Betreut von

Prof. Dr.-Ing. Wilhelm G. Spruth

Vorgelegt von

Jens Müller
(Matrikelnummer 2200565)

16. Mai 2005

Zusammenfassung

Der Schwerpunkt der Java 2 Platform, Enterprise Edition liegt im Bereich serverseitiger Geschäftsanwendungen. Transaktionsverarbeitung ist eine der wichtigsten Schlüsseltechnologien im Umfeld von Geschäftsanwendungen. Die Verbreitung objekt- und komponentenorientierter Entwicklungskonzepte führte zur Entwicklung von objektorientierten Transaktionsmonitoren. Mit Einführung der Java 2 Platform, Enterprise Edition und der serverseitigen Komponententechnologie Enterprise JavaBeans, entstanden auch auf Java basierende Transaktionsverarbeitungssysteme.

Im Lauf der Zeit wandelte sich die Java Plattform von einer einfachen virtuellen Maschine zu einer betriebssystemähnlichen Laufzeitumgebung. Im Zuge dieser Entwicklung wurde es insbesondere im Bereich der Transaktionsverarbeitung notwendig, Transaktionen innerhalb derselben virtuellen Maschine parallel zu verarbeiten, um einen akzeptablen Durchsatz zu erzielen. Auf der anderen Seite wurde auch der Wunsch laut, herkömmliche Anwendungen parallel ausführen zu können.

Java wurde jedoch nicht als virtuelles Betriebssystem konzipiert und es fehlt die bei Betriebssystemen übliche Abstraktion des Prozesses und in Folge dessen die damit verbundenen Mechanismen, wie z.B. Isolation und Ressourcenmanagement. Dennoch wurden Techniken entwickelt, die zumindest die Isolation garantieren sollten. Diese sind jedoch unzureichend und stellen keine zufriedenstellende Lösung dar. In vielen Szenarien mag die dadurch gebotene Isolation für die parallele Ausführung von Anwendungen und Transaktionen zwar ausreichend sein. Gerade aber bei der Transaktionsverarbeitung ist die Einhaltung der ACID-Eigenschaften und damit der Isolation einer der wichtigsten Aspekte. Soll das Transaktionsverarbeitungssystem absolute Transaktionssicherheit bieten, so ist daher derzeit vom Einsatz der EJB-Technologie abzuraten.

Diese Arbeit beschreibt die Defizite von Java in verschiedenen Bereichen, die in Zusammenhang mit der parallelen Ausführung von Anwendungen und Transaktionen relevant sind. Anschließend werden die analysierten Probleme praktisch nachvollzogen und diskutiert. Abschließend werden Lösungsansätze aufgezeigt und die derzeitigen Bemühungen einer offiziellen und umfassenden Lösung vorgestellt.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen benutzt und sowohl wörtliche, als auch sinngemäß entlehnte Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Tübingen, den 16. Mai 2005

(Jens Müller)

Danksagung

Ich danke meiner lieben Mutter und meinem leider viel zu früh verstorbenen Vater dafür, dass sie mir dieses Studium ermöglicht haben.

Herrn Professor Spruth danke ich herzlichst für die Betreuung dieser Arbeit und den Freiraum, den er mir dabei gegeben hat. Die vielen Gespräche, die wir teilweise bis spät in den Abend hinein geführt haben, werde ich in bester Erinnerung behalten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation

Java ist eine objektorientierte Programmiersprache, die von der Firma Sun entwickelt wurde. Als Java im Januar 1996 offiziell das Licht der Welt erblickte, hätte sicherlich niemand für möglich gehalten, welcher Siegeszug der Sprache bevorstehen sollte. In den darauf folgenden Jahren wurde die Weiterentwicklung schnell vorangetrieben. Bestand die Klassenbibliothek in den Anfängen aus noch rund 200 Klassen, so stieg der Umfang bis heute auf mehr als das Zehnfache an. Der große Vorteil von Java ist die Portabilität. Java-Anwendungen können ohne Modifikation auf unterschiedlichen Plattformen ausgeführt werden. Um dies zu erreichen, wird bei der Kompilierung ein Zwischencode erzeugt, der so genannte Bytecode. Er unterscheidet sich von Maschinencode durch seine Plattformunabhängigkeit. Der Bytecode wird innerhalb einer speziellen Laufzeitumgebung ausgeführt, der Java Plattform, deren zentraler Bestandteil die Java Virtual Machine (JVM, im Folgenden auch Virtual Machine) ist. Diese wiederum ist an jedes Betriebssystem angepasst.

Aufgrund der zunehmenden Komplexität wurde Java seit dem Erscheinen der Version 1.2 in drei Editionen unterteilt: In die Java 2 Platform, Micro Edition (J2ME), die Java 2 Platform, Standard Edition (J2SE) und die Java 2 Platform, Enterprise Edition (J2EE) [3]. Der Schwerpunkt der Java 2 Platform, Enterprise Edition liegt im Bereich serverseitiger Geschäftsanwendungen.

Transaktionsverarbeitung ist eine der wichtigsten Schlüsseltechnologien im Umfeld von Geschäftsanwendungen. Durch Transaktionen wird ein sicherer Zugriff konkurrierender Anwendungen auf gemeinsame Daten ermöglicht. Die Geschichte der Transaktionsverarbeitung reicht bis in die 60er Jahre des letzten Jahrhunderts zurück, als die ersten Flugplatzreservierungssysteme entstanden. Das wichtigste Transaktionsverarbeitungssystem ist das Customer Information Control System (CICS) [1] von IBM, das 1968 eingeführt wurde. Heutzutage werden mehr als 30 Milliarden Transaktionen mit einem Umsatz von einer Billion Dollar pro Tag von CICS verarbeitet. Hochleistungstransaktionssysteme, wie etwa ein Sysplex Cluster unter z/OS [2], können mehr als 15 000 Transaktionen pro Sekunde verarbeiten.

Die Verbreitung objekt- und komponentenorientierter Entwicklungskonzepte führte zur Entwicklung von objektorientierten Transaktionsmonitoren. Mit Einführung der Java 2 Platform, Enterprise Edition und der serverseitigen Komponententechnologie Enterprise JavaBeans (EJB) [4], entstanden auch auf Java basierende Transaktionsverarbeitungssysteme.

Durch die Verbreitung von Java angefangen beim Einsatz in mobilen Geräten wie Mobiltelefonen und Personal Digital Assistants bis hin zu Applikationsservern und Transaktionsverarbeitungssystemen wandelte sich die Java Plattform von einer einfachen virtuellen Maschine zu einer betriebssystemähnlichen Laufzeitumgebung. Im Zuge dieser Entwicklung wurde es insbesondere im Bereich der Transaktionsverarbeitung notwendig, Transaktionen innerhalb derselben virtuellen Maschine parallel zu verarbeiten, um einen akzeptablen Durchsatz zu erzielen. Auf der anderen Seite wurde auch der Wunsch laut, herkömmliche Anwendungen innerhalb derselben Virtual Machine parallel ausführen zu können.

Java wurde jedoch nicht als virtuelles Betriebssystem konzipiert und es fehlt die bei Betriebssystemen übliche Abstraktion des Prozesses und in Folge dessen die damit verbundenen Mechanismen, wie z.B. Isolation und Ressourcenmanagement. Dennoch wurden Techniken entwickelt, die zumindest die Isolation garantieren sollten. Diese sind jedoch unzureichend und stellen keine zufriedenstellende Lösung dar. In vielen Szenarien mag die dadurch gebotene Isolation für die parallele Ausführung von Anwendungen und Transaktionen zwar ausreichend sein. Gerade aber bei der Transaktionsverarbeitung ist die Einhaltung der ACID-Eigenschaften und damit der Isolation einer der wichtigsten Aspekte.

Diese Arbeit beschreibt die Defizite von Java in verschiedenen Bereichen, die in Zusammenhang mit der parallelen Ausführung von Anwendungen und Transaktionen relevant sind. Anschließend werden die analysierten Probleme praktisch nachvollzogen und diskutiert. Abschließend werden Lösungsansätze aufgezeigt und die derzeitigen Bemühungen einer offiziellen und umfassenden Lösung vorgestellt.

1.2 Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel untergliedert.

Kapitel 2 erläutert die Begriffe Transaktion und Transaktionsverarbeitung und stellt kurz die Komponententechnologie Enterprise JavaBeans vor.

Kapitel 3 beschäftigt sich mit dem Sicherheitskonzept von Java und den in Bezug auf Multitasking problematischen Bereichen.

Kapitel 4 dokumentiert die praktische Überprüfung der in Kapitel 3 aufgezeigten Probleme und diskutiert diese in verschiedenen Kontexten.

Kapitel 5 verschafft einen Forschungsüberblick der Thematik und ordnet die Lösungsansätze in vier verschiedene Bereiche ein.

Kapitel 6 stellt die derzeitigen Bemühungen hinsichtlich einer offiziellen Lösung vor.

Kapitel 7 fasst die Ergebnisse der Arbeit zusammen.

Im Anhang befindet sich der Quellcode der im Rahmen dieser Arbeit erstellten Software und eine Auflistung des Inhalts der beiliegenden DVD.

Kapitel 2

Transaktionsverarbeitung

2.1 Transaktionen

Jim Gray und Andreas Reuter definieren den Begriff ,,Transaktion" in ihrem Standardwerk ,,Transaction Processing: Concepts and Techniques" [GR93] folgendermaßen:

> A transaction is a collection of operations on the physical and abstract application state.

Das klassische Beispiel für eine Transaktion ist die Überweisung eines Geldbetrags von einem Konto auf ein anderes. Dieser Vorgang besteht aus mehreren Schritten. Es ist leicht einzusehen, dass bei einer Überweisung alle Schritte durchgeführt werden müssen, damit der Zustand der Konten konsistent bleibt. Außerdem müssen konkurrierende Zugriffe auf die Konten isoliert durchgeführt werden, so dass sie sich nicht gegenseitig beeinflussen. Diese und weitere Beobachtungen führen zu vier notwendigen Eigenschaften von Transaktionen, den so genannten ACID-Eigenschaften.

2.2 ACID-Eigenschaften

- **Atomicity (Unteilbarkeit):**
  Die Zustandsänderungen einer Transaktion sind atomar: Entweder werden alle durchgeführt oder keine, beispielsweise Datenbankänderungen.

- **Consistency (Konsistenzerhaltung):**
  Eine Transaktion ist eine korrekte Zustandstransformation. Alle durchgeführten Schritte verletzen keine der Integritätsbedingungen, die mit dem Zustand assoziiert sind. Dies erfordert, dass es sich bei einer Transaktion um ein korrektes Programm handelt.

- **Isolation (Abschirmung):**
  Auch wenn Transaktionen gleichzeitig ablaufen, wurden aus der Sicht jeder Transaktion andere Transaktionen ausschließlich entweder davor oder danach ausgeführt.

- **Durability (Dauerhaftigkeit):**
  Sobald eine Transaktion erfolgreich endet (COMMIT), überdauern ihre Zustandsänderungen Fehler.

2.3 Transaktionsverarbeitungssysteme

Ein Transaktionsverarbeitungssystem stellt Werkzeuge zur Verfügung, um die Programmierung, Ausführung und Administration von Anwendungen zu erleichtern oder zu automatisieren. Es handelt sich dabei um ein sehr komplexes System, bestehend aus Anwendungen, Ressourcenmanagern (z.B. Datenbankmanagementsystemen), Netzwerksteuerung, Entwicklungswerkzeugen und dem Transaktionsmonitor, einer Reihe von Diensten, die den eingehenden Transaktionsfluss verwalten und koordinieren. Die wichtigsten davon sind:

- **Transactional RPC:**
  Transactional RPC ermöglicht die Ausführung transaktionaler Remote Procedure Calls.

- **Transaktionsmanager:**
  Der Transaktionsmanager überwacht und koordiniert die Aktivitäten aller Transaktionsteilnehmer, steuert die Ausführung von Transaktionen gemäß des 2-Phasen-Sperrprotokolls und behebt Fehler bei Misserfolg.

- **Log Manager:**
  Der Log Manager protokolliert die Änderungen von Transaktionen, so dass eine konsistente Version aller Objekte im Falle eines Fehlers rekonstruiert werden kann.

- **Lock Manager:**
  Der Lock Manager stellt einen allgemeinen Mechanismus bereit, um gleichzeitigen Zugriff auf Objekte zu regeln. Dies hilft Ressourcenmanagern für Transaktionsisolation zu sorgen.

Beispiele für Transaktionsmonitore sind CICS von IBM, Tuxedo [9] von BEA und der COM+ [10] Transaktionsmonitor von Microsoft. Der Nachfolger von COM+/DCOM, das .NET Framework, enthält keinen eigenen Transaktionsmonitor, sondern greift auf die Transaktionsinfrastruktur von COM+ zurück. Der J2EE-Applikationsserver WebSphere [11] von IBM beinhaltet einen rein objektorientierten Transaktionsmonitor, einen Object Transaction Monitor (OTM), der nur für Enterprise Beans und Servlets ausgelegt ist. Dies gilt nicht für WebSphere unter z/OS.

Im Allgemeinen gibt es an jedem Standort oder Cluster eines Computernetzwerks ein eigenes Transaktionsverarbeitungssystem. Die Transaktionsmonitore dieser Systeme kooperieren miteinander, um dem Benutzer eine verteilte Ausführungsumgebung zur Verfügung zu stellen.

2.4 Enterprise JavaBeans

Enterprise JavaBeans ist eine serverseitige Komponententechnologie für die Entwicklung von verteilten Geschäftsanwendungen auf der Basis von Java, die unter Leitung der Firma Sun entstand. Sie ist Teil der J2EE-Architektur. Enterprise JavaBeans bietet weitgehend automatisierte Transaktions-, Persistenz-, Verteilungs- und Sicherheitsfunktionen. Die Implementierung dieser notwendigen Funktionen sollen Entwicklern von EJB-Anwendungen abgenommen werden, damit diese sich auf die Geschäftslogik konzentrieren können.

In einer Pressemitteilung [SM97] von Sun vom 2. April 1997 wird die EJB-Technologie erstmals offiziell erwähnt:

> Sun Microsystems, Inc., today launched a comprehensive Java Platform for the Enterprise, including a significant new technology initiative, Enterprise JavaBeans which uses Java to break through the complexity of building end-to-end business solutions. With Enterprise JavaBeans, developers can design and re-use small program elements to build powerful corporate applications. These "componentized" applications can run manufacturing, financial, inventory management, and data processing on any system or platform that is Java-enabled. [...] Enterprise JavaBeans will simplify development, deployment and maintenance of high-performance enterprise applications and provide the infrastructure to shield developers from system level complexity. [...] After initial review with industry partners is completed, the draft specification for Enterprise JavaBeans will be available in summer 1997. The final specification and implementation will be available by the end of the year.

Die erste EJB-Spezifikation wurde schließlich im März 1998 fertig gestellt. Im Dezember 1999 erschien die überarbeitete Version 1.1. Wichtige Verbesserungen wurden im August 2001 mit Version 2.0 der Spezifikation eingeführt. Die aktuelle Version 2.1 [SM03a] erschien im November 2003. Der zweite Entwurf der kommenden Version 3.0 [SM05] ist seit Februar 2005 verfügbar. Enterprise Beans sind gemäß der EJB-Spezifikation in Java geschriebene serverseitige Softwarekomponenten. Sie werden innerhalb einer speziellen Laufzeitumgebung ausgeführt, dem EJB-Container. Dieser stellt insbesondere die oben beschriebenen Funktionen zur Verfügung. Der EJB-Container selbst ist wiederum in einem Applikationsserver (siehe Abbildung 2.1) enthalten, der den Vorgaben der Java 2 Platform, Enterprise Edition entspricht. Der Applikationsserver kann weitere Container beinhalten, beispielsweise einen Web-Container für Java Server Pages (JSP) [5] und Servlets [6]. Externe Clients, wie z.B. Applets, Anwendungen oder CORBA-Clients [7] können per RMI-IIOP [8] auf Enterprise Beans zugreifen. Innerhalb des Servers können Servlets aber auch Enterprise Beans selbst als Clients fungieren.

Es gibt drei verschiedene Typen von Enterprise Beans: Entity, Session und Message-Driven Beans.

Entity Beans sind Komponenten, die eine objektorientierte Sicht auf persistent gespeicherte Entitäten, also eindeutig identifizierbare und über Attribute beschreibbare Informationseinheiten, repräsentieren. Diese Entitäten sind beispielsweise in einer Datenbank gespeichert. Handelt es sich in diesem Fall um eine relationale Datenbank, so korrespondieren Entity Beans jeweils mit genau einer Zeile der entsprechenden Datenbanktabelle. Entity Beans modellieren Geschäftskonzepte, beispielsweise ein Konto.

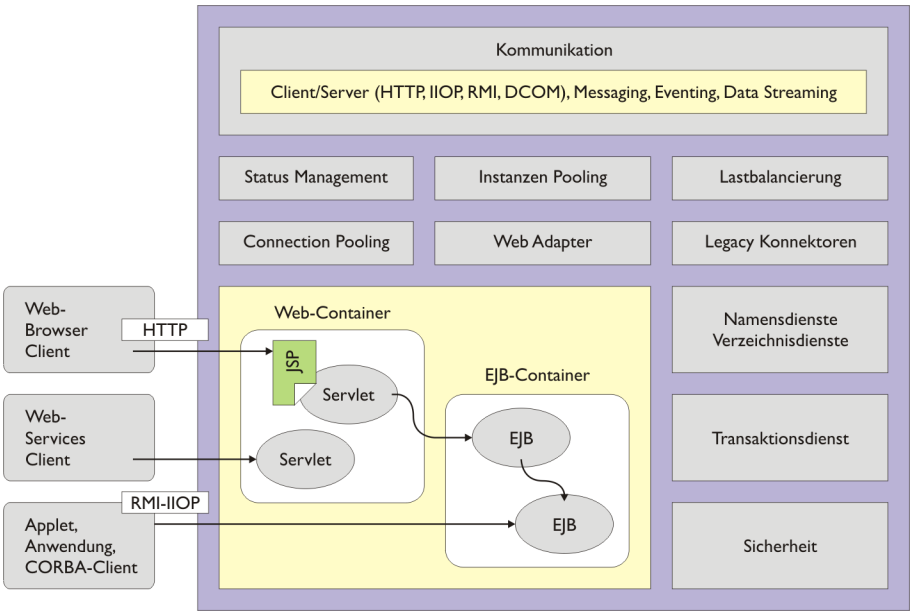

Abbildung 2.1: Die Bestandteile eines J2EE-Applikationsservers

Session Beans sind Komponenten, die auf dem Server ablaufende Geschäftslogik implementieren. Sie modellieren Geschäftsprozesse, wie z.B. eine Überweisung.

Message-Driven Beans entsprechen Session Beans, nur werden sie durch den Empfang einer Nachricht ausgelöst. Sie wurden erst mit Version 2.0 der Spezifikation eingeführt.

2.5 Transaktionen und Enterprise JavaBeans

Enterprise JavaBeans vereinfacht die Verwaltung von Transaktionen, da der EJB-Container die meisten Aufgaben bei der Ausführung von Transaktionen übernimmt. Sie werden daher auch Container-gesteuerte Transaktionen genannt. Transaktionssteuerung durch den EJB-Container wird auch als deklarative (implizite) Transaktionssteuerung bezeichnet, da Transaktionsattribute im Deployment Descriptor deklariert werden. Der Ablauf von Transaktionen kann aber auch explizit in der Bean-Klasse gesteuert werden. Man spricht dann von Bean-gesteuerten Transaktionen. Enterprise JavaBeans unterstützt neben lokalen Transaktionen, bei denen nur auf eine Datenbank zugegriffen wird, auch Transaktionen in verteilten Umgebungen mit mehreren Datenbanken.

Das Transaktionsverarbeitungssystem in Enterprise JavaBeans (siehe Abbildung 2.2) setzt sich aus folgenden Bestandteilen zusammen:

- **Transaktionsmanager:**
  Der Transaktionsmanager wurde bereits in Abschnitt 2.3 beschrieben. Die Schnittstelle zu Transaktionsmanagern ist in Enterprise JavaBeans die Java Transaktion API (JTA) ([SM02], [12]).

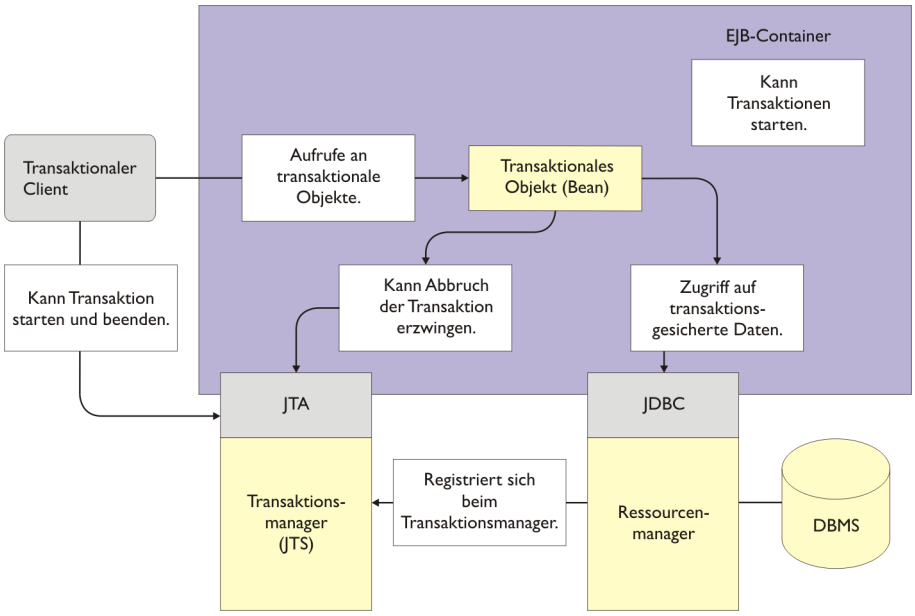

Abbildung 2.2: Das Transaktionsverarbeitungssystem in Enterprise JavaBeans

- **EJB-Container:**
  Der EJB-Container übernimmt in Zusammenarbeit mit dem Transaktionsmanager Verwaltungsaufgaben (z.B. Aufzeichnung des Transaktionskontexts). Container-gesteuerte Transaktionen werden vom EJB-Container gestartet. EJB-Container und Transaktionsmanager arbeiten bei der Ausführung von Transaktionen eng zusammen.

- **Ressourcenmanager:**
  Ressourcenmanager ermöglichen den Zugriff auf transaktionsgesicherte Daten (z.B. den Zugriff auf relationale Datenbanken über JDBC-Verbindungen). Sie registrieren sich beim Transaktionsmanager, sobald sie in eine Transaktion einbezogen werden.

- **Transaktionale Objekte:**
  Transaktionale Objekte nehmen an Transaktionen teil und greifen auf die von Ressourcenmanagern verwalteten Daten zu.

- **Transaktionale Clients:**
  Transaktionale Clients können Transaktionen starten. Java-Anwendungen, Applets, Servlets und Enterprise Beans werden als Clients von EJB-Anwendungen bezeichnet. Clients sind nicht zwangsläufig transaktionale Clients. Bei Container-gesteuerten Transaktionen haben Clients keinen Einfluss auf den Ablauf von Transaktionen.

Die EJB-Spezifikation schreibt vor, dass der EJB-Container die Java Transaction API, die aus mehreren Schnittstellen besteht, unterstützen muss.

Allerdings muss der EJB-Container Enterprise Beans nur die Schnittstelle `javax.transaction.UserTransaction` zur Verfügung stellen, damit Transaktionen Bean-gesteuert ablaufen können. Die anderen Schnittstellen sind zur Integration des Transaktions- und des Ressourcenmanagers vorgesehen. Der Transaktionsmanager muss die Java Transaction API ebenfalls unterstützen.

Optional kann der Transaktionsmanager die Schnittstellen des Java Transaction Service (JTS) ([SM99], [13]) implementieren, eine Java-Umsetzung der CORBA Object Transaction Service (OTS) 1.2.1 [OMG01] Spezifikation. Die Schnittstelle `javax.jts.TransactionService` erlaubt dem Object Request Broker (ORB), sich gegenüber dem Transaktionsmanager zu identifizieren. Dem Transaktionsmanager erlaubt sie, dem ORB die Sender- und Empfängerschnittstellen zu übergeben. Die OTS-Schnittstellen `org.omg.CosTransactions` und `org.omg.CosTSPortability` gewährleisten Interoperabilität und Portabilität. Sie definieren einen Standardmechanismus, der es jeder IIOP (bzw. RMI-IIOP) basierten Implementierung erlaubt, Transaktionskontext zu erzeugen und zwischen JTS Transaktionsmanagern zu propagieren.

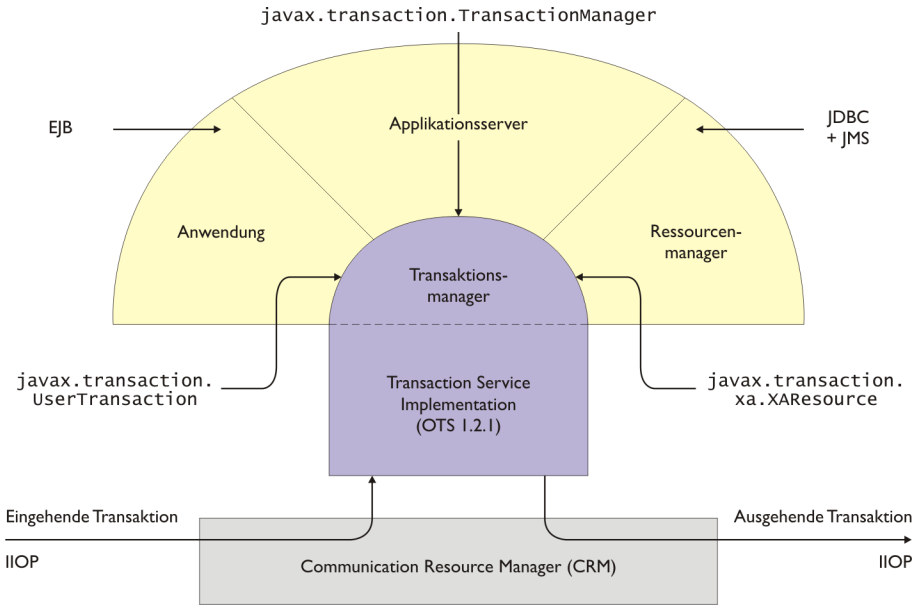

Abbildung 2.3: JTA und JTS

Abbildung 2.3 zeigt einen Transaktionsmanager, der die JTS-Schnittstellen implementiert. Die Java-Anwendung, der Applikationsserver und der Ressourcenmanager kommunizieren jeweils über eine andere JTA-Schnittstelle mit dem Transaktionsmanager. Die gestrichelte Linie veranschaulicht die internen Schnittstellen, die Zugriff auf die OTS-Implementierung ermöglichen. Der Transaktionsmanager ist nicht dazu verpflichtet, seine OTS-Implementierung Benutzern zugänglich zu machen, die ihn über die `javax.transaction.TransactionManager` Schnittstelle ansteuern.

2.6 Parallele Transaktionsverarbeitung

Um möglichst viele Transaktionen in einem gewissen Zeitraum zu verarbeiten, muss die Verarbeitung parallel durchgeführt werden. Eine sequentielle Abarbeitung wäre undenkbar, da Transaktionen, die sich über mehrere Minuten, Tage oder gar Wochen hinziehen (so genannte Long-Running Transactions), die Ausführung anderer Transaktionen verzögern würden.

Üblicherweise sind J2EE-Applikationsserver selbst in Java programmiert, was vor allem auf den Vorteil der Plattformunabhängigkeit und die damit leichtere Wartbarkeit zurückzuführen ist, da nur eine Codebasis vorhanden ist. Dies ist beispielsweise bei WebSphere unter AIX [14], Linux [15] und Windows [16] der Fall. Lediglich WebSphere unter z/OS bildet hier eine Ausnahme. Die J2EE-Spezifikation [SM03b] schreibt nicht vor, wie ein J2EE-Applikationsserver partitioniert sein muss. Es bleibt dem Produktanbieter überlassen, ob der Applikationsserver als einzelner Prozess implementiert ist, oder aus mehreren Prozessen innerhalb eines oder mehrerer Netzwerkknoten besteht.

Die parallele Ausführung von Transaktionen kann auf zwei Arten implementiert werden. Die erste Möglichkeit ist, für jede Transaktionen eine eigene Virtual Machine zu starten. Diese Methode scheidet aber aufgrund der schlechten Performance und des enormen Speicherverbrauchs aus. Die zweite Möglichkeit ist die Verwendung von Threads, die Java in Form der Klasse `Thread` bereitstellt. Die Benutzung von Threads allein reicht aber nicht aus, um Transaktionen unterschiedlicher EJB-Anwendungen sicher parallel auszuführen. Anwendungen und Transaktionen, die in derselben Virtual Machine ausgeführt werden, dürfen sich nicht gegenseitig beeinflussen können. Insbesondere kritische Transaktionen erzwingen eine strikte Einhaltung der ACID-Eigenschaften, in denen auch die Isolation gefordert wird. Dabei wird zwischen zwei Arten der Isolation unterschieden. Der klassische Isolationslevel bei Transaktionen bestimmt, in welchem Maß benötigte Ressourcen gesperrt werden. Er untergliedert sich in vier Stufen: Uncommitted Read, Read Committed, Read Stability und Serializable. Die andere Art der Isolation wird an dieser Stelle als Request Isolation bezeichnet und beschreibt die gegenseitige Beeinflussung parallel ablaufender Transaktionsthreads. In weiteren Kapiteln steht der Begriff Isolation bei Transaktionen immer für Request Isolation.

Kapitel 3

Isolation unter Java

3.1 Das Sicherheitskonzept von Java

Java [GJSB05] wird als sichere Sprache bezeichnet. Diese Sicherheit basiert auf vier grundlegenden Techniken:

- **Bytecode-Verifikation:**
  Durch Bytecode-Verifikation wird sichergestellt, dass dieser der Spezifikation entspricht. Diese Überprüfung findet beim erstmaligen Zugriff auf eine Klasse statt.

- **Typsicherheit:**
  Programme dürfen nur Operationen auf Instanzen von Typen ausführen, die die Sprache als sinnvoll erachtet. Unsichere Typumwandlungen (z.B. von `int` nach `Object`) sind nicht möglich und damit auch die Fälschung von Objektreferenzen.

- **Automatische Speicherbereinigung:**
  Automatische Speicherbereinigung (Garbage Collection) ist unabdingbar, um Typsicherheit zu gewährleisten, da diese durch noch bestehende Referenzen auf explizit gelöschte Objekte außer Kraft gesetzt werden würde.

- **Speicherschutz:**
  Der Speicherschutz verhindert Buffer Overflows bei Vektor- und Stackoperationen, z.B. beim Zugriff auf einen Index außerhalb des zulässigen Bereichs.

Darüber hinaus verwendet Java zwei Kontrollmechanismen, um über den Zugriff in Zusammenhang mit Objekten zu entscheiden:

- **Statische Zugriffskontrolle:**
  Statische Zugriffskontrollmechanismen stellen fest, welche Operationen ein Codesegment auf einem Objekt durchführen darf. Beispielsweise ist es einer Methode erlaubt, auf private Felder der Klasse zuzugreifen, in der sie definiert wurde, während dies von außerhalb nicht möglich ist.

- **Dynamische Zugriffskontrolle:**
  Dynamische Zugriffskontrollmechanismen überprüfen anhand des Call Stacks (Call Stack Inspection), also durch Rückverfolgung des Methodenaufrufs eines Objekts, ob das aufrufende Objekt über die dafür benötigte Berechtigung verfügt.

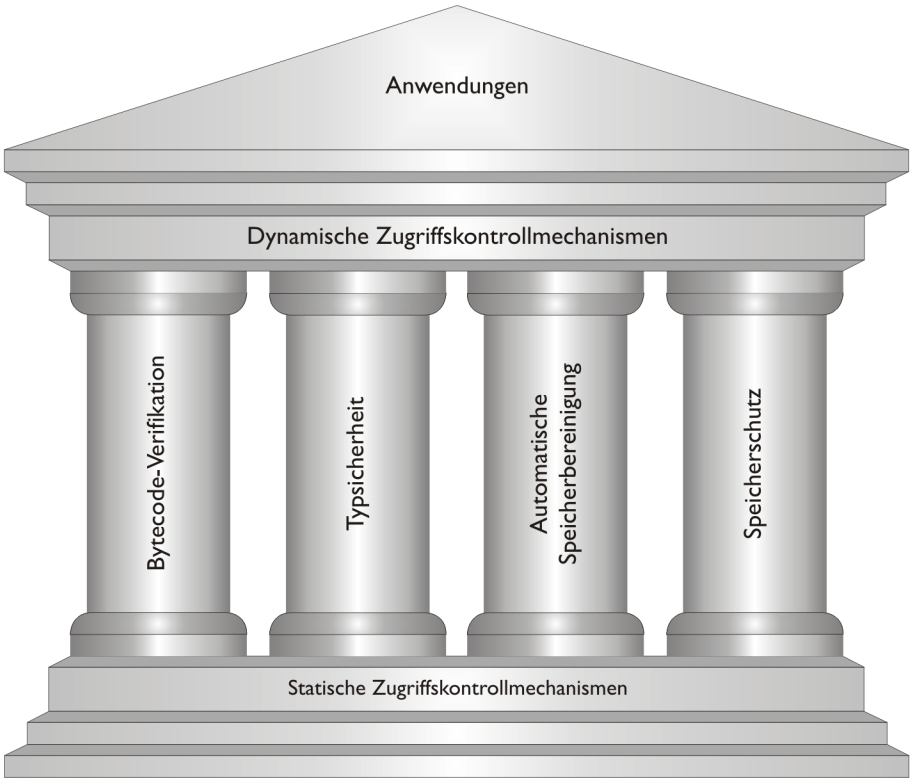

Abbildung 3.1: Das Sicherheitskonzept von Java

Das Sicherheitskonzept von Java bietet Anwendungen, die parallel innerhalb derselben Virtual Machine ausgeführt werden, eine gewisse Isolation. Beispielsweise ist es nicht möglich, dass eine Anwendung durch Fälschung einer Objektreferenz Daten einer anderen Anwendung verändert.

An dieser Stelle zeigt sich, dass die Trennlinie zwischen Programmiersprache und Betriebssystem bei Java zunehmend verschwimmt, was vor allem durch die Unterteilung in Programmiersprache und Laufzeitumgebung deutlich wird.

Java ist jedoch nicht als virtuelles Betriebssystem konzipiert worden. Das Sicherheitskonzept von Java stellt den inneren Schutz von Anwendungen sicher, es adressiert aber nicht die Bereiche, in denen Probleme bei der parallelen Ausführung von Anwendungen innerhalb derselben Laufzeitumgebung auftreten können: Isolation, Interprozesskommunikation, Ressourcenmanagement und die sichere Beendigung von Anwendungen. Sie werden in den nächsten Abschnitten näher beschrieben.

Java fehlt es an Mechanismen, wie sie in Betriebssystemen zur Lösung von Problemen in diesen Bereichen eingesetzt werden. Durch Hinzufügen dieser Funktionalität könnte Java durchaus ein virtuelles Betriebssystem auf Grundlage sprachbasierter Sicherheitsmechanismen darstellen. Sprachbasierte Sicherheit unterscheidet sich von der Sicherheit herkömmlicher Betriebssysteme, die meist auf adressbasierten Mechanismen aufbaut und Funktionalität der Hardware in Anspruch nimmt. Die Erweiterung sprachbasierter Sicherheit um Betriebssystemkonzepte allgemein und ein Vergleich zwischen adressbasierter und sprachbasierter Sicherheit sind Inhalt zweier Arbeiten, die an der Cornell Universität entstanden ([HE98], [Haw00]).

3.2 Defizite im Bereich der Isolation

Java fehlt die Abstraktion des Prozesses, wie sie von Betriebssystemen her bekannt ist, also z.B. eine Klasse, die einen Prozess repräsentiert. Der gängige Ansatz zur Isolation von Anwendungen ist die Benutzung der Class Loader, die im nächsten Abschnitt vorgestellt werden. Anhand ihrer lässt sich zwar eine Definition für Prozesse formulieren, diese ist aber nicht zufriedenstellend.

3.2.1 Mathematischer Exkurs: Binäre Relation, transitive Hülle

In den folgenden Abschnitten und Kapiteln wird der Begriff der transitiven Hülle benötigt, der hier kurz erläutert werden soll. Grundlage dafür sind in diesem Fall binäre Relationen.

Allgemein ist eine Relation eine Beziehung, die zwischen etwas bestehen kann. In der Mathematik ist eine binäre Relation R eine Teilmenge des kartesischen Produkts $A \times B$ zweier Mengen A und B:

$$R \subseteq A \times B \; mit \; A \times B := \{(a,b) | (a \in A) \wedge (b \in B)\}$$

Beispiel: Sei $\Re$ eine Relation, die beschreibt, dass zwei Klassen direkt miteinander in Verbindung stehen. Eine Klasse X steht direkt mit einer anderen Klasse Y in Verbindung, falls innerhalb von Klasse X der Name von Klasse Y auftaucht (z.B. als Typ einer lokalen Variable).

Sei nun A eine Menge, die aus den Klassen a, b und c besteht, wobei Klasse a direkt in Verbindung mit Klasse b und Klasse b direkt in Verbindung mit Klasse c steht. Diese Beziehung kann mathematisch so ausgedrückt werden:

$$A = \{a, b, c\}$$

$$A \times A = \{(a,a), (a,b), (a,c), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c)\}$$

$$\Re := steht \; direkt \; in \; Verbindung \; mit$$

$$\Re \subseteq A \times A = \{(a,b), (b,c)\}$$

Die transitive Hülle R^+ einer binären Relation R ist die Erweiterung der Relation um alle indirekt erreichbaren Paare.

Die mathematische Definition sieht folgendermaßen aus:

$$(x,y) \in R^+$$

$$\Leftrightarrow$$

$$\exists n \geq 2 : \exists x = x_1, ..., x_n = y \in M : \forall i \in \{1, ..., n-1\} : (x_i, x_{i+1}) \in R$$

Im obigen Beispiel besteht die transitive Hülle $\Re^+$ aus folgenden Paaren:

$$\Re^+ = \{(a,b), (b,c), (a,c)\}$$

Klasse a steht also indirekt in Verbindung mit Klasse c.

3.2.2 Class Loader

Bei der Kompilierung von Java-Klassen entsteht plattformunabhängiger Bytecode. Jede Klasse wird dabei in einer eigenen `class`-Datei gespeichert. Bei der Ausführung eines Java-Programms werden die von der Hauptklasse (die die `main`-Methode enthält) verwendeten Klassen und die wiederum mit ihnen direkt in Verbindung stehenden oder indirekt über die transitive Hülle erreichbaren Klassen aber nicht auf einmal in den Speicher geladen. Stattdessen werden Klassen dynamisch in den folgenden zwei Fällen geladen:

- Objekterzeugung durch den `new`-Operator
- Statische Referenz (z.B. `System.out`)

Für das Laden von Klassen sind Class Loader zuständig. Class Loader sind Instanzen von Klassen, die von der abstrakten Klasse `ClassLoader` abgeleitet sind. Jede Virtual Machine verfügt über einen standardmäßigen Class Loader, den so genannten Primordial Class Loader. Der Primordial Class Loader kann nur Klassen aus dem lokalen Dateisystem laden. Benutzerdefinierte Class Loader ermöglichen das Laden von Klassen von beliebigen Quellen, beispielsweise verwenden Applets Instanzen der Klasse `AppletClassLoader`, die Klassen unter Benutzung des Hypertext Transfer Protocols (HTTP) über das Netzwerk laden kann.

Der Einstiegspunkt der Klasse `ClassLoader` ist die Methode `loadClass`. Ihr wird der Name der zu ladenden Klasse übergeben. Der Class Loader verwaltet die bereits geladenen Klassen. Bevor eine neue Klasse geladen wird, überprüft ein benutzerdefinierter Class Loader normalerweise, ob dies bereits geschehen ist und kontrolliert anschließend, ob es sich um eine Systemklasse handelt. Falls dem so ist, wird das Laden der Klasse an den Primordial Class Loader delegiert.

3.2.3 Klassennamensräume

Ein Java-Prozess kann als Menge aus Threads definiert werden, die in einer Threadgruppe (`ThreadGroup`) verwaltet werden. Der Klassennamensraum eines Prozesses wird durch den Class Loader definiert, der die Hauptklasse der Anwendung lädt (z.B. die Klasse, die die `main`-Methode enthält). Der Klassennamensraum eines Class Loaders enthält Klassen, die er selbst geladen hat und alle oder eine Teilmenge der Klassen, die der übergeordnete Class Loader geladen hat. Beispielsweise könnte der Klassennamensraum eines benutzerdefinierten Class Loaders die durch ihn geladenen Klassen und die durch den Primordial Class Loader geladenen Systemklassen enthalten. In den folgenden Abschnitten stehen Prozesse für diese Definition.

Ein Thread kann nur auf Objekte zugreifen, die sich in der Objekthülle der mit ihm in derselben Threadgruppe verwalteten Threads befinden. Die Objekthülle eines Prozesses beinhaltet sämtliche Objekte, die während der Ausführung seiner Threads erzeugt wurden und weiterhin von deren Stacks aus erreichbar sind (z.B. durch lokale Variablen), sowie alle Objekte, die wiederum von diesen referenziert werden. Dies setzt sich rekursiv fort. Des Weiteren enthält sie alle Objekte, die von statischen Feldern von Klassen im Klassennamensraum des Prozesses referenziert werden.

Der Klassennamensraum enthält also in einer Datenstruktur gespeicherte Klassen, die als Vorlage für Instanzen, bzw. Objekte dieser Klasse dienen. Diese Objekte werden zur Laufzeit erzeugt. Die Objekthülle hingegen ist ein rein logisches Konstrukt. Sie umfasst alle Objekte, die ausgehend von den Threads und statischen Feldern von Klassen im Klassennamensraum des Prozesses erreichbar sind.

3.2.4 Sichere und unsichere statische Felder

In Bezug auf die Isolation kann man zwischen sicheren und unsicheren statischen Feldern unterscheiden. Sichere statische Felder können weder dazu benutzt werden, Objektreferenzen zwischen Prozessen auszutauschen, noch dazu, den Zustand eines Prozesses zu modifizieren. Konstante Felder (die mit dem Modifizierer `final` deklariert wurden) primitiven Typs, wie z.B. `int`, `float`, `char` und konstanten Klassentyps, wie etwa `Long`, `Double` und `String`, sind sichere statischen Felder. Ein Beispiel für ein sicheres statisches Feld ist das statische Feld `Integer.MAX_VALUE`.

Unsichere statische Felder sind alle nicht konstanten Felder und konstante Felder komplexerer Klassen, die nicht gekapselt sind, d.h. statische Felder, die anderen Klassen außerhalb ihres Pakets zugänglich sind oder den Zugriff durch statische Methoden (Klassenmethoden) ermöglichen. Beispielsweise ist das statische Feld `System.out` (der Standardausgabestrom) unsicher, da ein Prozess den Strom schließen könnte und ihn dadurch für alle anderen Prozess ebenfalls schließen würde.

Klassen, die unsichere statische Felder enthalten, werden unsichere Klassen genannt. Auch sind Klassen unsicher, die Methoden enthalten, bei deren Ausführung Nebenwirkungen mit Auswirkungen auf andere Prozesse auftreten können, wie es bei den Klassen `Runtime` und `System` der Fall ist. Deren `exit`-Methoden beenden die Virtual Machine und damit alle anderen Prozesse.

3.2.5 Multitasking unter Verwendung eines Class Loaders

Teilen sich Prozesse innerhalb derselben Virtual Machine denselben Class Loader, so teilen sie sich auch denselben Klassennamensraum. Die Schnittmenge ihrer Objekthüllen enthält alle Objekte, die von statischen Feldern der gemeinsamen Klassen referenziert werden. Objekte innerhalb dieser Schnittmenge können von allen Prozessen manipuliert werden, was eventuell die Integrität eines Prozesses verletzt.

In Abbildung 3.2 erzeugt Prozess 1 das Objekt `x` und weist es dem statischen Feld `obj` der Klasse `A` zu. Prozess 2 weist daraufhin den Wert dieses Felds der lokalen Variable `y` zu. Damit hat Prozess 2 Zugriff auf ein Objekt innerhalb der Objekthülle von Prozess 1 und könnte es manipulieren, was die Integrität von Prozess 1 verletzen würde.

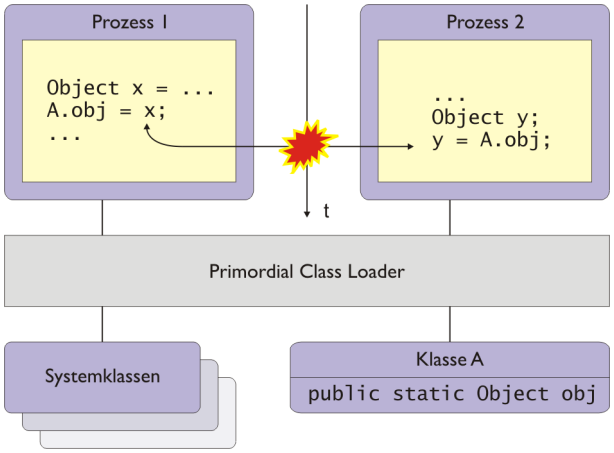

Abbildung 3.2: Multitasking unter Verwendung eines Class Loaders

3.2.6 Multitasking unter Verwendung mehrerer Class Loader

Um dieses und weitere Probleme bei der Ausführung mehrerer Prozesse innerhalb derselben Virtual Machine zu lösen, wird üblicherweise jeder Prozess und die von ihm verwendeten Klassen von einem separaten Class Loader geladen [LB98]. Dadurch ändert sich die Semantik statischer Felder von globalen Variablen auf JVM-Ebene zu globalen Variablen auf Prozessebene.

In Abbildung 3.3 verwenden beide Prozesse jeweils ihren eigenen Class Loader, um die Klasse `A` zu laden. Manipulationen an statischen Feldern der Klasse `A` durch einen Prozess hat keinen Einfluss auf statische Felder der Klasse `A` des jeweils anderen Prozesses.

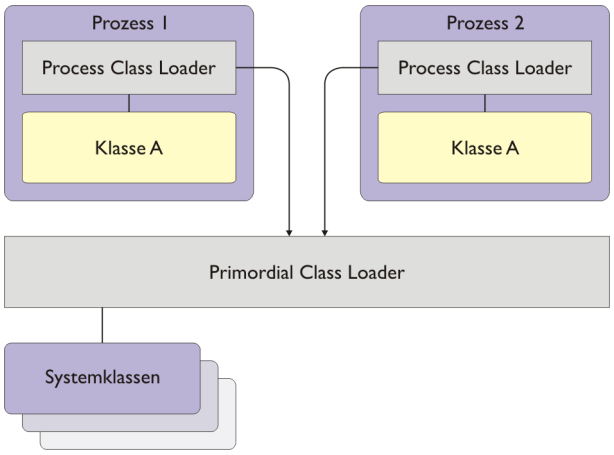

Abbildung 3.3: Multitasking unter Verwendung mehrerer Class Loader

Diese Methode wird auch von J2EE-Applikationsservern verwendet, um EJB-Anwendungen zu isolieren. Class Loader werden jedoch nicht dazu verwendet, Transaktionen derselben EJB-Anwendung zu isolieren, da hier die Verwendung statischer Variablen verboten und damit die Verwendung der Class Loader überflüssig ist. Zusätzlich würde bei Verwendung der Class Loader der Transaktionsdurchsatz durch das ständige Laden von Anwendungsklassen bei jeder Transaktion verringert.

3.2.7 Probleme bei der Verwendung mehrerer Class Loader

Im Vergleich zur Verwendung eines gemeinsamen Class Loaders bieten mehrere Class Loader einen relativ hohen Isolationsgrad. Die Isolation ist aber unvollständig, da statische Felder und synchronisierte Klassenmethoden von Systemklassen nicht repliziert werden.

Bei synchronisierten Klassenmethoden kann ein Problem auftreten, falls ein Thread, der sich innerhalb einer solchen Methode befindet, von einem anderen Thread desselben Prozesses ausgesetzt wird und sie in Folge dessen von anderen Prozessen nicht mehr ausgeführt werden kann. Der Grund dafür ist, dass es sich bei synchronisierten Klassenmethoden um Monitore handelt. Ein Monitor [SGG03] ist ein Konstrukt, mit dessen Hilfe Prozesse, bzw. Threads synchronisiert werden können. In einem Monitor kann zu jedem Zeitpunkt nur ein Thread aktiv sein. Erst wenn der ausgesetzte Thread wieder aufgenommen wird und die synchronisierte Klassenmethode verlässt, kann diese von anderen Threads ausgeführt werden.

Ein weiterer Nachteil entsteht durch die Replikation von Anwendungsklassen durch verschiedene Class Loader, die eigentlich der Isolation zugute kommt. Diese Vervielfachung ist aber bei Verwendung von Just-In-Time Compilern besonders problematisch, da Klassen unabhängig voneinander kompiliert und gespeichert werden, auch wenn sie bereits von einer anderen Anwendung geladen wurden. In diesem Fall wird erheblich mehr Speicher benötigt, da ein Byte Bytecode im Durchschnitt in fünf bis sechs Bytes Maschinencode übersetzt wird [CFM+97].

3.2.8 Weitere Isolationsdefizite

Die parallele Ausführung von Anwendungen unter Verwendung von Threads und mehrerer Class Loader ist zwar möglich, weitere Stellen im Design der Virtual Machine belegen aber, dass sie nicht für Multitasking konzipiert wurde und dadurch Probleme verursacht, die bei der Ausführung einer einzigen Anwendung nicht auftreten [CD01].

Bestimmte Datenstrukturen der Laufzeitumgebung werden von allen Anwendungen gemeinsam verwendet, z.B. gibt es nur eine einzige Datenbasis für internalisierte Strings (siehe Abschnitt 4.5). Werden internalisierte Strings als Sperrobjekte in `synchronized`-Abschnitten verschiedener Anwendungen benutzt, so kann dies zu unerwarteten Interaktionen zwischen den Anwendungen führen, sofern es sich um identische Zeichenketten handelt. Darüber hinaus handelt es sich bei der synchronisierten Klassenmethode `String.intern()` [17] um einen Monitor (siehe Abschnitt 3.2.7), der von allen innerhalb der Virtual Machine ausgeführten Anwendungen zugänglich ist. Wird eine große Anzahl an Strings von mehreren Anwendungen parallel der gemeinsamen Datenbasis für internalisierte Strings hinzugefügt (z.B. bei XML-Parsern), so könnten an dieser Stelle unerwartete Leistungseinbußen auftreten.

Besonders problematisch ist der für alle Objekte verantwortliche Finalisierungsthread. Finalisierer, die nach ihrem Aufruf beispielsweise aufgrund einer Endlosschleife nicht zurückkehren, blockieren den Finalisierungsthread, was wiederum zur Folge hat, dass keine weiteren Objekte finalisiert werden können.

Auch werden Ereignisse der grafischen Benutzeroberfläche von einem zentralen Thread entgegengenommen, was zu ähnlichen Problemen führen kann: Solange eine Ereignisbehandlungsroutine nicht endet, können nachfolgende Ereignisse nicht verarbeitet werden. Dies kann im schlimmsten Fall dazu führen, dass alle Prozesse in der Virtual Machine, die eine grafische Benutzeroberfläche verwenden, nicht mehr reagieren. Im Kontext von Enterprise JavaBeans sind derartige Probleme irrelevant, da die EJB-Spezifikation den Zugriff auf das Abstract Window Toolkit (AWT), die Standardschnittstelle für grafische Benutzeroberflächen, verbietet.

Die Einbindung plattformabhängigen Codes per Java Native Interface (JNI) kann ebenfalls zu Problemen führen. Plattformabhängiger Code wird im selben Adressraum wie die Virtual Machine ausgeführt und hat somit uneingeschränkten Zugriff auf alle Anwendungsdaten. Enterprise Beans dürfen wiederum keine plattformabhängigen Bibliotheken laden, der J2EE-Applikationsserver greift aber über die J2EE Connector Architecture (JCA) ([SM01], [18]) häufig auf plattformabhängigen Code zu. Ist dieser fehlerhaft, kann dies zum Absturz des Applikationsservers führen.

3.3 Defizite im Bereich der Interprozesskommunikation

Java-Prozesse kommunizieren üblicherweise durch gegenseitige Methodenaufrufe und Übergabe von Objektreferenzen. Diese unkontrollierte Ausbreitung von Referenzen birgt jedoch Nachteile.

Erstens ist es einem Prozess, der eine Referenz an einen anderen Prozess übergeben hat, nicht möglich, den Zugriff auf das ab diesem Zeitpunkt gemeinsam verwendete Objekt zu wiederrufen. Dies untergräbt aber den im Englischen als Principle of Least Privilege bezeichneten Grundsatz, der in diesem Fall besagt, dass ein Prozess nur so lange wie nötig auf eine Ressource (bzw. auf ein Objekt) Zugriff erhalten sollte.

Zweitens verschwimmen mit immer mehr gemeinsam verwendeten Objekten die Grenzen zwischen Java-Prozessen, da nicht mehr nachvollziehbar ist, welche Prozesse Zugriff auf ein Objekt haben. Noch undurchsichtiger wird diese Angelegenheit, wenn Prozesse, denen Objektreferenzen übergeben wurden, diese wiederum an Dritte weiterreichen. Für den Programmierer wird es immer schwieriger, zwischen gemeinsam verwendeten und nicht gemeinsam verwendeten Objekten zu unterscheiden. Diese Unterscheidung ist aber essentiell, da bei Objekten der ersten Kategorie jederzeit damit gerechnet werden muss, dass sie von anderen Prozessen modifiziert worden sein könnten. Wird dieser Sachverhalt außer Acht gelassen, sind Integrität und Sicherheit von Prozessen gefährdet. Die gemeinsame Verwendung von Objektreferenzen führt aber auch in anderen Bereichen zu Problemen.

3.4 Defizite im Bereich des Ressourcenmanagements

Was der Java-Laufzeitumgebung gänzlich fehlt, ist die Möglichkeit, Ressourcen zu kontrollieren. Beispielsweise könnte eine Anwendung sehr viel Speicher in Anspruch nehmen und so die Ausführung anderer Anwendungen verhindern.

Von Prozessen gemeinsam verwendete Objektreferenzen werfen auch Fragen im Bereich des Ressourcenmanagements auf. Beispielsweise ist unklar, wer für den Speicher verantwortlich ist, den ein gemeinsam verwendetes Objekt belegt. Falls dem Erzeugerprozess der Speicher ausgeht, könnte er das Objekt freigeben wollen und deswegen alle Referenzen auf `null` setzen. Jedoch wird die Speicherbereinigung den Speicher nicht freigeben, solange andere Prozesse dieses Objekt referenzieren. Diese können so Speicher blockieren, für den sie nicht verantwortlich sind. Um beide Prozesse für die Verwaltung des Objekts verantwortlich zu machen, müssten die Verwaltungsinformationen der Prozesse bei jeder Zuweisung aktualisiert werden.

Bei der Suche nach einer Lösung stellt sich auch die Frage, ob und wie der von gemeinsam verwendeten Objekten belegte Speicher auf die jeweiligen Prozesse aufgeteilt werden sollte. Eine Möglichkeit wäre, den Speicher gleichmäßig zu verteilen und die Anteile erneut zu kalkulieren, falls ein weiterer Prozess hinzukommt, bzw. wegfällt. Der aus dieser Berechnung resultierende Speicherverbrauch eines Prozesses kann aber ein asynchrones Verhalten aufweisen, da er ohne explizite Speicheranforderung ansteigen kann (siehe Abbildung 3.4).

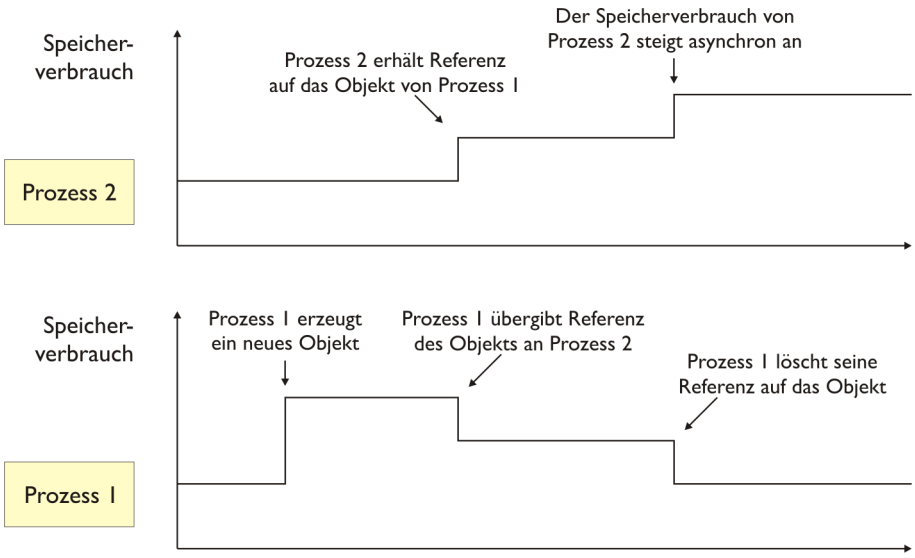

Abbildung 3.4: Asynchrones Verhalten des Speicherverbrauchs eines Prozesses

3.5 Defizite im Bereich der Beendigung von Prozessen

Zu alledem bietet Java keine Möglichkeit, Anwendungen explizit zu beenden. Von der Verwendung der Primitiva `Thread.stop()`, `Thread.suspend()`, `Thread.resume()` und `Runtime.runFinalizersOnExit(boolean)` wird seit Version 1.2 abgeraten [19]. Dafür gibt es mehrere Gründe [HE99a]:

- **Unterbrechung zum falschen Zeitpunkt**:
  Falls sich ein Thread zum Zeitpunkt seiner Unterbrechung oder Beendigung innerhalb eines Systemaufrufs befindet, könnte das System in einem inkonsistenten Zustand hinterlassen werden oder ein Deadlock auftreten.

- **Beendigung bösartigen Codes nicht möglich**:
  Die Beendigung aller Threads einer bösartigen Anwendung beendigt nicht zwangsweise den Code der Anwendung, der in überschriebenen Methoden von Objekten, die durch die Anwendung erzeugt und wiederum an andere Anwendungen weitergegeben wurden, weiterhin existiert. Wenn eine dieser Anwendungen solch eine Methode aufruft, wird der bösartige Code wiederbelebt. Java entschärft dieses Problem, indem elementare Datentypen (z.B. `String`) durch das Schlüsselwort `final` versiegelt werden.

- **Verletzung der Typintegrität durch beschädigte Objekte**:
  Ein Thread könnte ein Objekt sperren, anschließend darauf eine atomare Operation durchführen und währenddessen beendet werden. Dadurch wäre die Sperrung aufgehoben und das Objekt könnte in einem beschädigten Zustand hinterlassen werden, was dessen Typintegrität verletzten würde.

Selbst wenn ein sicherer Mechanismus existieren würde, läge auch hier ein Problem vor, falls der zu beendende Prozess Referenzen von ihm erzeugter Objekte anderen Prozessen übergeben hätte und diese Verbindungen weiterhin bestünden. In diesem Fall wäre eine vollständige Rückgewinnung des Speichers nicht möglich, was ein unnatürliches Verhalten darstellt und weitere negative Auswirkungen haben könnte.

3.6 Konsequenzen

Die beschriebenen Probleme werfen Zweifel auf, ob die derzeitigen Mechanismen ausreichen, um für ausreichende Isolation bei der parallelen Ausführung von Anwendungen und Transaktionen zu sorgen.

Die EJB-Spezifikation schränkt asoziales Verhalten ein, die oben genannten Probleme kann sie aber nur unzureichend lösen. Die Probleme in Zusammenhang mit statischen Feldern und synchronisierten Klassenmethoden, internalisierten Strings, Finalisierern und plattformabhängigem Code sowie die Nachteile der Replikation von Anwendungsklassen und dem nicht vorhandenen Ressourcenmanagement sind nicht von der Hand zu weisen. Die praktische Konsequenz dieser Probleme ist, dass auch J2EE-Server mit mehreren Prozessoren üblicherweise nur eine Anwendung ausführen, so dass prozessbasierte Mechanismen des zugrunde liegenden Betriebssystems benutzt werden können, um den Anforderungen an die Isolation und das Ressourcenmanagement nachzukommen [JDC+04]. Aber auch wenn nur eine Anwendung ausgeführt wird, bestehen die gleichen Probleme bei der parallelen Ausführung von Transaktionen dieser Anwendung.

Es ist anzunehmen, dass die seit 1998 verfügbare EJB-Technologie relative sichere Transaktionsverarbeitung ermöglicht. In wieweit die beschriebenen Probleme tatsächlich zur Beeinflussung parallel ablaufender Anwendungen und Transaktionen in derselben Virtual Machine führen können, wird im nächsten Kapitel untersucht.

Kapitel 4

Praktische Überprüfung der Probleme

4.1 Vorgehensweise

In diesem Kapitel werden die im letzten Kapitel aufgezeigten Probleme praktisch durch kurze Testprogramme nachvollzogen und deren Relevanz diskutiert, wobei sie auch immer im Kontext transaktionaler Objekte im Rahmen der Java 2 Enterprise Edition betrachtet werden.

4.2 Statische Felder

Die Verwendung mehrerer Class Loader löst die in Abschnitt 3.2.5 beschriebenen Probleme, um für jede in der Virtual Machine ausgeführte Anwendung einen separaten statischen Zustand zu erzeugen. Bei WebSphere ist zu beachten, dass die Verwendung mehrerer Class Loader deaktiviert werden kann, indem als Laderegel für Anwendungsklassen `SINGLE` eingestellt wird. Die Voreinstellung ist jedoch `MULTIPLE` und sollte nicht verändert werden.

Parallel ablaufende Transaktionen derselben Anwendung könnten aber immer noch durch statische Variablen der Anwendungsklassen kommunizieren. Die EJB-Spezifikation jedoch verbietet das Lesen und Schreiben nicht-konstanter Felder, gibt aber eine andere Begründung als die Isolation von Transaktionen an:

> An enterprise bean must not use read/write static fields. Using read-only static fields is allowed. Therefore it is recommended that all static fields in the enterprise bean class be declared as final.
>
> (This rule is required to ensure consistent runtime semantics because while some EJB containers may use a single JVM to execute all enterprise bean's instances, others may distribute the instances across multiple JVMs.)

WebSphere forciert dieses Verbot jedoch nicht, weder beim Deployment einer EJB-Anwendung noch zur Laufzeit. Eine fehlerhaft programmierte Anwendung, die von statischen Feldern in Anwendungsklassen Gebrauch macht, könnte also die Integrität von Transaktionen verletzen.

4.3 Statische Felder von Systemklassen

Wie in Abschnitt 3.2.7 erläutert, bleiben bei der Verwendung mehrerer Class Loader statische Felder und Klassenmethoden von Systemklassen unangetastet.

In den `java.*`-Paketen der Java 2 Platform, Standard Edition 5.0 gibt es in vielen Klassen statische Felder. Bei einem Großteil handelt es sich um öffentliche, konstante Felder primitiven Typs, die hinsichtlich der Isolation keine Gefahrenquelle darstellen. Problematisch aber können öffentliche, konstante Felder nicht-primitiven Typs sein, z.B. die Eingabe-, Ausgabe- und Fehlerströme der Klasse `System`. Hier könnte eine Anwendung durch den Aufruf von `System.out.close()` nachfolgende Ausgaben einer anderen Anwendungen verhindern.

Weitere ungewollte Interaktionen zwischen Anwendungen könnten durch statische Felder stattfinden, die zwar von außen nicht zugänglich sind, der Zugriff aber durch Klassenmethoden ermöglicht wird. Bei den Klassen innerhalb der `java.*`-Pakete in Tabelle 4.1 ist dies der Fall. Die Klassen des `java.beans`-Pakets wurden nicht berücksichtigt, da deren Klassenmethoden nur in Zusammenhang mit Entwicklungswerkzeugen Sinn machen.

Paket/Klasse	Zugriffsmethoden
`java.lang.System`	`getProperties`/`setProperties`
	`clearProperty*`/`getProperty`/
	`setProperty`
`java.lang.Thread`	`getDefaultUncaughtExceptionHandler*`
	`setDefaultUncaughtExceptionHandler*`
`java.net.Authenticator`	`requestPasswordAuthentication`/
	`setDefault`
`java.net.CookieHandler*`	`getDefault`/`setDefault`
`java.net.HttpURLConnection`	`getFollowRedirects`/
	`setFollowRedirects`
`java.net.ProxySelector*`	`getDefault`/`setDefault`
`java.net.ResponseCache*`	`getDefault`/`setDefault`
`java.net.URLConnection`	`getDefaultAllowUserInteraction`/
	`setDefaultAllowUserInteraction`
	`getFileNameMap`/`setFileNameMap`
`java.rmi.server.RemoteServer`	`getLog`/`setLog`
`java.rmi.server.RMISocketFactory`	`getFailureHandler`/`setFailureHandler`
`java.security.Policy`	`getPolicy`/`setPolicy`
`java.security.Security`	`getProperty`/`setProperty`
`java.sql.DriverManager`	`getLoginTimeout`/`setLoginTimeout`
	`getLogWriter`/`setLogWriter`
`java.util.Locale`	`getDefault`/`setDefault`
`java.util.TimeZone`	`getDefault`/`setDefault`

Tabelle 4.1: Klassen mit statischen Zugriffsmethoden (wiederverwendbar)

Bei manchen Klassen ist eine Wertzuweisung statischer Feldern durch Klassenmethoden sogar nur einmal möglich (siehe Tabelle 4.2). Falls also ein Prozess einem derartigen Feld einen Wert zugewiesen hat, führen weitere Zuweisungsversuche anderer Anwendungen, die in derselben Virtual Machine ausgeführt werden, zu Ausnahmen.

Paket/Klasse	Zugriffsmethoden
java.net.ServerSocket	setSocketFactory
java.net.Socket	setSocketImplFactory
java.net.URL	setURLStreamHandlerFactory
java.net.URLConnection	setContentHandlerFactory
java.rmi.ActivationGroup	setSystem
java.rmi.server.RMISocketFactory	getSocketFactory/setSocketFactory

Tabelle 4.2: Klassen mit statischen Zugriffsmethoden (nicht wiederverwendbar)

Es ist einleuchtend, dass die korrespondierenden Felder dieser Klassenmethoden Löcher im Grenzzaun der Anwendungen darstellen. Um an dieser Stelle vollständige Isolation zu gewährleisten, müssten auch unsichere statische Felder und Klassenmethoden von Systemklassen für jede Anwendung repliziert werden.

Bei herkömmlichen Anwendungen sind gemeinsam verwendete statische Felder von Systemklassen nicht akzeptabel. Beispielsweise könnten Anwendungen unterschiedliche Socket Factorys verwenden, was innerhalb derselben Virtual Machine nicht möglich ist. Es stellt sich die Frage, wie ernst diese Problematik im Umfeld von Enterprise JavaBeans ist.

Abgesehen davon, ob die Benutzung der Methoden in den obigen Tabellen für Enterprise Beans Sinn macht, verbietet die EJB-Spezifikation die Benutzung einiger der aufgelisteten Methoden:

> The enterprise bean must not attempt to set the socket factory used by `ServerSocket`, `Socket`, or the stream handler factory used by `URL`.
>
> (These networking functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security and decrease the container's ability to properly manage the runtime environment.)

Des Weiteren ist der Aufruf der in blau hervorgehobenen Methoden abhängig vom Sicherheitsmanager. Der Aufruf dieser Methoden innerhalb einer Session Bean unter WebSphere mit aktivierter Java 2-Sicherheit wurde bei allen zurückgewiesen. Die am Ende mit einem Stern versehenen Methoden stehen J2EE-Servern derzeit nicht zur Verfügung, da sie in der Java 2 Platform, Standard Edition 5.0 eingeführt wurden, auf der aber erst die kommende Version 1.5 der Java 2 Platform, Enterprise Edition aufbauen wird.

Der Sicherheitsmanager von WebSphere ist also zumindest in diesem Bereich auf maximale Sicherheit eingestellt. Rechte können nur explizit durch Erstellung von Richtlinien vergeben werden. Standardmäßig ist die Java 2-Sicherheit jedoch inaktiviert. Dazu heißt es in der WebSphere-Dokumentation [IBM04]:

> Since Java 2 security is relatively new, many existing or even new applications might not be prepared for the very fine-grain access control programming model that Java 2 security is capable of enforcing. Administrators should understand the possible consequences of enabling Java 2 security if applications are not prepared for Java 2 security. Java 2 security places some new requirements on application developers and administrators. [...] Java 2 security is a programming model that is very pervasive and has a huge impact on application development. It is disabled by default, but is enabled automatically when global security is enabled. [...] However, it does provide an extra level of access control protection on top of the J2EE role-based authorization. It particularly addresses the protection of system resources and APIs. Administrators should need to consider the benefits against the risks of disabling Java 2 Security.

Auch in Bezug auf die Isolation sollte die Java 2-Sicherheit also auf jeden Fall aktiviert sein. Nicht nur, um den Zugriff auf Klassenmethoden von Systemklassen einzuschränken, sondern auch um vor Anwendungen zu schützen, die destruktive Methoden aufrufen: Beispielsweise führt bei deaktivierter Java 2-Sicherheit der Aufruf von `Runtime.exit(int)`, bzw. `System.exit(int)` zur Beendigung des Applikationsservers und damit auch zur Beendigung aller laufenden Transaktionen — auch wenn die EJB-Spezifikation Enterprise Beans die Beendigung der Virtual Machine untersagt.

Bei aktivierter Java 2-Sicherheit scheinen statische Felder von Systemklassen zu einer gegenseitigen Beeinflussung von Anwendungen nicht beizutragen, da der Zugriff vom Sicherheitsmanager verweigert wird, sofern nicht entsprechende Richtlinien eingerichtet wurden. Die in den Tabellen aufgelisteten Klassenmethoden innerhalb der `java.*`-Pakete, die keine Sicherheitsüberprüfung vornehmen, dürften unkritisch sein. Um zu einem endgültigen Ergebnis zu kommen, müssten aber die statischen Felder aller Klassen der Java 2 Platform, Enterprise Edition und deren Semantik untersucht werden, was den Rahmen dieser Arbeit gesprengt hätte.

4.4 Synchronisierte Klassenmethoden von Systemklassen

In Abschnitt 3.2.7 wurden auch Probleme in Zusammenhang mit synchronisierten Klassenmethoden beschrieben, wenn der aufrufende Thread einer solchen Methode durch einen anderen ausgesetzt wird. In Abschnitt 3.5 wurde jedoch darauf hingewiesen, dass von der Benutzung des Threadprimitivs `Thread.suspend()` unter anderem aus diesem Grund abgeraten wird und heutzutage nicht mehr benutzt werden sollte.

Zusätzlich verbietet die EJB-Spezifikation Enterprise Beans, Threads überhaupt zu verwenden:

> The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread, or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.
>
> (These functions are reserved for the EJB container. Allowing the enterprise bean to manage threads would decrease the container's ability to properly manage the runtime environment.)

Wie schon bei statischen Variablen überprüft aber auch hier WebSphere weder statisch noch dynamisch, ob Enterprise Beans von Threads Gebrauch machen: Im Test gelang es mit wenigen Zeilen Code, Threads und Deadlocks zu erzeugen. Instanzen einer fehlerhaft programmierten Session Bean, die nicht der Spezifikation entspricht, könnten sich auf diese Weise gegenseitig blockieren (siehe Listing 4.1 und Listing 4.2).

Listing 4.1 zeigt eine von `Thread` abgeleitete Klasse, die eine synchronisierte Klassenmethode namens `waitTwoSeconds` enthält. Bei Aufruf dieser Methode wartet der aufrufende Thread zwei Sekunden, bevor er seine Ausführung fortsetzt. Wird ein Thread als Instanz dieser Klasse erzeugt, ruft er genau diese Methode auf, was an sich unproblematisch ist — zwei Sekunden nach dem Start endet der Thread wieder.

```
public class TestThread extends Thread {

    public static synchronized void waitTwoSeconds() {
        try {
            Thread.sleep(2000);
        }
        catch (InterruptedException e) {}
    }

    public void run() {
        TestThread.waitTwoSeconds();
    }
}
```

Listing 4.1: TestThread.java

Listing 4.2 zeigt ein Codefragment einer Session Bean, die verbotenerweise einen Thread der Klasse `TestThread` erzeugt, diesen startet und seine Ausführung nach einer Sekunde aussetzt. Da der Thread innerhalb einer synchronisierten Klassenmethode ausgesetzt wird, kann kein anderer Thread die Methode verwenden und muss so lange warten, bis der ausgesetzte Thread wiederaufgenommen wird. Dieses Problem kann im obigen Beispiel nur bei Session Beans derselben Anwendung auftreten, bei Aufruf einer synchronisierten Klassenmethode einer Systemklasse, könnte so auch eine von einem anderen Class Loader geladene Anwendung in Mitleidenschaft gezogen werden.

```
...

TestThread thread = new TestThread();
thread.start();

try {
    Thread.sleep(1000);
}
catch (InterruptedException e) {}

thread.suspend();

TestThread.waitTwoSeconds();

...
```

Listing 4.2: Session Bean (Auszug)

4.5 Internalisierte Strings

Zeichenketten werden in Java durch Instanzen der Klasse `String` repräsentiert. Strings können mit der `equals`-Methode auf Gleichheit überprüft werden.

Ein internalisierter String entsteht durch Aufruf der `intern()`-Methode eines Strings. Dabei wird ein neuer String gleichen Inhalts erzeugt und einer allen Anwendungen gemeinsamen Datenbasis hinzugefügt, falls dieser nicht bereits dort enthalten ist. Andernfalls wird lediglich eine Referenz auf den bereits internalisierten String zurückgegeben. Daraus folgt für alle Strings `s` und `t`, dass `s.intern() == t.intern()` nur dann `true` zurückliefert, falls `s.equals(t)` `true` zurückliefert.

Der Vorteil bei internalisierten Strings ist, das sie mit dem `==`-Operator schneller auf Gleichheit überprüft werden können.

Werden internalisierte Strings, wie in Abschnitt 3.2.8 erläutert, als Sperrobjekte in `synchronized`-Abschnitten verschiedener Anwendungen innerhalb derselben Virtual Machine benutzt, so könnte eine Anwendung eine andere von der Ausführung abhalten, falls identische Zeichenketten verwendet werden.

Listing 4.3 zeigt eine von `Thread` abgeleitete Klasse. Wird ein Thread als Instanz dieser Klasse gestartet, so internalisiert er den bei seiner Erzeugung übergebenen String und sperrt ihn für eine zufällige Zeitdauer zwischen 0 und 1000 Millisekunden.

```
public class StringInternThread extends Thread {

    private String lock;
    private int id;

    private static int counter = 1;

    StringInternThread(String lock) {
        this.lock = lock;

        id = counter++;
    }

    public void run() {
        java.util.Random random = new java.util.Random(id);

        while (true)
        {
            synchronized(lock.intern()) {
                System.out.println(id);

                try {
                    Thread.sleep(random.nextInt(1000));
                }
                catch (InterruptedException e) {}
            }
        }
    }

    public static void main(String[] args) {...}
}
```

Listing 4.3: StringInternThread.java

Listing 4.4 zeigt die `main`-Methode des Testprogramms. Dort werden drei Threads erzeugt und jeweils ein anderer String übergeben.

```
    public static void main(String[] args) {
        new StringInternThread(new String("lock1")).start();
        new StringInternThread(new String("lock2")).start();
        new StringInternThread(new String("lock3")).start();
    }
```

Listing 4.4: StringInternThread.java (main-Methode)

Das Ergebnis der Ausgabe ist zufällig, die Threads beeinflussen sich nicht gegenseitig:

```
1 2 3 2 2 2 2 3 2 1 2 3 1 3 2 ...
```

In Listing 4.5 ist die `main`-Methode des modifizierten Testprogramms abgebildet. Hier werden zwar auch jeweils unterschiedliche Instanzen der Klasse String übergeben, bei der anschließenden Internalisierung wird aber immer dieselbe Referenz zurückgeliefert. Die Strings werden mit `new` erzeugt, um unterschiedliche Objekte zu erhalten, da konstante Zeichenketten automatisch internalisiert werden. Da nun alle drei Threads dasselbe Sperrobjekt verwenden, kann sich zu jedem Zeitpunkt nur noch ein Thread innerhalb des `synchronized`-Abschnitts aufhalten. Die Reihenfolge, in der die Threads den Abschnitt durchlaufen, entspricht nun dem ihrer Erzeugung.

```
    public static void main(String[] args) {
        new StringInternThread(new String("lock")).start();
        new StringInternThread(new String("lock")).start();
        new StringInternThread(new String("lock")).start();
    }
```

Listing 4.5: StringInternThread.java (main-Methode)

Die Ausgabe liefert daher folgendes Ergebnis:

```
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 ...
```

Dieses Verhalten könnte zu unerwarteter gegenseitiger Beeinflussung von Anwendungen führen, auch wenn internalisierte Strings wahrscheinlich selten als Sperrobjekte benutzt werden. Um eine vollständige Isolation zu erreichen, müsste eine über ein Prozessmodell verfügende Virtual Machine für jeden Prozess eine eigene Datenbasis für internalisierte Strings unterhalten.

Bei Enterprise JavaBeans wird das Problem auch hier durch die EJB-Spezifikation indirekt gelöst, indem die Verwendung von Primitiva zur Synchronisation untersagt wird:

> An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.
>
> (Synchronization would not work if the EJB container distributed enterprise bean's instances across multiple JVMs.)

Es ist auffallend, dass auch dieser Passus ein ganz anderes Ziel als die Isolation verfolgt, aber auch in diesem Fall indirekt dazu beiträgt. Allerdings hat auch diesmal WebSphere nichts an Session Beans auszusetzen, die diesem Verbot nicht Folge leisten.

4.6 Finalisierung und Ereignisverarbeitung

Finalisierer und Ereignisse des AWT werden jeweils von einem eigenen Thread verarbeitet. Diese Verarbeitung findet sequentiell statt, was die Isolation von Anwendungen gefährden kann.

4.6.1 Finalisierung

Finalisierer werden in Java durch die `finalize`-Methode der Klasse `Object` repräsentiert. Sie enthält keine Logik, kann aber in abgeleiteten Klassen überschrieben werden. Diese Methode wird aufgerufen, bevor die Speicherbereinigung den Speicher des Objekts freigibt.

Finalisierer können dazu benutzt werden, Ressourcen freizugeben, deren Rückgewinnung mehr als die Freigabe des belegten Speichers erfordert, beispielsweise die Beendigung einer Datenbankverbindung. Die Java-Spezifikation garantiert aber nicht, wann, in welcher Reihenfolge und ob Finalisierer überhaupt aufgerufen werden.

Ein Problem tritt auf, falls die Ausführung eines Finalisierers nicht endet, z.B. aufgrund einer Endlosschleife. In diesem Fall können weder andere Finalisierer aufgerufen noch der Speicher der zugehörigen Objekte freigegeben werden. Eine Anwendung könnte so indirekt die Ausführung der Finalisierer, bzw. die Freigabe von Ressourcen einer anderen Anwendung verhindern.

Listing 4.6 zeigt die Klasse `ClassWithFinalizer`, deren Finalisierer eine Endlosschleife enthält.

```
public class ClassWithFinalizer {

    private int id;

    ClassWithFinalizer(int id)  {
        this.id = id;
    }

    protected void finalize() {
        while (true) {
            System.out.println(
                "Ausführung des Finalisierers von Objekt " + id);

            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {}
        }
    }

    public static void main(String[] args) {...}
}
```

Listing 4.6: ClassWithFinalizer.java

Listing 4.7 zeigt die `main`-Methode der Klasse `ClassWithFinalizer`. Darin werden drei Instanzen dieser Klasse erzeugt, die aber sofort nach ihrer Erzeugung nicht mehr erreichbar sind und somit der von ihnen belegte Speicher bei der nächsten Speicherbereinigung freigegeben wird. Deren Durchführung wird anschließend der Virtual Machine durch Aufruf von `System.gc()` vorgeschlagen. Im Experiment wurde die Speicherbereinigung dann auch tatsächlich ausgelöst. Die Endlosschleife am Schluss der Methode verhindert, dass die Virtual Machine beendet wird.

```
    public static void main(String[] args) {
        int counter = 0;

        while(++counter <= 3) {
            System.out.println(
                "Erzeugung von Objekt " + counter);

            new ClassWithFinalizer(counter);
        }

        System.gc();

        while (true) {
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {}
        }
    }
```

Listing 4.7: ClassWithFinalizer.java (main-Methode)

Die Ausgabe des Programms bestätigt, dass durch die Endlosschleife die Ausführung anderer Finalisierer nicht erfolgt:

```
Erzeugung von Objekt 1
Erzeugung von Objekt 2
Erzeugung von Objekt 3
Ausführung des Finalisierers von Objekt 3
Ausführung des Finalisierers von Objekt 3
Ausführung des Finalisierers von Objekt 3
...
```

Dieses Problem kann selbstverständlich auch bei J2EE-Servern eintreten. Es wäre denkbar, dass ein blockierter Finalisierer die Freigabe von Speicher verhindert und dadurch Transaktionen eventuell nicht mehr ausgeführt werden können.

4.6.2 Ereignisverarbeitung

Ein ähnliches Problem kann auch bei der Verarbeitung von Ereignissen des AWT auftreten. Auch hier werden eintreffende Ereignisse in eine Warteschlange eingereiht und sequentiell verarbeitet, d.h. die mit dem Ereignis verknüpfte Behandlungsroutine wird aufgerufen. Dabei wird die Verarbeitung weiterer Ereignisse erst fortgesetzt, wenn die vorherige Ereignisbehandlungsroutine zurückkehrt.

Gerät eine Ereignisbehandlungsroutine in eine Endlosschleife, werden keine Ereignisse mehr verarbeitet und die grafische Benutzeroberfläche kann folglich nicht mehr auf Eingaben des Benutzers reagieren. Diese Situation ist in Abbildung 4.1 dargestellt.

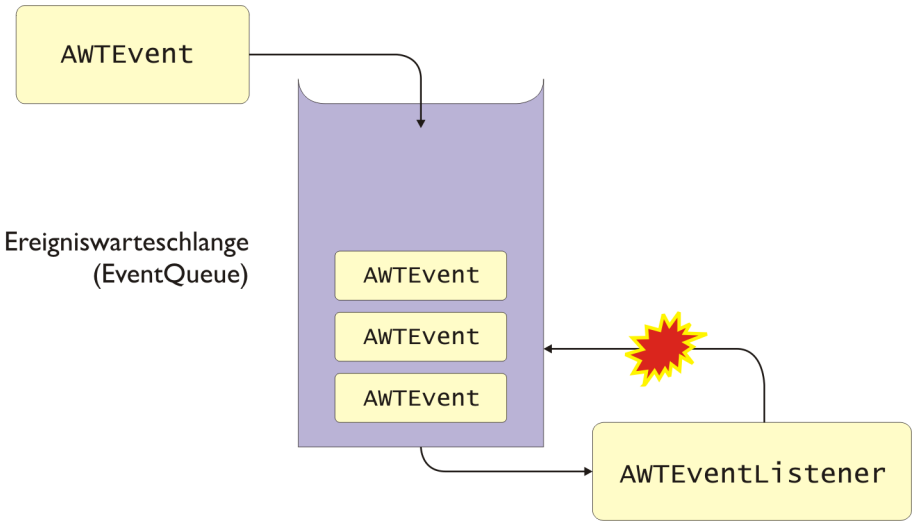

Abbildung 4.1: Ereignisverarbeitung unter Java

Listing 4.8 zeigt eine Java-Anwendung, die zwei Fenster mit jeweils einem Button erzeugt (siehe Abbildung 4.2).

```
import java.awt.event.*;
import javax.swing.*;

public class Frame extends javax.swing.JFrame {

    Frame(int x, int y) {
        setBounds(x, y, 200, 100);

        JButton jbutton = new JButton(" Diplomarbeit ");

        jbutton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                while (true) {
                    try {
                        Thread.sleep(1000);
                    }
                    catch (InterruptedException exception) {}
                }
            }
        });

        this.add(jbutton);
    }

    public static void main(String[] args) {
        (new Frame(0, 0)).setVisible(true);
        (new Frame(0, 100)).setVisible(true);
    }
}
```

Listing 4.8: Frame.java

Abbildung 4.2: Java-Anwendung mit Swing-GUI

Wird einer dieser Buttons gedrückt, gerät die zugehörige Ereignisbehandlungsroutine in eine Endlosschleife. Die grafische Benutzeroberfläche reagiert nicht mehr und wird auch nicht mehr aktualisiert (siehe Abbildung 4.3).

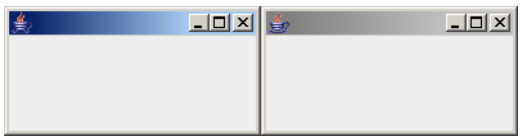

Abbildung 4.3: Folgen einer Endlosschleife in einer Ereignisbehandlungsroutine

Im Beispiel werden die Frames der Einfachheit halber innerhalb desselben Threads erzeugt. Aber auch wenn die Frames zu verschiedenen Anwendungen gehören, führt eine Endlosschleife zum gleichen Resultat. An dieser Stelle ist überhaupt keine Isolation vorhanden. Eine gemeinsam verwendete Ereigniswarteschlange ist bei parallel ausgeführten Anwendungen inakzeptabel. Selbst wenn keine Endlosschleife vorliegt, sondern die Ausführung einer Ereignisbehandlungsroutine einfach nur mehrere Sekunden dauert, sind grafische Benutzeroberflächen anderer Anwendungen während dieser Zeit vollständig blockiert.

Bei Enterprise JavaBeans ist dieser Punkt nicht relevant, da es bei ihnen keinen Sinn macht, eine grafische Benutzeroberfläche zu benutzen, was sich auch in der Spezifikation widerspiegelt:

> "An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.
>
> (Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.)"

4.7 Plattformabhängiger Code

In gewissen Situation ist es erforderlich, plattformabhängigen Code in Java-Anwendungen einzubinden. Verheerende Auswirkungen können dabei Bibliotheken haben, die in einer unsicheren Sprache geschrieben sind, beispielsweise in C oder C++. Greifen diese auf einen unzulässigen Speicherbereich zu, führt dies zum Absturz der Virtual Machine, da sie im selben Adressraum wie der Bibliotheksaufruf ausgeführt wird.

Listing 4.9 zeigt die Klasse `NativeCall`, die im statischen Initialisierer eine gleichnamige Bibliothek lädt. In der `main`-Methode wird dann die als extern deklarierte Methode `crash` der Bibliothek aufgerufen.

```
public class NativeCall {

    public static native void crash();

    static {
        System.loadLibrary("NativeCall");
    }

    public static void main(String[] args) {
        crash();
    }
}
```

Listing 4.9: NativeCall.java

Listing 4.10 zeigt die mit dem Hilfsprogramm `javah` aus `NativeCall.java` erstellte C(++) Header-Datei. Dort wird die in der Java-Anwendung benutzte Methode `crash` deklariert.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeCall */

#ifndef _Included_NativeCall
#define _Included_NativeCall
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     NativeCall
 * Method:    crash
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_NativeCall_crash
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 4.10: NativeCall.h

Die eigentlich Implementierung befindet sich in der Datei `NativeCall.cpp` (siehe Listing 4.11). Der Compiler erzeugt aus den beiden Dateien eine Windows-Bibliothek namens `NativeCall.dll`.

```
#include "stdafx.h"
#include "NativeCall.h"
#include <jni.h>

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    return TRUE;
}

JNIEXPORT void JNICALL Java_NativeCall_crash(JNIEnv *, jclass)
{
    *((unsigned void*)NULL) = 0;
}
```

Listing 4.11: NativeCall.cpp

Innerhalb der `crash`-Methode wird eine Zugriffsverletzung provoziert, indem versucht wird, an der Adresse des `NULL`-Zeigers eine Schreiboperation durchzuführen. Dieser Vorgang schlägt fehl und führt zum Absturz der Virtual Machine:

```
#
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  EXCEPTION_ACCESS_VIOLATION (0xc0000005) at pc=0x10001010, pid=1640, tid=3100
#
# Java VM: Java HotSpot(TM) Client VM (1.5.0-b64 mixed mode)
# Problematic frame:
# C  [NativeCall.dll+0x1010]
#
# An error report file with more information is saved as hs_err_pid1640.log
#
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
#
```

Eine Anwendung könnte so durch den Zugriff auf eine fehlerhafte Bibliothek alle in derselben Virtual Machine ausgeführten Anwendungen beenden.

Enterprise Beans scheinen auf den ersten Blick vor dieser potentiellen Gefahrenquelle geschützt zu sein, da die EJB-Spezifikation das Laden plattformabhängiger Bibliotheken verbietet:

> "An enterprise bean must not attempt to load a native library.
>
> (This function is reserved for the EJB container: Allowing the enterprise would create a security hole.)"

Unter WebSphere wird bei aktivierter Java 2-Sicherheit der Versuch, eine Bibliothek durch `System.loadLibrary` zu laden, mit einer Ausnahme zurückgewiesen.

Bei näherer Untersuchung der J2EE-Architektur stellt sich jedoch heraus, das im Rahmen der J2EE Connector Architecture der Zugriff auf plattformabhängige Bibliotheken zwangsweise erlaubt ist. Andernfalls wäre die Einbindung eines bestehenden und nicht auf Java basierenden Enterprise Information Systems (EIS) nicht möglich.

Tritt in einer solchen Bibliothek eine wie oben beschriebene Zugriffsverletzung auf, wird der gesamte Applikationsserver und alle laufenden Transaktionen in Mitleidenschaft gezogen. Dies stellt in der Tat ein schwerwiegendes Problem dar.

Eine derartige Situation wurde im Rahmen der Diplomarbeit simuliert. Die Grundlage hiefür stellte eine rudimentäre Implementierung eines Ressourcenadapters dar. Eine zum Adapter gehörige Klasse war dabei für das Laden einer plattformabhängigen Bibliothek zuständig und stellte eine Methode zu Verfügung, deren Aufruf eine Zugriffsverletzung innerhalb der Bibliothek erzeugte.

Der Aufruf dieser Methode führte wie beschrieben zur Beendigung des Applikationsservers.

4.8 Denial-of-Service-Angriffe

Eine große Schwäche von Java ist das fehlende Ressourcenmanagement. Eine fehlerhafte Anwendung könnte so viele Ressourcen für sich in Anspruch nehmen, dass die Ausführung anderer Anwendungen dadurch verhindert werden würde, z.B. weil nicht mehr genügend Speicher vorhanden ist. Aber auch Denial-of-Service-Angriffe, bei denen versucht wird, die von einem Server angebotenen Dienste durch Überlastung zum Erliegen zu bringen, sind denkbar.

Ein Angriff könnte beispielsweise darauf abzielen, große Speichermengen anzufordern, die im selben Augenblick schon nicht mehr erreichbar sind und sofort wieder freigegeben werden können. Sowohl durch die reine Speicheranforderung, als auch durch die anschließend ausgelöste Speicherbereinigung könnten die CPU-Ressourcen des Servers derart belastet werden, dass für die Ausführung anderer Anwendungen keine oder nur noch wenig Rechenzeit zur Verfügung steht.

Um auch in diesem Fall praktische Ergebnisse zu erzielen, wurde eine EJB-Anwendung sowie ein externer Java-Client entwickelt, um derartige Angriffe wenigstens ansatzweise zu simulieren (siehe Abbildung 4.4). Dabei startet der Client zwei Hintergrundthreads, die jeweils ununterbrochen Geld von einem Konto auf ein anderes überweisen. Parallel zur Transaktionsverarbeitung kann dann ein Denial-of-Service-Angriff durchgeführt werden. Bei einem solchen Angriff werden unter Einbeziehung mehrerer Parameter und mit Hilfe einer speziellen Session Bean große Mengen Speicher angefordert. Dieser Vorgang hat massiven Einfluss auf die Transaktionsverarbeitung.

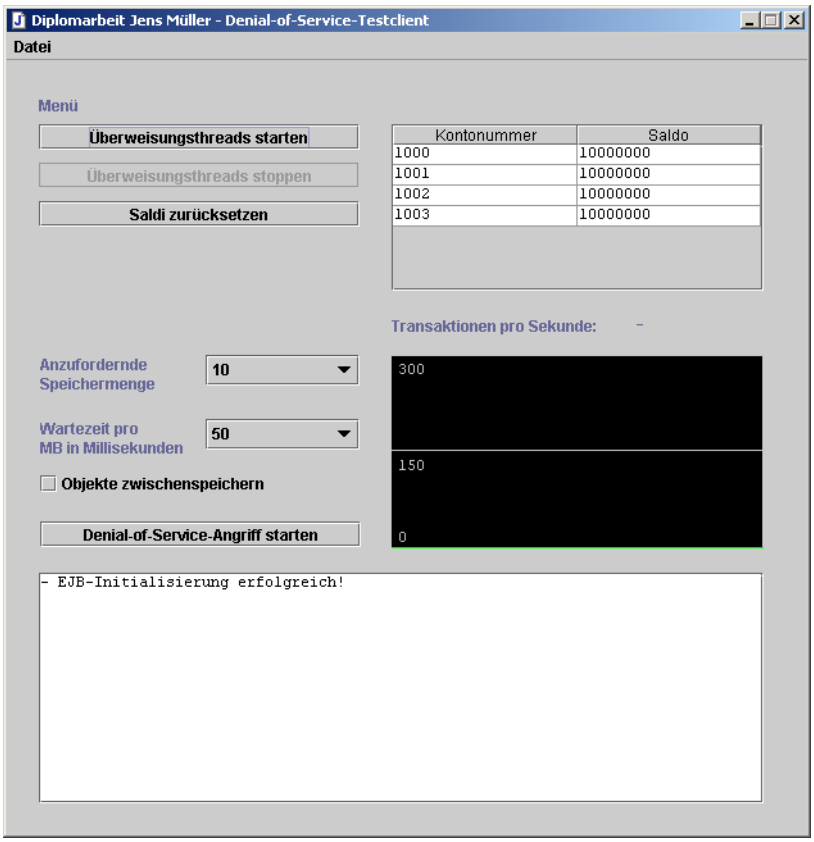

Abbildung 4.4: Der Denial-of-Service-Testclient

4.8.1 Testumgebung

Die Testumgebung der Simulation bestand aus einem Server und einem Client. Der Server war mit einer AMD Athlon XP 3000+ CPU (2,17 Gigahertz) und 512 Megabyte Arbeitsspeicher bestückt. Als Betriebssystem wurde Microsoft Windows Server 2003, Enterprise Edition mit Service Pack 1 verwendet. Auf diesem Rechner waren sowohl der WebSphere Application Server 6.0.1.0 als auch die DB2 Universal Database Enterprise Server Edition 8.2.2.0 von IBM installiert.

Der Client verfügte über eine Intel Pentium 4 Mobile CPU 1,6 Gigahertz und 1 Gigabyte Arbeitsspeicher. Hier war Microsoft Windows XP mit Service Pack 2 im Einsatz. Der Testclient wurde mit mit dem Rational Application Developer 6.0.0.1 (Interim Fix 001) von IBM entwickelt und auch innerhalb dieser Umgebung ausgeführt.

4.8.2 Entity und Session Beans

Die Unternehmensanwendung (siehe Anhang A.1) innerhalb des Applikationsservers bestand aus der Entity Bean `Account` (siehe Anhang A.1.2) und den Stateless Session Beans `Transfer` (siehe Anhang A.1.3) und `Test` (siehe Anhang A.1.4).

Die Entity Bean `Account` repräsentierte dabei ein Konto und bestand aus den Feldern `id` und `value` für Kontonummer und Saldo.

Die korrespondierende Datenbanktabelle in DB2 wurde mit folgenden Anweisungen erzeugt:

```
CREATE SCHEMA ADMINISTRATOR;

CREATE TABLE ADMINISTRATOR.ACCOUNT
  (ID INTEGER NOT NULL,
   VALUE INTEGER);

ALTER TABLE ADMINISTRATOR.ACCOUNT
  ADD CONSTRAINT PK_ACCOUNT PRIMARY KEY (ID);
```

Die Session Bean `Transfer` enthielt die von außen zugängliche Methode `transfer` zur Überweisung eines Geldbetrags:

```
public void transfer(
    int primaryKey1, int primaryKey2, int value);
```

Die Variablen `primaryKey1` und `primaryKey2` stehen dabei für Quell- und Zielkontonummer, `value` ist der zu überweisende Betrag.

Die Session Bean `Test` enthielt die von außen zugängliche Methode `allocateMemory`, die `megabytes` Megabytes anfordert, wobei nach jedem angeforderten Megabyte `millisecondsToWait` Millisekunden gewartet wird:

```
public void allocateMemory(
    int megabytes, int millisecondsToWait, boolean store);
```

Genau genommen werden dabei Instanzen der Klasse `Garbage` erstellt, die ein Array der Größe eines Megabytes als Instanzvariable enthält. Der Grund hierfür ist die Möglichkeit, den Zeitpunkt der Finalisierung dieser Objekte aufgrund einer Speicherbereinigung festzustellen, da in diesem Fall die `finalize`-Methode der Objekte aufgerufen wird. Die Wartezeit zwischen der Erzeugung von Objekten hat Auswirkungen auf die CPU-Zeit, die für die Transaktionsverarbeitung verbleibt.

Der Parameter `store` gibt an, ob die Instanzen der Klasse `Garbage` bis zum Ende des Vorgangs in einem Array zwischengespeichert werden sollen oder nicht. Der Unterschied ist sehr wichtig: Werden die Instanzen nicht zwischengespeichert, sind sie bereits nach ihrer Erzeugung nicht mehr erreichbar und der von ihnen benutzte Speicher kann bei der Speicherbereinigung freigegeben werden. Werden die Objekte aber bis zum Ende der Ausführung der Methode zwischengespeichert, so kann bei zu viel angefordertem Speicher eine Ausnahme vom Typ `OutOfMemoryError` auftreten. Deren Konsequenzen werden in einem späteren Abschnitt aufgezeigt.

4.8.3 Der Denial-of-Service-Testclient

Der Denial-of-Service-Testclient (siehe Anhang A.2) ist eine Java-Anwendung, die eine Swing-GUI (Graphical User Interface) verwendet. Beim Start stellt der Testclient eine Verbindung zum Applikationsserver her.

Die linke obere Hälfte des Programms enthält die Interaktionselemente. Die eingangs erwähnten Hintergrundthreads können mit den ersten beiden Buttons gestartet, bzw. gestoppt werden. Der erste Thread nimmt permanent Überweisungen von Konto 1000 auf Konto 1002 vor, der zweite von Konto 1001 auf Konto 1003. Die Kontostände werden in der Tabelle rechts oben angezeigt und alle fünf Sekunden aktualisiert. Der dritte Button setzt die Kontostände zurück. Der Transaktionsdurchsatz wird auf der rechten Seite angezeigt und sein Verlauf grafisch dargestellt.

Der vierte Button dient zur Ausführung eines Denial-of-Service-Angriffs. Dessen Parameter werden durch die Werte der zwei Kombinationsfelder und des Auswahlkastens bestimmt. Abbildung 4.5 zeigt den Testclient in Betrieb.

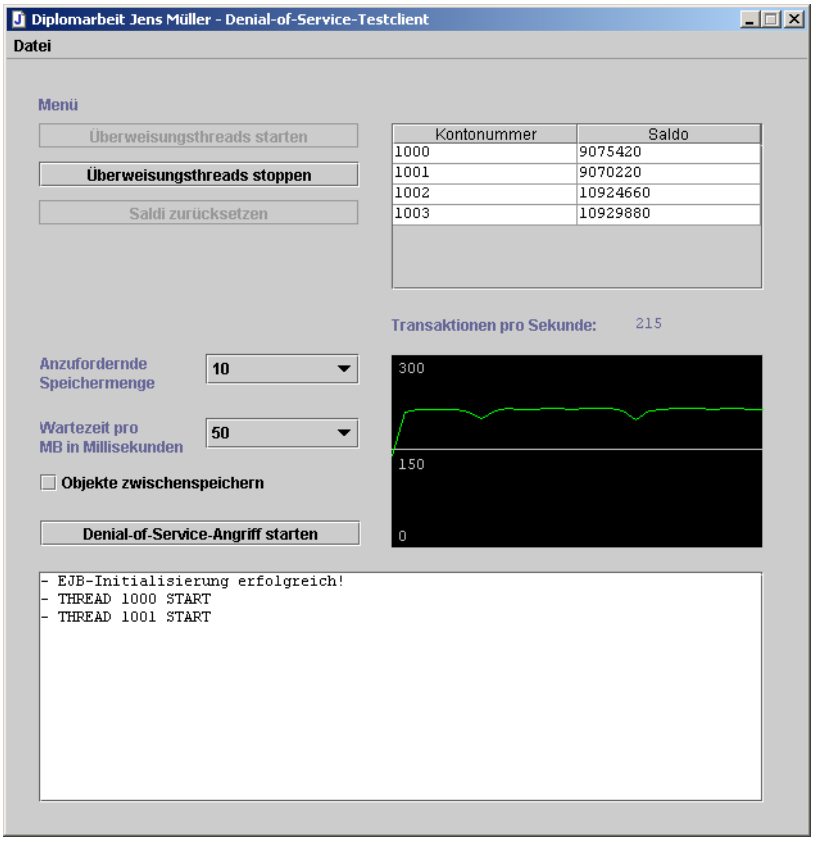

Abbildung 4.5: Der Denial-of-Service-Testclient in Betrieb

Auf dem Testsystem betrug der durchschnittliche Transaktionsdurchsatz um die 200 Transaktionen pro Sekunde. Im Lauf der Simulation wurden nacheinander Denial-of-Service-Angriffe mit unterschiedlichen Parametern durchgeführt. Die Ergebnisse werden in den nächsten drei Abschnitten geschildert.

4.8.4 Denial-of-Service-Angriff 1

Beim ersten Angriff wurde eine anzufordernde Speichermenge von 200 Megabyte gewählt, bei einer Wartezeit von 50 Millisekunden nach jedem angeforderten Megabyte.

Wie in Abbildung 4.6 zu sehen, resultierten diese Parameter in einem mittelschweren Angriff, der den Transaktionsdurchsatz ungefähr um zwei Drittel reduzierte.

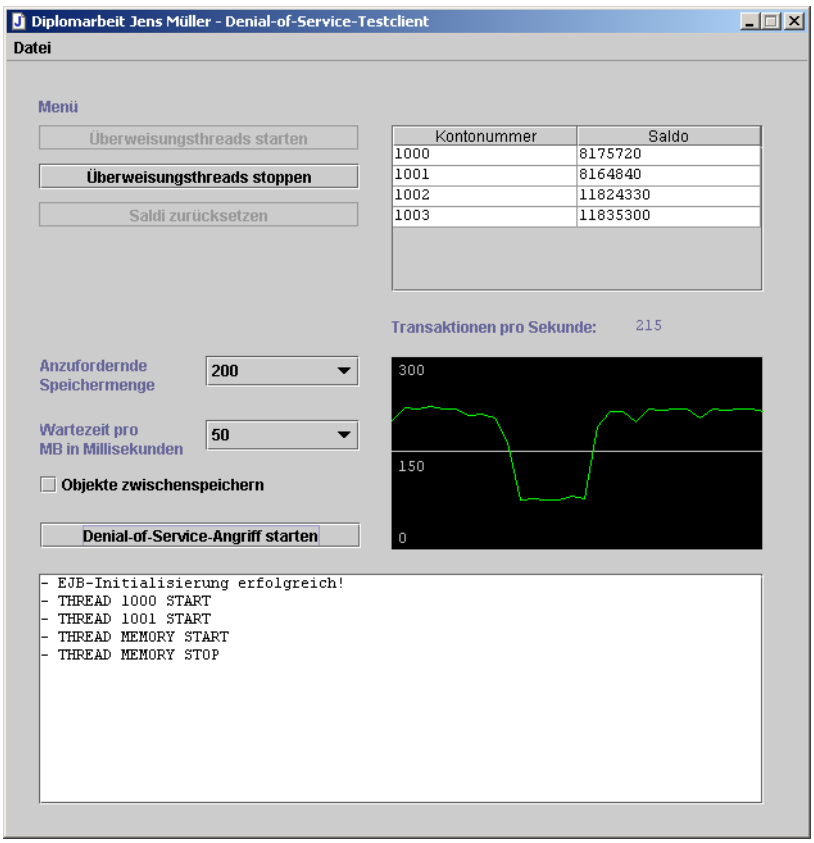

Abbildung 4.6: Ein mittelschwerer Denial-of-Service-Angriff

Da die erzeugten Objekte nicht zwischengespeichert wurden, trat zu keinem Zeitpunkt eine Situation ein, in der nicht genügend Speicher vorhanden war. Die Analyse der Logging-Daten ergab, dass die Speicherbereinigung während der Objekterzeugung mehrmals angestoßen wurde, was sicherlich zur Verminderung des Transaktionsdurchsatzes beitrug.

4.8.5 Denial-of-Service-Angriff 2

Beim zweiten Angriff wurde eine anzufordernde Speichermenge von 500 Megabyte ohne Wartezeit gewählt. Dies sollte zu einem massiv verringerten Transaktionsdurchsatz führen.

Abbildung 4.7 zeigt wie zu erwarten einen drastischen Einbruch des Transaktionsdurchsatzes, der beinahe zum Erliegen kommt.

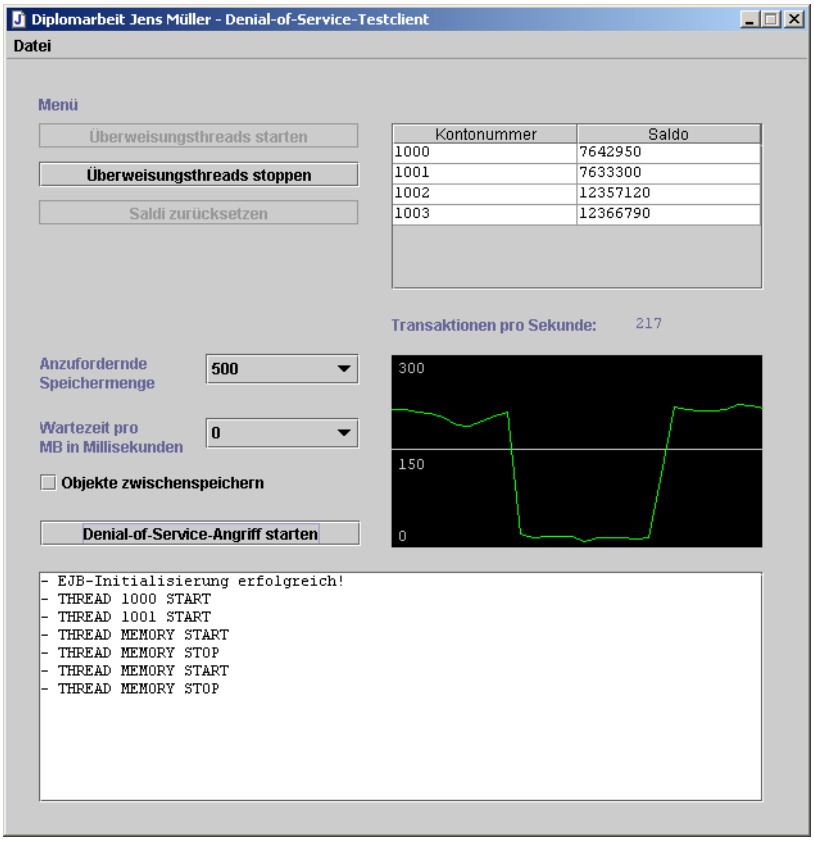

Abbildung 4.7: Ein schwerer Denial-of-Service-Angriff

Auch bei diesem Angriff war jederzeit genügend Speicher vorhanden, da die Speicherbereinigung permanent durchgeführt wurde. Da zwischen den einzelnen Objekterzeugungen keine Wartezeit verging, blieb der Transaktionsverarbeitung praktisch keine Rechenzeit mehr zur Verfügung. Solch ein Angriff könnte beliebig lange durchgeführt werden, um die angebotenen Dienste des Servers dauerhaft zu blockieren.

4.8.6 Denial-of-Service-Angriff 3

Der letzte Angriff wurde lediglich mit einer Speichermenge von 100 Megabyte ohne Wartezeit durchgeführt. Dafür wurde jedoch die Zwischenspeicherung der erzeugten Objekte verlangt.

In Abbildung 4.8 ist eine entsprechend kurze Unterbrechung der Transaktionsverarbeitung zu sehen, dafür sinkt der Durchsatz jedoch kurzzeitig auf 0.

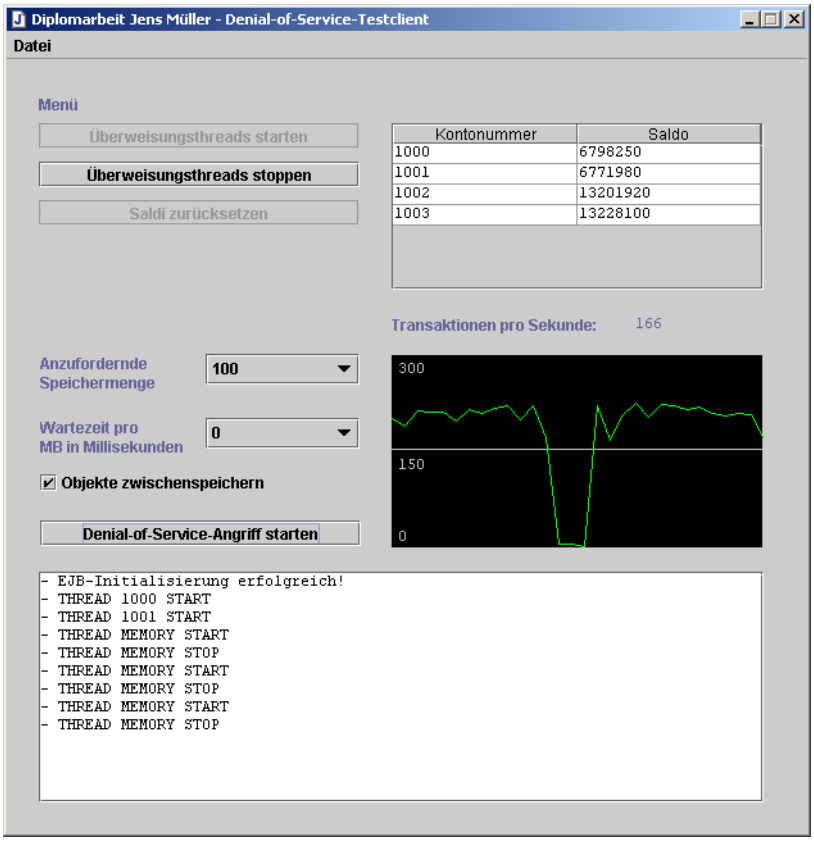

Abbildung 4.8: Ein kurzer aber schwerer Denial-of-Service-Angriff

Aufgrund ihrer Zwischenspeicherung konnten die erzeugten Objekte erst nach dem Angriff von der Speicherbereinigung freigegeben werden. Da die Grenze des für Enterprise Beans verfügbaren Speichers zuvor jedoch überschritten worden war, wurde eine Ausnahme vom Typ `OutOfMemoryError` vom Applikationsserver erzeugt. Darüber hinaus erstellte WebSphere zwei Dateien, die Diagnoseinformationen bezüglich der Virtual Machine und ein Abbild aller Objekte auf dem Heap und Referenzen zwischen diesen Objekten enthielten:

```
JVMDG217: Dump Handler is Processing a Signal - Please Wait.
JVMDG315: JVM Requesting Heap dump file
JVMDG318: Heap dump file written to (...)
JVMDG303: JVM Requesting Java core file
JVMDG304: Java core file written to (...)
JVMDG274: Dump Handler has Processed OutOfMemory.
```

Die Erzeugung dieser Dateien bei einem `OutOfMemoryError` ist bei WebSphere standardmäßig aktiviert, kann aber abgestellt werden. Dies ist in Produktivumgebungen sicherlich nicht der Fall, da durch sie wertvolle Informationen bei Fehlern gewonnen werden können.

Auch bei Wiederholung von nur kurzen Denial-of-Service-Angriffen kann jedoch die Erstellung dieser Dateien, die in diesem Fall zusammen eine Größe von ungefähr 15 Megabyte hatten, die Transaktionsverarbeitung vollständig unterbrechen.

4.9 Bewertung der Ergebnisse

Die Ergebnisse der in den letzten Abschnitten durchgeführten Untersuchungen können nicht generell bewertet werden, sondern sind im jeweiligen Kontext zu sehen. Beispielsweise spielen die Probleme in Zusammenhang mit plattformabhängigem Code bei einer EJB-Anwendung, bei der kein Zugriff auf Ressourcenadapter erfolgt und die in einem Applikationsserver ausgeführt wird, dessen Sicherheitsmanager das Laden von plattformabhängigen Bibliotheken verbietet, keine Rolle.

4.9.1 Isolation bei herkömmlichen Java-Anwendungen

Die Isolation herkömmlicher Java-Anwendungen innerhalb derselben Virtual Machine ist unzureichend. Die derzeitige Architektur von Java ist für die parallele Ausführung von Anwendungen nicht konzipiert worden. Die Probleme in Zusammenhang mit statischen Feldern und synchronisierten Klassenmethoden von Systemklassen, internalisierten Strings, systemweiten Ereignis- und Finalisierungswarteschlangen, plattformabhängigem Code und das fehlende Ressourcenmanagement sprechen eine deutliche Sprache. Diese Probleme sind besonders kritisch, wenn im Voraus nicht bekannt ist, welche Anwendungen parallel ausgeführt werden, deren Funktionalität also unbekannt ist. Die obigen Risiken werden natürlich vermindert, wenn nur bekannter Code ausgeführt wird und klar ist, dass keine Isolationsprobleme auftreten können. Bei zunehmender Komplexität der Software wird dies aber immer schwieriger.

4.9.2 Isolation bei EJB-Anwendungen

Die EJB-Spezifikation verringert den Handlungsspielraum von transaktionalen Objekten. Dadurch werden einige der bei herkömmlichen Java-Anwendungen möglichen Isolationsprobleme irrelevant. Zumindest WebSphere prüft jedoch weder beim Deployment noch zur Laufzeit, ob die Verbote der EJB-Spezifikation eingehalten werden. Eine fehlerhaft programmierte Enterprise Bean könnte so ohne Warnung die Isolation gefährden. Davon abgesehen lässt die parallele Ausführung von EJB-Anwendungen innerhalb desselben Applikationsservers dennoch zu wünschen übrig. Mittelmäßige Isolation, fehlendes Ressourcenmanagement und das Problem, EJB-Anwendungen nicht stabil beenden zu können, sind auch hier anzutreffen. Wie zuvor erwähnt wird daher meist nur eine EJB-Anwendung innerhalb eines Applikationsservers ausgeführt.

4.9.3 Isolation bei Transaktionen einer EJB-Anwendung

Die Probleme bei parallel ausgeführten EJB-Anwendungen lassen sich auf parallel ausgeführte Transaktionen derselben EJB-Anwendung übertragen. Im Unterschied zu parallel ausgeführten EJB-Anwendungen teilen Transaktionen derselben EJB-Anwendung aber auch den statischen Zustand der zugehörigen Anwendungsklassen. Dieser kann jedoch nur zum Problem werden, wenn die Regeln der EJB-Spezifikation nicht eingehalten werden, z.B. wenn eine Session Bean nicht-konstante, statische Felder enthält. So könnte eine Transaktionen den statischen Zustand verändern und eine nachfolgende oder parallel ausgeführte Transaktion beeinflussen.

Aber auch wenn dies nicht der Fall ist, kann ein fehlerhafter Ressourcenadapter sämtliche Transaktionen beenden oder eine Transaktion, die alle Ressourcen für sich in Anspruch nimmt, die Ausführung anderer Transaktionen verzögern oder sogar verhindern. Zwar handelt es sich um extreme Situationen, sie auszuschließen wäre jedoch fatal.

4.9.4 Fazit

Ein pauschales Urteil kann in Anbetracht der unterschiedlichen Sichtweisen nicht gefällt werden. Soll Java zukünftig die Rolle einer betriebssystemähnlichen Plattform spielen und mehrere Anwendungen innerhalb derselben Virtual Machine sicher ausführen können, so muss die Architektur grundlegend verändert werden. Da dies auch die Firma Sun vor einigen Jahren erkannt hat, arbeiten deren Forscher seit geraumer Zeit an einer Lösung, mit der sich unter anderem das vorletzte Kapitel dieser Arbeit befasst.

Enterprise JavaBeans bietet relativ sichere Transaktionsverarbeitung, sofern sich Programmierer an die Regeln der Spezifikation halten. WebSphere ist in dieser Beziehung nicht sehr hilfreich, da Enterprise Beans, die Verbote der Spezifikation missachten, nicht beanstandet werden. Andere Applikationsserver wurden nicht untersucht.

In vielen Umgebungen mag die gebotene Sicherheit ausreichend sein. Soll das Transaktionsverarbeitungssystem aber absolute Transaktionssicherheit bieten, so ist derzeit vom Einsatz der EJB-Technologie abzuraten. Beispielsweise sind Szenarien denkbar, die bei einem Fehler in einer plattformabhängigen Bibliothek fordern, dass nur die verantwortliche Transaktion in Mitleidenschaft gezogen werden darf, nicht jedoch der gesamte Applikationsserver. Bei der derzeitigen Architektur ist dies aber der Fall.

Das ist auch der Grund, warum Firmen wie IBM oder SAP sich dazu veranlasst sahen, eigene Lösungen zu entwickeln, die allerhöchste Sicherheitsanforderungen erfüllen. Unter anderem wird auch darüber in den nächsten zwei Kapiteln berichtet.

Kapitel 5

Lösungsansätze

„Good fences make good neighbors"
(aus Mending Wall von Robert Frost)

Die Probleme der Virtual Machine in Bezug auf die parallele Ausführung von Anwendungen und Transaktionen werden in zahlreichen Publikationen aufgegriffen, in denen bisweilen weitgehend vollständige Lösungen vorgeschlagen werden. Teilweise ist die daraus entstandene Software öffentlich verfügbar. Die Mechanismen dieser Lösungsvorschläge, die die Isolation von Anwendungen gewährleisten sollen, können dabei in folgende Kategorien eingeteilt werden: Isolation durch Class Loader, Isolation durch Bytecode-Transformation, Isolation durch Modifikation der Virtual Machine und Isolation durch Betriebssystemprozesse. Einige dieser Ansätze zielen auf eine betriebssystemähnliche Laufzeitumgebung ab, wobei teilweise auch Ressourcenmanagement möglich ist.

5.1 Isolation durch Class Loader

In einigen Veröffentlichungen werden Class Loader zur Isolation von Anwendungen vorgeschlagen, es werden jedoch meist Lösungen für das Problem statischer Felder von Systemklassen beschrieben.

5.1.1 Echidna

Ein Projekt, bei dem dies nicht der Fall ist, das aber der Vollständigkeit halber genannt werden soll, ist Echidna ([Gor99], [20]). Echidna ist eine Java-Bibliothek, um separate Anwendungen komfortabel innerhalb derselben Virtual Machine auszuführen und bis zu einem gewissen Grad zu kontrollieren. Ein Prozessmanager (siehe Abbildung 5.1) spielt dabei die zentrale Rolle.

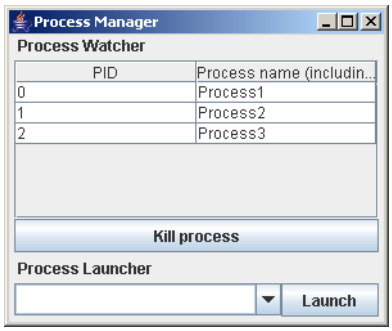

Abbildung 5.1: Der Prozessmanager von Echidna

5.1.2 Der Ansatz von Balfanz und Gong

Bereits 1997 untersuchten Balfanz und Gong die Virtual Machine auf Schwächen hinsichtlich der parallelen Ausführung von Anwendungen und veröffentlichten ihre Erkenntnisse samt einer modifizierten Virtual Machine in [BG97]. Die Isolation erfolgt aber dennoch auf Grundlage mehrerer Class Loader.

Die von Balfanz und Gong vorgenommenen Änderungen zielten darauf ab, Betriebssystemkonzepte auf Java zu übertragen:

- Definition von Anwendungen als Menge aus Threads, die durch eine Klasse namens `Application` repräsentiert werden. Weitere Eigenschaften dieser Definition sind unter anderem die Assoziation mit einem Benutzer sowie eigene Eingabe-, Ausgabe- und Fehlerströme.

- Einführung von Benutzern und Authentifizierung mit Hilfe eines Login-Programms.

- Einführung benutzerbasierter Zugriffskontrolle

- Einführung von anwendungsspezifischen Threads für die Ereignisverarbeitung

- Replikation bestimmter Systemklassen (z.B. `System`), um bei Anwendungen den Eindruck zu erwecken, sie würden allein in der Virtual Machine ausgeführt.

- Installation eines neuen Sicherheitsmanagers (System Security Manager), der primär die Aufgabe hat, Anwendungen vor anderen Anwendungen zu schützen.

5.1.3 J-Kernel

Eine wesentlich komplexere portable Java-Bibliothek, die Class Loader als Isolationsmechanismus verwendet, ist J-Kernel [21]. J-Kernel erweitert die zu Grunde liegende Virtual Machine um so genannte Multiple Protection Domains ([HCC+98], [ECC+99]) sowie Kanäle zur Kommunikation zwischen diesen Domains, ohne die Virtual Machine dabei zu modifizieren. Des Weiteren bietet J-Kernel die Möglichkeit des Widerrufs von Zugriffsrechten, die Sicherung von Threads, rudimentäres Ressourcenmanagement und die Möglichkeit, Protection Domains zu beenden.

Protection Domains werden in der Terminologie von J-Kernel Tasks genannt. Interprozesskommunikation und Zugriffskontrolle werden durch Capabilities realisiert. In [Lev84] werden diese wie folgt definiert: Konzeptuell ist eine Capability ein Token, Ticket oder Schlüssel, der dem Besitzer das Recht gibt, in einem Computersystem auf eine Entität oder ein Objekt zuzugreifen. Eine Capability wird als Datenstruktur implementiert, die zwei Felder beinhaltet: Einen eindeutigen Objektbezeichner und Zugriffsrechte.

Die Sicherheit der J-Kernel-Bibliothek basiert auf drei Kernkonzepten: Capabilities, Tasks und Task-übergreifenden Methodenaufrufen:

- **Capabilities:**
  Capabilities repräsentieren indirekte Referenzen auf Ressourcen anderer Tasks. Eine Capability kann jederzeit durch den erzeugenden Task wiederrufen werden. Die Verwendung einer widerrufenen Capability hat eine Ausnahme zur Folge.

- **Tasks:**
  Tasks sind Mengen aus Threads innerhalb eigener Klassennamensräume. Wenn ein Task endet, werden alle Capabilities, die er erzeugt hat, wiederrufen, so dass der von ihm benutzte Speicher freigegeben werden kann. Die in Abschnitt 3.3 und Abschnitt 3.5 beschriebenen Probleme werden so größtenteils gelöst.

- **Task-übergreifende Methodenaufrufe:**
  Task-übergreifende Methodenaufrufe werden ausgeführt, indem Methoden von Capabilities, die von anderen Tasks erzeugt wurden, aufgerufen werden. Bei deren Aufruf gilt eine spezielle Konvention: Argumente und Rückgabewerte werden per Referenz übergeben, sofern auch sie Capabilities sind. Falls es sich aber um primitive Datentypen oder Objekte handelt, die keine Capabilities sind, wird eine Kopie übergeben. Bei einem solchen Kopiervorgang wird diese Konvention rekursiv auf alle Felder des Objekts angewandt, so dass die gesamte transitive Hülle erreichbarer Felder berücksichtigt wird. Auf diese Weise sind Capabilities die einzigen Objekte, die von Tasks gemeinsam verwendet werden. Referenzen auf reguläre Objekte können nicht an andere Tasks übergeben werden.

Jeder Task verfügt über einen eigenen Class Loader (siehe Abschnitt 3.2.6). Ein Task kann durch Bereitstellung einer Capability vom Typ `SharedClass` eine geladene Klasse anderen Tasks zur Verfügung stellen, um sie gemeinsam verwenden zu können. Dies ist die Grundlage Task-übergreifender Kommunikation. Unsichere statische Felder sind in gemeinsam verwendeten Klassen verboten, wodurch eine gemeinsame Verwendung von Objekten, die keine Capabilities sind, durch diese statischen Felder unterbunden wird.

Wie bereits erwähnt wird bei Task-übergreifenden Methodenaufrufen eine Kopie aller Argumente erzeugt, die keine Capabilities sind. Dies geschieht standardmäßig durch Serialisierung, wie es auch bei Java Remote Method Invocation (Java RMI) [22] der Fall ist. Bei der Serialisierung werden Objekte sequentiell auf eine externe Darstellungsform (z.B. auf eine Datei) abgebildet. Später kann aus dieser externen Darstellungsform ein Objekt zurückgewonnen werden, dessen Zustand mit dem des ursprünglichen Objekts identisch ist. Zusätzlich stellt die J-Kernel-Bibliothek einen effizienteren Fast Copy Mechanismus zur Verfügung. Fast Copy Klassen müssen explizit als solche ausgewiesen werden.

J-Kernel bietet rudimentäres Ressourcenmanagement auf Basis der JRes-Schnittstelle ([CCH+98], [CE98]). JRes stellt Mechanismen zur Verfügung, die es ermöglichen, die für Threads verfügbaren Ressourcen zu begrenzen, wie etwa Speicher, CPU-Zeit und Netzwerkressourcen. Aufgrund der Tatsache, dass JRes eine Bibliothek und nicht Teil der Virtual Machine ist, können Ressourcen nur bis zu einem gewissen Grad kontrolliert werden. Beispielsweise ist es zwar möglich festzustellen, ob ein Thread mehr CPU-Zeit in Anspruch nimmt, als ihm zusteht und als Reaktion kann seine Priorität herabgesetzt werden. Ohne aber direkt in der Virtual Machine Änderungen vorzunehmen ist es nicht möglich, den Scheduling-Algorithmus zu verändern.

Der Einsatz der JRes-Schnittstelle führt zu geringen Leistungseinbußen, da sie Modifikationen am Bytecode geladener Klassen vornimmt. Diese könnten durch Integration in die Virtual Machine beseitigt werden.

5.2 Isolation durch Bytecode-Transformation

5.2.1 Der Safe Shared Class Loader

In [KP00] wird eine sichere Variante des in Abschnitt 3.2.5 vorgestellten Ansatzes beschrieben. Statische Felder werden dabei für jeden Prozess dupliziert, es wird jedoch nur ein Class Loader verwendet. Dadurch ergeben sich keine Nachteile in Folge replizierter Anwendungsklassen.

Beim Safe Shared Class Loader werden Klassen und Schnittstellen auf statische Felder untersucht und anschließend transformiert. Klassen, die statische Felder beinhalten, werden dabei in zwei neue Klassen aufgeteilt: Die NSP-Klasse (non-static part) enthält die nicht-statischen Felder und sämtliche Methoden. Die SP-Klasse (static part) enthält die statischen Felder, die aber in nicht-statische Felder umgewandelt werden. Zur Laufzeit wird für jeden Prozess eine Instanz der SP-Klasse erzeugt und diesem zugeordnet (siehe Abbildung 5.2). Dadurch ändert sich die Semantik statischer Felder und entspricht der bei Verwendung mehrerer Class Loader.

Jedes statische Feld in der ursprünglichen Klasse wird durch zwei Methoden für Lese- und Schreibzugriff in der NSP-Klasse ersetzt. Diese Zugriffsmethoden eruieren zunächst, welche Instanz der SP-Klasse dem aktuellen Prozess zugeordnet ist und lesen oder schreiben anschließend das korrespondierende Feld. Darüber hinaus wird eventuell vorhandene Funktionalität im statischen Initialisierer der ursprünglichen Klasse in den Konstruktor der SP-Klasse verlagert. Abschließend müssen aufgrund der Transformation alle Zugriffe auf statische Felder in allen Klassen durch korrespondierende Methodenaufrufe ersetzt werden. Die Umwandlung von Schnittstellen erfolgt auf ähnliche Weise.

Zur Interprozesskommunikation werden auch hier Capabilities vorgeschlagen, die aber nicht Bestandteil des Safe Shared Class Loaders sind. Der in Abschnitt 5.1.3 erwähnte Fast Copy Mechanismus kann bei allen Klassen benutzt werden.

Die Transformationen können direkt auf kompilierte Java-Klassen angewandt werden, der zugehörige Quellcode muss nicht vorhanden sein. Ob diese Transformationen dynamisch erfolgen können, geht aus der Veröffentlichung nicht hervor.

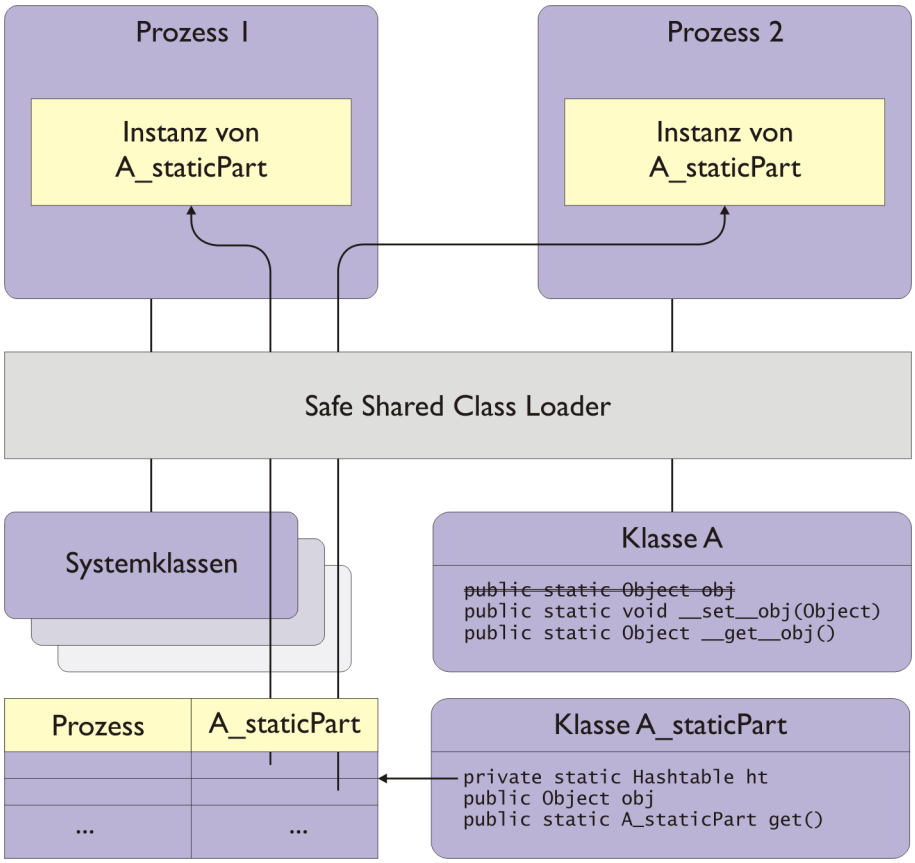

Abbildung 5.2: Multitasking unter Verwendung des Safe Shared Class Loaders

Bei Verwendung des Safe Shared Class Loaders treten nur in wenigen Spezialfällen Probleme auf, die durch Modifikation einiger Systemklassen beseitigt werden könnten. Der Vorteil dieses Ansatzes ist, das kein Bytecode durch mehrfaches Laden von Klassen repliziert wird. Lediglich statische Felder von Klassen werden für jeden Prozess dupliziert. Bei der parallelen Ausführung von Anwendungen wird dadurch nachweislich [KP00] weniger Hauptspeicher verbraucht und die Ausführung beschleunigt.

Im Vergleich zur J-Kernel-Bibliothek werden bei Verwendung des Safe Shared Class Loaders alle Klassen gemeinsam verwendet, sie müssen nicht erst als solche ausgewiesen werden. Außerdem dürfen sie statische Felder enthalten, was bei der J-Kernel-Bibliothek verboten ist. Die Problematik synchronisierter Klassenmethoden wird beim Safe Shared Class Loader nicht aufgegriffen.

Der Safe Shared Class Loader ist nicht öffentlich verfügbar und wird auch in der einschlägigen Literatur zu diesem Thema nicht erwähnt.

5.2.2 Der Ansatz von Czajkowski (1. Implementierung)

In [Cza00] wird ein Ansatz beschrieben, der dem Safe Shared Class Loader sehr ähnlich ist: Ein einziger Class Loader ist für das Laden von Anwendungsklassen, die gegebenenfalls transformiert werden, verantwortlich. Auch hier ist die grundlegende Idee, statische Felder zu extrahieren und transparent für jede Anwendung eine eigene Kopie zu erstellen. Des Weiteren werden auch synchronisierte Klassenmethoden berücksichtigt, um das in Abschnitt 3.2.7 erwähnte Problem zu beheben. Es werden zwei Prototypen vorgestellt: Eine Variante transformiert Bytecode statisch oder dynamisch, die andere ist eine modifizierte Virtual Machine.

Der Ansatz von Czajkowski unterscheidet sich nur geringfügig vom Safe Shared Class Loader: Klassen, die statische Felder oder synchronisierte Klassenmethoden enthalten, werden durch drei neue Klassen ersetzt.

Als Beispiel stelle man sich eine Klasse X vor, die statische Felder enthält. Sie wird in drei Klassen aufgeteilt:

- Die ursprüngliche Klasse, deren statische Felder aber entfernt wurden.
- Die Klasse X$sFields, die die in der ursprünglichen Klasse entfernten Felder enthält, allerdings als nicht-statische Felder.
- Die Klasse X$aMethods, die für jede Anwendung eine Instanz der Klasse X$sFields bereitstellt, sowie über Lese- und Zuweisungsmethoden für jedes Feld verfügt, das aus der ursprünglichen Klasse entfernt wurde. X$aMethods greift dabei anhand der Identität des aufrufenden Threads auf die korrekte Instanz zu.

Anschließend muss in der ursprünglichen Klasse jeder Zugriff auf statische Felder durch die entsprechenden Lese-, bzw. Zuweisungsmethoden der Klasse X$aMethods ersetzt werden.

Daraus resultiert ungeachtet der Anzahl an Anwendungen, die die Klasse X verwenden, nur eine Kopie der modifizierten Klasse X und der Klasse X$aMethods. Im Gegensatz zum Safe Shared Class Loader muss dabei nicht zwischen Klassen und Schnittstellen unterschieden werden. Da nur noch ein Class Loader verwendet wird, tritt das in Abschnitt 3.2.6 erläuterte Problem in Zusammenhang mit synchronisierten Klassenmethoden bei allen Klassen auf, wird aber durch entsprechende Umformungen gelöst.

Bei der gleichzeitigen Ausführung mehrerer Instanzen einer Anwendung werden auch bei diesem Ansatz verglichen mit der Verwendung mehrerer Class Loader Anwendungen schneller ausgeführt. Weiterhin wird wesentlich weniger Hauptspeicher verbraucht.

Sowohl der Safe Shared Class Loader als auch die erste Variante des Ansatzes von Czajkowski fokussieren die Isolation auf Objekt- und Methodenebene. Der Umgang mit plattformabhängigem Code und Betriebssystemkonzepte wie die Interprozesskommunikation, Ressourcenmanagement und die Beendigung von Anwendungen werden nicht behandelt.

5.3 Isolation durch Modifikation der Virtual Machine

5.3.1 Der Ansatz von Czajkowski (2. Implementierung)

Der zweite in [Cza00] vorgestellte Prototyp basiert auf der K Virtual Machine (KVM) [SM00]. Die KVM ist eine speziell für mobile Geräte mit geringem Speicher konstruierte Virtual Machine. Sie wurde als Versuchsobjekt gewählt, da die parallele Ausführung von Anwendungen auf mobilen Geräte damals aus zweierlei Gründen nicht möglich war: Zum einen konnte aufgrund mangelnden Arbeitsspeichers nicht mehr als eine Instanz der KVM gestartet werden, auf der anderen Seite unterstützte die KVM keine Class Loader.

Die modifizierte KVM analysiert geladene Klassen und repliziert alle statischen Felder entsprechend der maximalen Anzahl parallel ausführbarer Anwendungen. Ähnlich wird bei synchronisierten Klassenmethoden verfahren. Wenn eine Anwendung geladen wird, überprüft die modifizierte KVM zunächst, ob die maximale Anzahl parallel ausführbarer Anwendungen bereits erreicht ist. Falls dem nicht so ist, erhält sie eine eindeutige Identifikationsnummer.

Im Gegensatz zur ersten Implementierung handelt es sich bei diesem Prototyp um eine umfassendere Lösung: Anwendungen können angehalten und beendet werden, ohne andere Prozesse dadurch zu beeinflussen. Zusätzlich bietet die modifizierte KVM eine Schnittstelle zur Interprozesskommunikation und eine an JRes (siehe Abschnitt 5.1.3) angelehnte Schnittstelle zum Ressourcenmanagement.

5.3.2 K0 (GVM)

K0 (vormals GVM) und Alta ([BTS+98], [BTS+00], [TL98], [Tul99]) sind modifizierte Versionen von Kaffe [23], einer quelloffenen Virtual Machine für Java. Dabei steht das Ressourcenmanagement im Mittelpunkt. Beide Projekte wurden an der Universität von Utah durchgeführt.

Das Design von K0 ähnelt dem eines traditionellen monolithischen Kernels. Das Hauptziel von K0 ist die komplette Isolation von Ressourcen zwischen Prozessen sowie die gemeinsame Verwendung von Objekten.

Ein K0-Prozess besteht aus einem Klassennamensraum, einer Menge aus Threads und ist mit einem eigenen Heap assoziiert. Die Erzeugung getrennter Klassennamensräume wird auch bei K0 durch die Verwendung separater Class Loader erreicht. Klassen ohne unsichere statische Felder und andere Daten können von Prozessen gemeinsam verwendet werden, indem sie in einem speziellen Speicherbereich, auf den alle Prozesse Zugriff haben, abgelegt werden. Ein Objekt im Heap eines Prozesses kann ein gemeinsam verwendetes Objekt referenzieren und umgekehrt, direkte Referenzen auf Objekte in den Heaps anderer Prozesse sind aber verboten, um die Integrität von Prozessen nicht zu verletzen.

Darüber hinaus ist K0 in der Lage, den Speicherverbrauch von Prozessen umfassend zu kontrollieren und die für Prozesse und Threads verfügbare CPU-Zeit zu steuern. Dabei wird CPU Inheritance Scheduling [FS96] eingesetzt, das mit der von Threadgruppen erzeugten Hierarchie kombiniert wird: Threadgruppen innerhalb von Prozessen können die ihnen zustehende CPU-Zeit wiederum zwischen ihren Threads aufteilen.

Prinzipiell ist es bei K0 möglich, eine Speicherbereinigung des Heaps eines Prozesses unabhängig von den Heaps anderer Prozesse durchzuführen. Den Angaben der Entwickler zufolge unterstützte der anfängliche Prototyp aber weder die getrennte Speicherbereinigung noch die Speicherbereinigung von Klassen. Die Verwendung separater Heaps hat den zusätzlichen Vorteil der Vermeidung so genannter Priority Inversions: Wenn eine Speicherbereinigung durchgeführt werden soll, müssen Threads anderer Prozesse, die eine hohe Priorität haben, nicht unterbrochen werden.

Der Nachfolger von K0 wird im übernächsten Abschnitt behandelt.

5.3.3 Alta

Alta verwendet ein hierarchisches Prozessmodell, dessen Design auf dem des Mikrokernels Fluke [24] beruht, der ebenfalls an der Universität von Utah entwickelt wurde. Das Prozessmodell [FHL+96] von Fluke ermöglicht es einem Prozess, die Ressourcen seiner Threads zu verwalten, in vielerlei Hinsicht wie ein Betriebssystem die Ressourcen seiner Prozesse verwaltet. Falls dies nicht erwünscht ist, kann diese Aufgabe einem in der Hierarchie höheren Prozess überlassen werden.

Auch bei Alta besteht ein Prozess aus einer Menge aus Threads, die in einer eigenen Threadgruppe verwaltet werden. Statische Felder geladener Klassen werden für jeden Prozess repliziert.

Interprozesskommunikation kann durch die Übergabe roher Bytedaten, Capabilities oder Objektreferenzen erfolgen. Die gemeinsame Verwendung von Objektreferenzen kann jedoch zu den in Abschnitt 3.3 beschriebenen Problemen führen, wodurch die Integrität eines Prozesses verletzt werden kann. Die Entwickler argumentieren aber, dass die Verwendung von Objektreferenzen in bestimmten Fällen sinnvoll sein könnte. Ein Prozess kann die Übergabe von Objektreferenzen aber verbieten, falls dies gewünscht ist.

Alta unterstützt die explizite Verwaltung von Speicher und CPU-Zeit durch die Alta System API. Der Zugriff auf das Netzwerk oder auf Dateien wird durch Server kontrolliert, mit denen Anwendungen kommunizieren können. Im Gegensatz zu K0 ist die Speicherbereinigung bei Alta ein Systemdienst, was eine Angriffsfläche für Denial-of-Service-Angriffe bietet, da eine Anwendung fortlaufend große Mengen an Speicher anfordern könnte, was eventuell eine Speicherbereinigung zur Folge hätte, die andere Prozesse unterbrechen würde. Durch Begrenzung des für eine Anwendung verfügbaren Speichers und der für sie reservierten CPU-Zeit können solche Probleme aber entschärft werden.

5.3.4 KaffeOS

KaffeOS ([BHL00], [Bac02]) ist der Nachfolger von K0 und ebenfalls eine modifizierte Version von Kaffe. Auch hier wurde besonders Wert auf die Isolation von Anwendungen und umfassendes Ressourcenmanagement gelegt, wobei die Entwickler sich am Design von regulären Betriebssystemen orientierten. KaffeOS bietet ebenfalls die Abstraktion des Prozesses. Im Vergleich zu Kaffe ist KaffeOS bis zu 11% langsamer, bietet dafür aber einen hohen Grad an Sicherheit.

Das Design von KaffeOS basiert auf den folgenden Prinzipien:

- **Isolation von Prozessen:**
  Jeder Prozess hat den Eindruck, als beanspruche er die Virtual Machine für sich allein. Ein Prozess kann weder beabsichtigt noch unbeabsichtigt auf die Daten eines anderen Prozesses zugreifen, da jedem Prozess ein eigener Class Loader zugewiesen ist und jeder Prozess über einen eigenen Heap verfügt. Des Weiteren dürfen Prozesse nicht unbegrenzt Ressourcen auf Systemebene blockieren und damit andere Prozesse von deren Nutzung abhalten. Um die Probleme in Zusammenhang mit internalisierten Strings zu beheben, verfügt jeder Prozess über eine eigene Datenbasis.

- **Sichere Beendigung von Prozessen:**
  Prozesse können aufgrund eines internen Fehlers oder eines externen Ereignisses abrupt enden. In beiden Fällen wird sichergestellt, dass die Integrität anderer Prozesse oder des Systems nicht verletzt wird. Beispielsweise darf ein Prozess nicht beendet werden, falls er anhand einer Sperre eine Ressource auf Systemebene blockiert. Um die sichere Beendigung von Prozessen zu gewährleisten, ist KaffeOS wie ein Betriebssystem in Kernel- und Benutzermodus unterteilt [BH99]. Einige Teile des Systems müssen im Kernelmodus ausgeführt werden, um dessen Integrität sicherzustellen (siehe Abbildung 5.3). Je nach Modus gelten unterschiedliche Regeln in Bezug auf die Beendigung von Prozessen und das Ressourcenmanagement. Beispielsweise können Prozesse, die im Benutzermodus ausgeführt werden, nach Belieben beendet werden. Prozesse, die im Kernelmodus ausgeführt werden, können nicht willkürlich beendet werden, da sie den Kernel in einem konsistenten Zustand hinterlassen müssen. Tritt eine Ausnahme im Kernelmodus auf, weiß KaffeOS damit umzugehen.

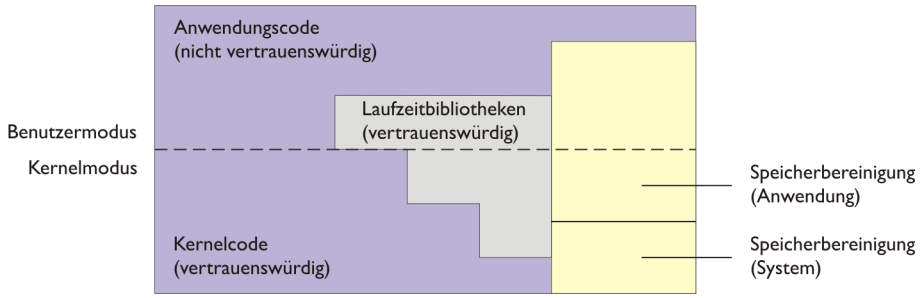

Abbildung 5.3: Kernel- und Benutzermodus in KaffeOS

- **Vollständige Rückgewinnung verbrauchten Speichers:**
  Wenn ein Prozess beendet wird, muss der von ihm verbrauchte Speicher vollständig zurückgewonnen werden. Um dies zu erreichen, sind Benutzerprozessen bestimmte Schreiboperationen verboten. Beispielsweise darf kein Prozess einem Zeiger eine Referenz auf ein Objekt im Heap eines anderen Prozesses zuweisen. Um dies zu garantieren, verwendet KaffeOS so genannte Write Barriers [Wil92]. KaffeOS verwendet Mechanismen der verteilten Speicherbereinigung, um einzelne Heaps unabhängig von anderen zu bereinigen.

- **Präzise Buchhaltung über die von Prozessen benutzten Ressourcen:**
  Angeforderter Speicher und benutzte CPU-Zeit werden für jeden Prozess abgerechnet, wobei Obergrenzen gesetzt werden können. Dadurch können negative Auswirkungen fehlerhafter oder schädlicher Anwendungen, wie etwa Denial-of-Service-Angriffe, wirkungsvoll abgewehrt werden. Der Speicherverbrauch von Prozessen kann umfassend kontrolliert werden: Es wird nicht nur der bei der Erzeugung von Objekten allozierte Speicher, sondern jegliche Speicheranforderung innerhalb der Virtual Machine im Auftrag von Prozessen berücksichtigt. Ansätze, deren Ressourcenmanagement auf der Transformation von Bytecode basiert (z.B. JRes), sind zu Letzterem nicht in der Lage, da die Virtual Machine dazu modifiziert werden müsste.

- **Gemeinsame Verwendung von Objekten durch Prozesse:**
  Prozesse können Objekte auf direkte Weise gemeinsam verwenden, um miteinander zu kommunizieren. Zu diesem Zweck kann jeder Prozess dynamisch spezielle Heaps anlegen und darin gewöhnliche Objekte erzeugen, die von allen Prozessen aus erreichbar sind. Diese Objekte dürfen keine Zeiger enthalten, die Objekte in den Heaps von Prozessen referenzieren, da diese Referenzen bei Beendigung eines Prozesses eine vollständige Rückgewinnung verbrauchten Speichers verhindern würden. Die Heapstruktur von KaffeOS ist in Abbildung 5.4 dargestellt.

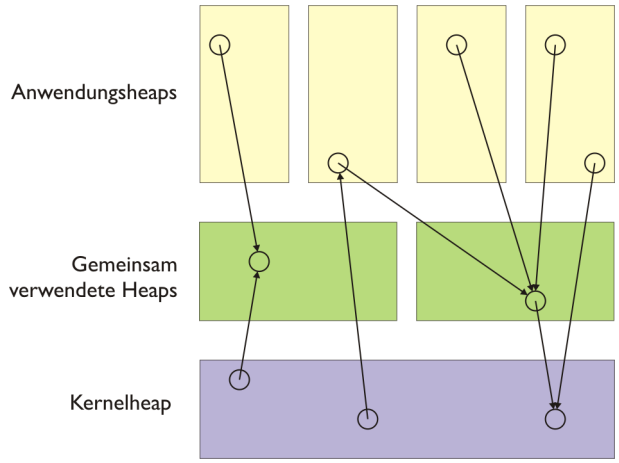

Abbildung 5.4: Heapstruktur in KaffeOS

- **Hierarchische Speicherverwaltung:**
  KaffeOS bietet ein hierarchisches Speicherverwaltungsmodell. Dabei wird jedem Heap ein so genanntes Memlimit zugeordnet, das aus einer Obergrenze nutzbaren Speichers und dem aktuellen Verbrauch besteht. Diese Memlimits bilden eine Hierarchie: Jedes Memlimit hat ein übergeordnetes Memlimit. Eine Ausnahme bildet dabei das Memlimit, das die Obergrenze nutzbaren Speichers aller Heaps verwaltet. Fordert ein Prozess Speicher an, wird das seinem Heap zugeordnete Memlimit belastet. Umgekehrt wird freigegebener Speicher diesem Memlimit gutgeschrieben. Dieser Prozess setzt sich rekursiv auf den übergeordneten Memlimits fort.

Alta, K0 und KaffeOS sind allesamt Prototypen einer Virtual Machine, die die Abstraktion des Prozesses unterstützt. KaffeOS ist dabei die am weitesten fortgeschrittene Entwicklung und vereint die Erfahrungen der Vorgängerprojekte. Auch wenn KaffeOS aus dem Jahr 2000 stammt, bietet es im Vergleich zu aktuellen kommerziellen Virtual Machines einen deutlich höheren Grad an Sicherheit.

5.3.5 Luna

Luna ([HE99a], [Haw00], [HE02]) ist ein Projekt, das wie die J-Kernel-Bibliothek an der Cornell Universität durchgeführt wurde. Das Luna-System besteht sowohl aus einer Erweiterung des Typsystems von Java als auch aus einer modifizierten Laufzeitumgebung. Da es sich nicht mehr um reines Java handelt, wurde der frei verfügbare Guavac Java-Compiler angepasst, um Luna Bytecode zu erzeugen und für dessen Ausführung eine modifizierte Version von Marmot [FKR+99] entwickelt. Das Marmot-System von Microsoft umfasst einen hochoptimierenden Compiler, eine Laufzeitumgebung und eine Laufzeitbibliothek, die eine große Teilmenge von Java umfasst.

Luna fokussiert vor allem die in Zusammenhang mit der Interprozesskommunikation bestehenden Probleme, geht diese aber in Gegensatz zu anderen Projekten auf unterster Ebene im Typsystem der Sprache an.

Luna verfügt ebenfalls über ein eigenes Prozessmodell. Prozesse werden wie in der J-Kernel-Bibliothek Tasks genannt und fassen eine Menge aus Threads zu einer Einheit zusammen.

Anwendungsklassen werden für jeden Prozess einzeln geladen. Aus der Literatur geht nicht hervor, ob dafür Class Loader oder andere Mechanismen verwendet werden. Systemklassen werden gemeinsam verwendet, wobei hinsichtlich der Isolation einige Klassen umgeschrieben werden mussten, um für jeden Task einen separaten statischen Zustand zu unterhalten.

Interprozesskommunikation basiert auf dem Konzept so genannter Remote Pointer, die im Folgenden entfernte Referenzen genannt werden. Diese ähneln Capabilities, sind aber fest im Typsystem verankert. In Luna dürfen Tasks ausschließlich über entfernte Referenzen kommunizieren. Entfernte Referenzen sind Referenzen von einem Task in einen anderen und unterscheiden sich im Typ von lokalen Referenzen durch eine abschließende Tilde (z.B. `String~`). Das Typsystem von Luna ist in Abbildung 5.5 abgebildet und zeigt den zum Typsystem von Java hinzugefügten Typ `ReferenceType~`.

```
Type = PrimitiveType | ReferenceType | ReferenceType~

PrimitiveType = boolean | byte | short | long | char | float | double

ReferenceType = ClassType | InterfaceType | Type[]
```

Abbildung 5.5: Das Typsystem von Luna

Bei Beendigung eines Tasks werden alle entfernten Referenzen automatisch widerrufen und die Objekte, Klassen und Threads eines Tasks sicher deallociert. Der Widerruf einer Capability ist aber auch manuell möglich, z.B. um einem Task das Zugriffsrecht auf ein vormals gemeinsam verwendetes Objekt zu entziehen. Ein derartiges Vorgehen wird durch Objekte der Klasse `Permit` [HE99b] realisiert.

Eine entfernte Referenz besteht aus einer Referenz auf das eigentliche Objekt und einer Referenz auf eine Instanz der Klasse `Permit`. Der in Luna eingeführte @-Operator wandelt eine lokale Referenz in eine entfernte Referenz um:

```
Permit p = new Permit();
String s = "Diplomarbeit";
String~ sr = s @ p;
```

Im Anschluss kann die entfernte Referenz anderen Tasks übergeben werden. Diese haben so lange Zugriff auf das dahinter stehende Objekt, bis der Zugriff widerrufen wird, was durch den Aufruf der Methode `revoke()` der entsprechenden Instanz der Klasse `Permit` erfolgt. Ein Zugriff auf eine widerrufene entfernte Referenz führt zu einer Ausnahme:

```
p.revoke(); // Zugriff auf die entfernte Referenz wird widerrufen

int length = sr.length(); // Ausnahme!
```

Um sicherzustellen, das nur entfernte Referenzen zwischen verschiedenen Tasks ausgetauscht werden, ist es nicht möglich, einem Feld eines Objekts, auf das über eine entfernte Referenz zugegriffen wird, eine lokale Referenz zuzuweisen oder sie einer Methode des Objekts als Parameter zu übergeben. Die Zuweisung, bzw. Übergabe entfernter Referenzen und primitiver Datentypen ist dagegen erlaubt.

Im Kontrast dazu sind Leseoperationen flexibler gestaltet. Eine lokale Referenz innerhalb eines entfernten Objekts kann gelesen oder von einer seiner Methoden zurückgegeben werden, wird jedoch automatisch in eine entfernte Referenz umgewandelt.

Der große Vorteil von Luna ist, dass sowohl die gemeinsame Verwendung von Objekten ohne allzu große Einschränkungen und Kosten, als auch eine sichere Beendigung von Tasks und ein Widerruf des Zugriffs auf gemeinsam verwendete Objekte möglich ist.

5.4 Isolation durch Betriebssystemprozesse

5.4.1 Persistent Reusable Java Virtual Machine

Die Entwicklung der Persistent Reusable Java Virtual Machine (PRJVM) ([DBC+00], [BPW+00], [IBM03]) durch IBM ist auf zwei Gründe zurückzuführen: Erstens herrschte Bedarf an einer Virtual Machine, die selbst im Falle eines Absturzes keine anderen Transaktionen in Mitleidenschaft ziehen und zweitens höchste Zuverlässigkeit ohne den Overhead des EJB-Frameworks bieten sollte.

Da auf dem damaligen Stand der Technik, an dem sich bis heute nichts geändert hat, eine vollständige Isolation von Transaktionen innerhalb derselben Virtual Machine nicht möglich war, blieb den Technikern bei IBM als einziger Ausweg außer einer kompletten Neuentwicklung die Verwendung separater Virtual Machines für die parallele Verarbeitung von Transaktionen. Das Transaktionsverarbeitungssystem benötigt dadurch zwar erheblich mehr Speicher, bietet aber vollständige Isolation.

Um höchste Sicherheit zu garantieren, darf eine Virtual Machine nach der Verarbeitung einer Transaktion allerdings nicht wiederverwendet werden. Der Grund dafür ist, dass eine Transaktion den Zustand der Virtual Machine verändern könnte und so eine Beeinträchtigung der nachfolgenden Transaktion herbeiführen würde, beispielsweise durch Veränderung statischer Felder einer Systemklasse. Daher muss für jede Transaktion eine neue Virtual Machine erzeugt werden. Die Dauer des Startvorgangs einer Virtual Machine ist aber inakzeptabel, da zwischen 20 und 100 Millionen Maschineninstruktionen ausgeführt werden müssen. Um dies zu vermeiden, wurden mit der Persistent Reusable Java Virtual Machine neue Konzepte eingeführt, die einen Neustart im Normalfall überflüssig machen.

Das wichtigste dieser Konzepte ist die Möglichkeit, eine Virtual Machine, die bereits eine Transaktion verarbeitet hat, in ihren Ursprungszustand zu versetzen. Diese Funktionalität wird durch eine neue JNI-Funktion namens `ResetJavaVM` implementiert. Damit diese Funktion aktiviert ist, muss die Persistent Reusable Java Virtual Machine zuvor mit dem Parameter `-Xresettable` aufgerufen werden.

Die Instandsetzung einer Virtual Machine kann aber fehlschlagen, falls ihr Zustand nach einer ausgeführten Transaktion irreparabel ist. Dieser Fall kann z.B. durch modifizierte Systemeigenschaften, geladene plattformabhängige Bibliotheken, erzeugte Threads und Prozesse oder umgeleitete Ein- und Ausgabeströme eintreten. In Folge dessen liefert der Aufruf der `ResetJavaVM`-Funktion `false` zurück.

Um möglichst viele Transaktionen parallel zu verarbeiten, werden mehrere Instanzen der Persistent Reusable Java Virtual Machine erzeugt. Eingehende Transaktionen werden von einer Master Virtual Machine auf diese Instanzen verteilt. Sie ist auch dafür zuständig, die jeweilige Instanz nach Abschluss einer Transaktion wieder instand zu setzen. Schlägt dieser Versuch fehl, wird die Instanz entfernt und eine neue Virtual Machine gestartet.

Die Persistent Reusable Java Virtual Machine unterteilt Klassen in drei Kategorien:

- **Systemklassen:**
  Systemklassen bestehen aus den Klassen primitiver Datentypen und den vom Primordial Class Loader und dem Class Loader für Standarderweiterungen geladenen Klassen. Die Lebensdauer dieser Klassen entspricht der Lebensdauer der Virtual Machine. Sie werden auch bei einer Instandsetzung nicht entfernt.

- **Middleware-Klassen:**
  Middleware-Klassen gehören üblicherweise zur Infrastruktur des Applikationsservers, beispielsweise die Klassen des EJB-Containers. Die Objekte dieser Infrastruktur könnten ihren Zustand auch bei Durchführung einer Instandsetzung beibehalten wollen und würden in diesem Fall nicht entfernt werden. Auch könnten sie gewisse Einstellungen für alle Transaktionen an der Virtual Machine vornehmen oder plattformabhängige Bibliotheken laden. Middleware-Klassen sind dazu berechtigt und versetzten die Virtual Machine dadurch nicht in einen irreparablen Zustand. Sie werden daher auch als vertrauenswürdige Middleware bezeichnet.

- **Anwendungsklassen:**
  Alle Klassen, die weder System- noch Middleware-Klassen sind, werden als Anwendungsklassen eingestuft. Sie enthalten die Geschäftslogik und wissen nichts über ihre Wiederverwendbarkeit oder den Vorgang einer Instandsetzung. Anwendungsklassen sind nicht vertrauenswürdig. Daher gelten wie für Enterprise Beans gewisse Regeln, die bei Verletzung zu einem irreparablen Zustand der Virtual Machine führen. Die Lebensdauer von Anwendungsklassen entspricht der Lebensdauer der zugehörigen Transaktion. Sie werden daher nach deren Abschluss entfernt. Das System kann aber einige oder alle Anwendungsklassen als gemeinsam verwendbare Klassen identifizieren. Bytecode und kompilierter Code gemeinsam verwendbarer Klassen werden von der Persistent Reusable Java Virtual Machine zwischengespeichert und wiederverwertet, falls die entsprechende Anwendung erneut ausgeführt wird.

Für jede dieser drei Kategorien ist ein eigener Class Loader zuständig. Dadurch wird unter anderem auch auf Methodenebene zwischen den Kategorien unterschieden. Beispielsweise gelten für Methoden von Anwendungsklassen mehr Einschränkungen in Bezug auf die Modifikation ihrer Umgebung als für Methoden von Middleware-Klassen.

Zusätzlich wird für jede der drei Kategorien ein eigener Heap verwaltet: Der System Heap, der Middleware Heap und der Transient Heap. Dabei werden für den Middleware und den Transient Heap unterschiedliche Speicherbereinigungsverfahren verwendet. Der Hauptvorteil dieser Unterteilung ist, dass transaktionale Daten sehr schnell bereinigt werden können.

Die Persistent Reusable Java Virtual Machine steht derzeit nur unter z/OS zur Verfügung und ist bei Kunden von IBM im Produktiveinsatz. Sie bietet derzeit die sicherste Transaktionsverarbeitung unter Java.

Kapitel 6

Application Isolation API

Isolate *noun.* pronounciation: ***isolet***.
1. A thing that has been isolated, as by geographic, ecologic or social barriers
(Quelle: American Heritage Dictionary)

Die zahlreichen Projekte, die im letzten Kapitel vorgestellt wurden und sich mit der Verbesserung von Java im Bereich der Isolation, der Interprozesskommunikation, des Ressourcenmanagements und der Beendigung von Prozessen befassen, machen deutlich, wie groß der Handlungsbedarf von offizieller Seite ist.

Die Bemühungen, eine einheitliche Lösung der bestehenden Probleme zu finden, führten zur Gründung einer Expertengruppe innerhalb des Java Community Process, die im April 2001 ihre Arbeit aufnahm. Innerhalb des Java Community Process werden technische Spezifikationen entwickelt, die in zukünftige Versionen von Java einfließen sollen. Diese Spezifikationen werden Java Specification Requests genannt. Der Java Specification Request, der sich mit der Isolation von Anwendungen befasst, trägt den Namen JSR 121: Application Isolation API Specification [25]. Zusätzliche Informationen sind auf der JSR-121 Interest Site [26] verfügbar. Die Expertengruppe wird von Grzegorz Czajkowski (Sun Microsystems, Inc.) geleitet und besteht unter anderem aus Vertretern von Hewlett-Packard, IBM, Motorola, Oracle, Philips, SAP und Sun, was die Bedeutung dieser Spezifikation zum Ausdruck bringt.

6.1 Isolates

Die Application Isolation API [JCP02] ermöglicht der Virtual Machine im Gegensatz zur bisherigen Verfahrensweise die vollständig isolierte Ausführung von Anwendungen. Zentraler Bestandteil der API ist die Klasse `Isolate`, die die Abstraktion einer isolierten Berechnung repräsentiert und Methoden zum starten, aussetzen, wiederaufnehmen und beenden enthält. Isolates können sicher beendet werden, ohne andere Anwendungen dadurch zu beeinflussen. Mit Hilfe so genannter Aggregates können mehrere Isolates zusammengefasst werden.

Eine Anwendung, die aus der Klasse `HelloWorld` besteht, könnte folgendermaßen aufgerufen werden:

```
try {
    Isolate isolate = new Isolate("HelloWorld", null);
    isolate.start();
} catch (IsolateStartupException e) {
    System.err.println(e);
}
```

Im Idealfall können parallel ausgeführte Isolates den Isolationsgrad von Betriebssystemprozessen erreichen. Wie diese Isolation erreicht wird, lässt die Spezifikation offen. Sie fordert lediglich, dass sie gewährleistet ist. Folglich kann die Isolation sowohl auf sprachbasierter als auch auf adressbasierter Sicherheit beruhen. Konkret heißt dies, dass eine Implementierung sowohl den Ansatz mehrerer Isolates innerhalb derselben Virtual Machine (1:n), als auch den Ansatz eines Isolates pro Virtual Machine bei Einsatz mehrerer Virtual Machines (1:1) verfolgen kann. Ferner ist auch eine Mischung beider Ansätze (m:n) denkbar.

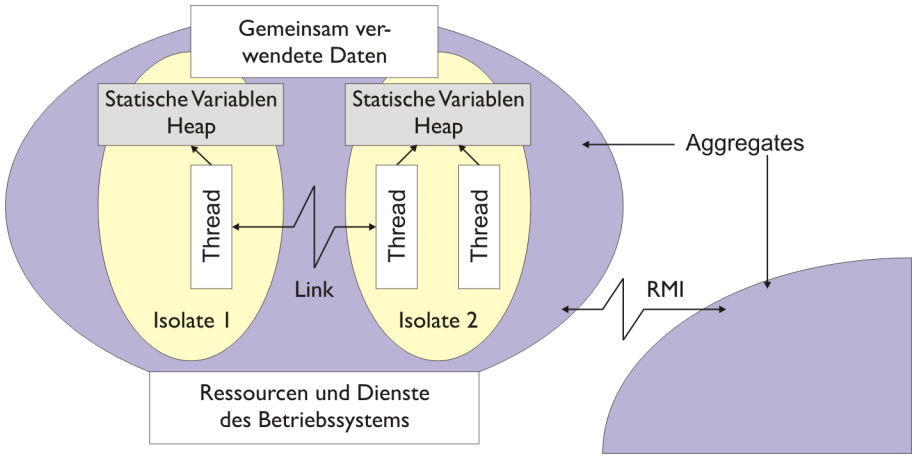

Abbildung 6.1: Aggregates/Isolates/Threads

Weiterhin können unterschiedliche Implementierungen einen unterschiedlichen Isolationsgrad zur Verfügung stellen. Alle konformen Implementierungen müssen jedoch Anwendungen gewisse Voraussetzungen garantieren, wie etwa logisch disjunkte Heaps oder separate Kopien statischer Felder. Darüber hinaus gehende Implementierungen könnten beispielsweise JNI-Code isoliert von anderen Anwendungen ausführen. Aber auch die Anpassung bestehender prozessbasierter Implementierungen (z.B. die Persistent Reusable Java Virtual Machine) an die Application Isolation API, ist denkbar.

Da Anwendungen isoliert voneinander ausgeführt werden, können sie ihre gegenseitige Präsenz außer durch Benutzung der Application Isolation API nicht feststellen. Die API stellt hierfür einen Namensdienst zur Verfügung, der Isolates durch eindeutige Namen identifizieren kann.

Interprozesskommunikation zwischen Isolates kann sowohl per RMI als auch durch so genannte Links erfolgen. Links stellen eine systemnahe Kommunikationsebene zur Verfügung, die zur effizienten Übertragung einfacher Datentypen (Byte-Felder, Byte-Puffer, serialisierte Objekte, Sockets und Strings) ausgelegt ist. Die Kommunikation findet durch abgeleitete Objekte der abstrakten Klasse `Link` statt. Dabei handelt es sich um uni- oder bidirektionale Kommunikationsverbindungen zwischen zwei Isolates, die Instanzen der Klasse `LinkMessage` austauschen. Ein Datenaustausch zweier Isolates könnte beispielsweise auf folgende Weise ablaufen:

```
// Sender-Isolate
...
LinkMessage message;
Data data = new Data();
message = LinkMessage.newSerializableMessage(data);
link.send(message);

// Empfänger-Isolate
...
LinkMessage message = link.receive();
Data data = (Data)message.getSerializable();
```

Die API definiert keine Schnittstellen für die Verwaltung von Ressourcen. Die Expertengruppe merkt aber an, dass zukünftige Entwicklungen in diesem Bereich die Application Isolation API berücksichtigen müssen. Eine gute Zusammenfassung der Thematik liefert Ciarán Bryce [Bry04]. Präsentationsunterlagen zu JSR-121 im Rahmen der JavaOne-Konferenzen in San Francisco sind auf der Interest Site vorhanden ([27], [28]).

Die Bedeutung dieses Java Specification Requests ist sicherlich unumstritten. Umso unverständlicher ist aber, warum die endgültige Spezifikation bis heute nicht fertig gestellt ist. Zuerst war eine Veröffentlichung in der Java 2, Standard Edition 5.0 (Codename Tiger) angedacht, später sollte die Spezifikation in Form eines Zusatzpakets sogar noch davor erscheinen [29]. Aufgrund der nahenden Fertigstellung von Tiger und der dafür benötigten Ressourcen wurde die Weiterentwicklung der Application Isolation API im Frühjahr 2004 vorerst auf Eis gelegt ([30], [31]). Im darauf folgenden Herbst wurde bekannt gegeben, dass nun doch wieder eine Veröffentlichung als Teil der kommenden Java 2, Standard Edition 6.0 (Codename Mustang) geplant sei, jedoch in abgespeckter Form. So sollte lediglich eine 1:1-Variante Einzug halten, die nur ein Isolate pro Virtual Machine unterstützt, dafür aber wenigstens eine offizielle Referenzimplementierung existiert ([32], [33], [34]). Ende 2004 entschied Sun jedoch, auch dieses Vorhaben nicht in die Tat umzusetzen und JSR-121 auch in Mustang nicht zu integrieren [35]. Wie aus den Beiträgen der JSR-121 Mailingliste entnommen werden kann, scheint Sun momentan weder Geld noch Ressourcen zur Weiterentwicklung der Spezifikation bereitzustellen. Seit Dezember 2004 gibt es keine Neuigkeiten über den weiteren Verlauf.

Trotz der laufenden Entwicklung des JSR-121 existieren bereits drei Referenzimplementierungen, die auf den Entwürfen der Spezifikation aufbauen. Sie werden in den nächsten drei Abschnitten vorgestellt.

6.2 JanosVM

Die JanosVM ([THL01], [Flu03a], [36]) ist der erste öffentlich verfügbare Prototyp einer Virtual Machine, die JSR-121 größtenteils implementiert. Die JanosVM wurde wie auch Alta, K0 und KaffeOS an der Universität von Utah entwickelt und baut auf den Erfahrungen der Vorgängerprojekte auf. Auch hier bildet eine modifizierte Version von Kaffe die Grundlage. In der Pressemitteilung [Flu03b] zur Veröffentlichung heißt es:

> The University of Utah's Flux Research Group announces a new release of the JanosVM, v1.0. This release includes the first publically available prototype of an emerging part of the Java standard, JSR-121 (Isolates), which provides an API for reliably controlling separate computational entities. This release also include a basic web server that uses Isolates, lazier class loading, stricter class file checking, stack overflow detection using guard pages, run-time access checking, a resync with the current CVS version of Kaffe, tests for class file integrity, chroot()'ing for teams, and the usual bug fixes.
>
> The Janos Virtual Machine (JanosVM) is an Open Source virtual machine for executing Java bytecodes. Unlike almost all virtual machines, the JanosVM supports multiple, separate process-like entities within a single VM, without reliance on any underlying OS or hardware support for such separation. The JanosVM supports asynchronous termination of uncooperative, buggy, or malicious Java applications.
>
> The JanosVM exposes the primitives needed to build a multi-process JVM, allowing users of the JanosVM to build their own customized Java-oriented operating systems. The JanosVM does not provide a complete environment for running untrusted user code. Rather, it provides clean and efficient building blocks for building robust Java-based OS's, such as may be needed by embedded systems, PDAs, servlet environments, peer-to-peer platforms, or active networks.
>
> The JanosVM is free software, developed from our KaffeOS which itself is based on Kaffe, a GPL'd highly portable Java virtual machine.

Die JanosVM verfolgt bei der Implementierung der Application Isolation API den Ansatz mehrerer Isolates pro Virtual Machine (1:m). Technisch gesehen handelt es sich bei der JanosVM nicht um Java, da die zu Grunde liegende Kaffe Virtual Machine eine von Sun unabhängige Eigenentwicklung darstellt.

Zu einer Telefonkonferenz der JSR-121 Expertengruppe am 27. Oktober 2004 war auch Pat Tullmann eingeladen. Tullmann ist einer der führenden Entwickler der JanosVM und war früher ebenfalls Mitglied der Expertengruppe. Während dieser Konferenz machte er die Bemerkung, dass die JanosVM seit über einem Jahr auf Eis liegen würde [34].

6.3 SAP VM Container

Javas Defizite im Bereich der Isolation veranlassten auch die Firma SAP, Transaktionsverarbeitung unter Java sicherer zu gestalten. Dies führte zur Entwicklung des SAP VM Containers ([KKL+02], [Smi04a], [Smi04b], [Par04]), bei dem eine Konformität zur Application Isolation API angestrebt wird.

Als Vorbild diente der SAP Web Application Server, dessen starke Isolation auf prozessbasierter Sicherheit beruht. Er besteht aus einem Dispatcher, der eingehende Anfragen auf eine Reihe von Arbeiterprozessen verteilt. Bei diesen handelt es sich um Betriebssystemprozesse, deren Isolation durch die Hardware erfolgt. Ein Arbeiterprozess verarbeitet zu jedem Zeitpunkt nur eine Anfrage. Im Falle eines Absturzes ist nur der jeweilige Benutzer betroffen, alle anderen Anfragen werden dadurch nicht beeinflusst.

Im SAP Web Application Server werden die Zustandsdaten einer Sitzung — auch Benutzerkontext genannt — nicht im Arbeiterprozess, sondern in einem von allen Arbeiterprozessen gemeinsam verwendeten Speicherbereich abgelegt. Auf diese Weise kann der Benutzerkontext bei der nächsten Anfrage einem beliebigen, nicht verwendeten Arbeiterprozess zugeordnet werden. Diese Operation ist sehr schnell, da keine Daten kopiert werden müssen. Dieses Konzept verhindert, dass Arbeiterprozesse zwischen den einzelnen Anfragen innerhalb einer Sitzung brach liegen. Genau diese Technik macht sich auch der SAP VM Container zunutze.

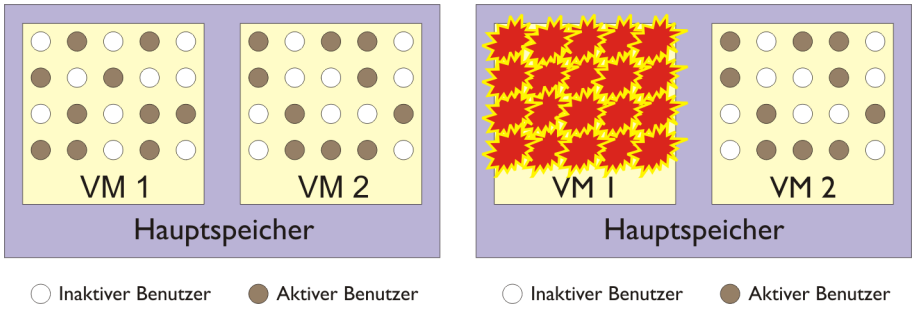

Abbildung 6.2: Bisheriges Problem beim Absturz einer Virtual Machine

Gewöhnlich sind mehrere Tausend Benutzer mit derselben Virtual Machine assoziiert. Kommt es zu einem Absturz, werden alle laufenden Transaktionen in Mitleidenschaft gezogen (siehe Abbildung 6.2).

Im ersten Schritt wird beim SAP VM Container die Anzahl der zu jedem Zeitpunkt mit einer Virtual Machine assoziierten Benutzer auf mehrere Hundert verringert. Der zweite Schritt besteht darin, wie beim SAP Web Application Server den Benutzerkontext zwischen den einzelnen Anfragen einer Sitzung in einem Speicherbereich außerhalb der Virtual Machine auszulagern (siehe Abbildung 6.3). Da normalerweise nur etwa 10 Prozent der mit einer Virtual Machine assoziierten Benutzer zeitgleich Anfragen senden und die Zustandsdaten der restlichen Sitzungen extern gespeichert sind, wird bei einem Absturz die Anzahl der betroffenen Transaktionen weiter verringert. Die Technik, den Benutzerkontext außerhalb der Virtual Machine zu speichern, wird als Shared Closure bezeichnet.

Die Shared Closures API ist eines der Hauptmerkmale des SAP VM Containers und hat eine Semantik ähnlich der Serialisierung bei RMI. Aus dem Namen lässt sich folgern, dass nicht nur einzelne Objekte, sondern die gesamte transitive Hülle aller erreichbaren Objekte ausgehend von einem Wurzelobjekt kopiert und gemeinsam verwendet wird. Dieses Verhalten gleicht dem der Serialisierung, nur dass Operationen auf den Objekten schneller ausgeführt werden und ein spezieller Mechanismus namens Mapping unterstützt wird.

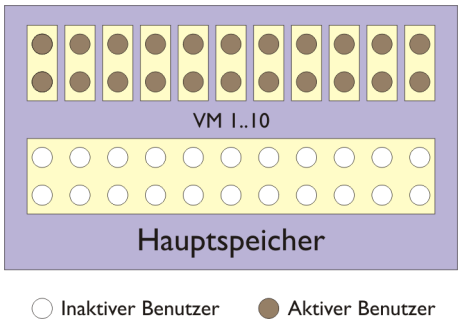

Abbildung 6.3: Auslagerung des Benutzerkontexts

Eine Shared Closure wird erzeugt oder aktualisiert, indem eine Objektreferenz an die API übergeben wird. Die Objekte innerhalb der transitiven Hülle ausgehend von diesem Wurzelobjekt werden anschließend in den gemeinsam verwendeten Speicherbereich kopiert, ohne dass Änderungen an den Objekten in der Virtual Machine vorgenommen werden.

Auf einer bestehenden Shared Closure können zwei verschiedene Operationen ausgeführt werden:

- **Copy-Operation:**
  Die Objekte innerhalb der transitiven Hülle werden in den Heap einer anderen Virtual Machine kopiert (siehe Abbildung 6.4). Die Objekte werden zu gewöhnlichen lokalen Objekten und können modifiziert werden. Diese Operation wird z.B. verwendet, um einen extern gespeicherten Benutzerkontext bei Absturz einer Virtual Machine auf eine andere zu übertragen.

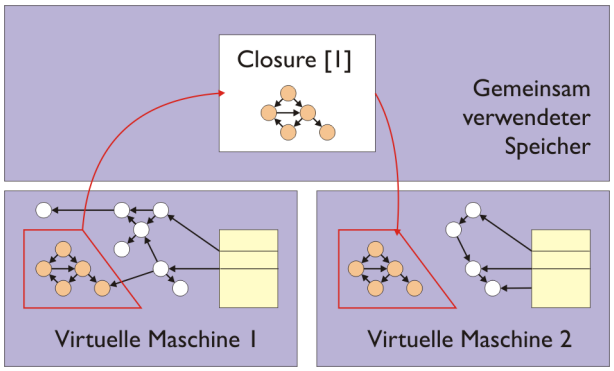

Abbildung 6.4: Erzeugung und Kopie einer Shared Closure

- **Map-Operation:**
  Die Objekte innerhalb der transitiven Hülle werden nicht in den Heap einer anderen Virtual Machine kopiert, sondern nur in ihren Adressraum eingeblendet (siehe Abbildung 6.5). Im Vergleich zu Copy ist diese Operation sehr schnell, da keine Daten transferiert werden. Insbesondere wird kein neuer Speicher benötigt, die in den Adressraum der Virtual Machine eingeblendeten Objekte sind jedoch schreibgeschützt. Diese Operation eignet sich am besten, um selten geänderte Konfigurationsdaten gemeinsam zu verwenden.

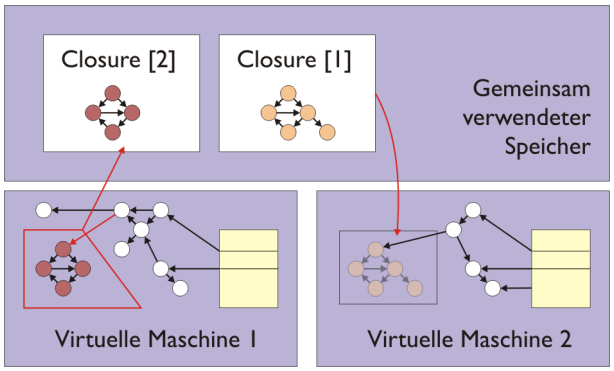

Abbildung 6.5: Versionierung und Einblendung einer Shared Closure

Ein impliziter Versionierungsmechanismus stellt sicher, dass die bei Aktualisierung einer Shared Closure entstehenden Änderungen keinen Einfluss auf eingeblendete Shared Closures einer Vorgängerversion haben. Nachfolgende Einblendungen verwenden aber immer die aktuellste Version.

In den Forschungslabors wird an einer Lösung des Isolationsproblems gearbeitet, die über den bisherigen Ansatz hinausgeht. Ziel ist die Vereinigung der virtuellen Maschinen für ABAP- und Java-Anwendungen. Beide virtuellen Maschinen werden innerhalb desselben Prozesses ausgeführt und bieten vollständige Isolation, da zu jedem Zeitpunkt nur eine Anfrage verarbeitet wird.

In diesem Zusammenhang steht ein neues Konzept namens Process Attachable Virtual Machines. Dabei wird die virtuelle Maschine vom korrespondierenden Prozess getrennt, indem sie durch eine Kopie des von ihr belegten Speichers schnell zwischen Prozessen ausgetauscht werden kann (siehe Abbildung 6.6).

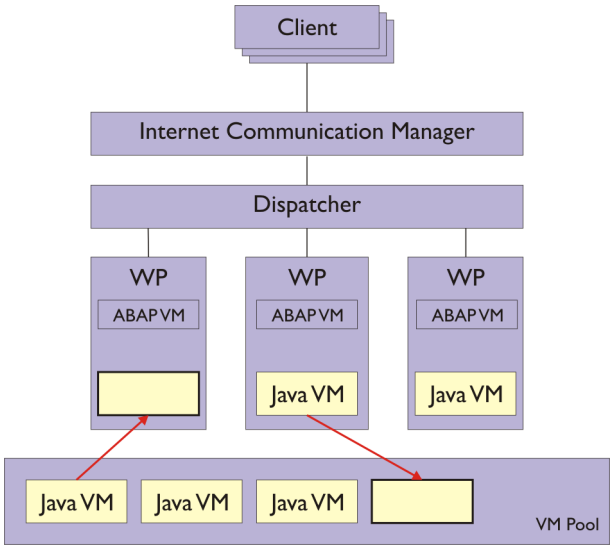

Abbildung 6.6: Trennung von Virtual Machine und Prozess

Das Speicherabbild einer geladenen virtuellen Maschine inklusive des Applikationsservers und der installierten Anwendungen dient weiterhin als Vorlage, um in kürzester Zeit neue virtuelle Maschinen erzeugen zu können.

Die Anzahl der Arbeiterprozesse werden dem Arbeitsspeicher des Systems entsprechend konfiguriert. Die Anzahl der virtuellen Maschinen überschreitet normalerweise die der Arbeiterprozesse und die Kapazität des Arbeitsspeichers. Dieser Sachverhalt ist zurückzuführen auf Situationen, in denen eine virtuelle Maschine durch gewisse Operationen keine CPU-Zeit in Anspruch nimmt. In solchen Fällen wird die virtuelle Maschine vom Prozess getrennt, der durch Zuordnung einer anderen virtuellen Maschine neue Anfragen verarbeiten kann.

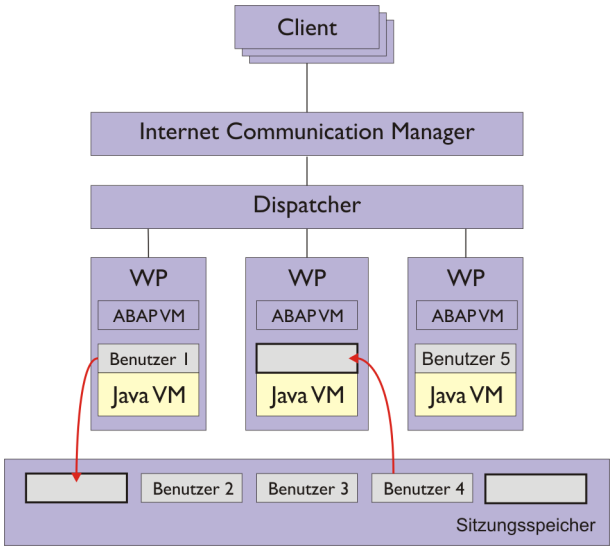

Abbildung 6.7: Trennung von Virtual Machine und Benutzerkontext

Auch bei diesem Konzept wird der Benutzerkontext unter Zuhilfenahme der Shared Closures API in einem gemeinsam verwendeten Speicherbereich ausgelagert. Nach jeder Anfrage wird die virtuelle Maschine vom Benutzerkontext getrennt (siehe Abbildung 6.7). Auf diese Weise können virtuelle Maschinen unabhängig vom Benutzerkontext verwendet werden.

6.4 Multi-Tasking Virtual Machine

Im Rahmen des Barcelona-Projekts [37] entwickelt die Firma Sun momentan eine neue Virtual Machine, die höchsten Sicherheitsansprüchen Rechnung trägt und Multi-Tasking Virtual Machine (MVM) ([CD01], [SM04], [Hei05]) genannt wird. Sie implementiert JSR-121 und verfolgt den Ansatz mehrerer Isolates innerhalb derselben Virtual Machine (1:n).

Das Design der MVM wurde durch drei Zielvorgaben bestimmt. Erstens sollten sich Anwendungen an keiner Stelle innerhalb der Virtual Machine beeinflussen können. Zweitens sollte jeder Anwendung der Eindruck vermittelt werden, als würden keine weiteren Anwendungen innerhalb derselben Virtual Machine ausgeführt. Drittens sollte das Design zu einer guten Leistung und Skalierbarkeit führen. Die Multi-Tasking Virtual Machine ist eine ganzheitliche Lösung, die alle in Kapitel 3 beschriebenen Defizite der Virtual Machine adressiert und behebt.

Die Untersuchung aller Komponenten der virtuellen Maschine und eine anschließende Entscheidung, ob diese gemeinsam von Anwendungen genutzt werden können oder nicht, stellt das Grundprinzip des Designs dar [CDN02]. Beispielsweise kann die Laufzeitrepräsentation von Klassen größtenteils gemeinsam verwendet werden und nur ein Teil muss pro Isolate repliziert werden, z.B. der statische Zustand. Zu diesem Zweck unterhält die MVM eine Liste, die für jedes Isolate wiederum eine Liste enthält, in der die für jedes Isolate individuellen Zustandsdaten aller geladenen Klassen gespeichert sind. Die Entwickler ziehen dabei Parallelen zu einer Vorgängerversion der Persistent Reusable Java Virtual Machine [DBC+00].

Zusätzlich zur gemeinsam verwendeten Laufzeitrepräsentation von Klassen werden zum Teil auch dynamisch kompilierte Klassen gemeinsam verwendet. Insgesamt wird durch die gemeinsame Nutzung des Bytecodes und des Codes der kompilierten Klassen der Startvorgang von Anwendungen erheblich beschleunigt. Lediglich beim erstmaligen Zugriff auf eine Klasse, müssen die damit verbundenen Schritte durchgeführt werden.

Die Probleme in Zusammenhang mit den vormals gemeinsam verwendeten Ereignis- und Finalisierungswarteschlangen werden ebenfalls durch Replikation gelöst. Plattformabhängiger Code wird sicher in einem separaten Prozess ausgeführt [CDW01] (siehe Abbildung 6.8). In Folge dessen stellt auch ein fehlerhafter Ressourcenadapter kein allzu großes Problem mehr dar. Insgesamt werden so die Defizite im Bereich der Isolation beseitigt (siehe Abschnitt 3.2).

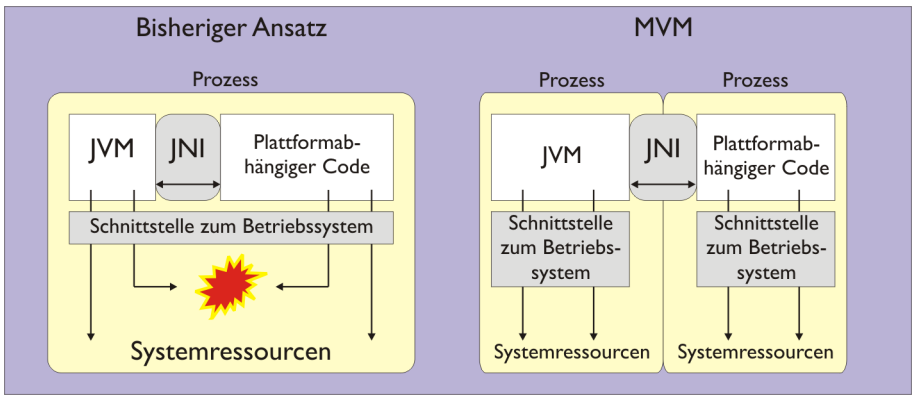

Abbildung 6.8: Ausführung plattformabhängigen Codes bisher und bei der MVM

Da Anwendungen in der MVM gemäß JSR-121 isoliert voneinander ausgeführt werden, verschwinden die bisherigen Probleme in Zusammenhang mit der Interprozesskommunikation und der Beendigung von Prozessen (siehe Abschnitt 3.3 und 3.5). Es ist einer Anwendung schlicht und einfach nicht mehr möglich, anderen Anwendungen Objektreferenzen zu übergeben. Selbstverständlich können Isolates dennoch miteinander kommunizieren. Dies ist sowohl über die in JSR-121 enthaltene Link API als auch per RMI möglich. Darüber hinaus verwendet die MVM jedoch einen dritten Mechanismus zur Interprozesskommunikation, der im Rahmen des Projekts Incommunicado unabhängig von der JSR-121-Expertengruppe entwickelt wurde [PCDV02] und sollte zumindest damals in zukünftigen Versionen an die Link API angepasst werden.

Der als Cross-Isolate Method Invocation (XIMI) bezeichnete Mechanismus ist speziell auf die Kommunikation zwischen Isolates zurechtgeschnitten und ist damit in Bezug auf Komplexität und Leistung RMI weit überlegen. Die Kommunikation findet dabei mit Hilfe so genannter Portals statt. Ein Portal ist eine Instanz der abstrakten Klasse `Portal` und befindet sich auf der Empfängerseite der Verbindung. Zum Zweck der Kommunikation muss das als Server fungierende Isolate ein Portal erzeugen und über eine bestehende Kommunikationsverbindung (z.B. als Nachricht über einen Link oder über ein anderes Portal) versenden. Jedes Portal ist einem Zielobjekt und mehreren Stub-Objekten zugeordnet. Das Portal und das Zielobjekt befinden sich innerhalb des Servers, während die Stub-Objekte innerhalb des Clients residieren. Die Semantik eines Isolate-übergreifenden Methodenaufrufs ähnelt der bei RMI insofern, als dass Portals per Referenz übergeben, alle anderen Objekte aber gemäß der Serialisierungsemantik kopiert werden, obwohl die Implementierung keine explizite Serialisierung durchführt, sondern dies auf effizientere Weise erledigt. Ähnlich wird bei Rückgabewerten verfahren.

Ein weiteres Konzept der MVM ist der Mehrbenutzerbetrieb [CDT03]. Dadurch können mehrere Anwendungen innerhalb derselben Virtual Machine unter unterschiedlichem Benutzerkontext ausgeführt werden. So kann ein Benutzer sicher auf seine privaten Dateien zugreifen und plattformabhängige Bibliotheken laden, ohne andere Anwendungen zu beeinflussen.

Realisiert wird dies indem Isolates von verschiedenen Benutzern durch eine separate Anwendung namens Jlogin gestartet werden. Jlogin dient als Hintergrundprozess, der benutzerbezogene Anfragen der Isolates verarbeitet, beispielsweise den Zugriff auf eine Datei im privaten Verzeichnis des entsprechenden Benutzers oder das Laden einer plattformabhängigen Bibliothek. Jedes Isolate verfügt über eigene Eingabe-, Ausgabe- und Fehlerströme, die mit dem korrespondierenden Jlogin-Prozess verbunden sind. Des Weiteren kann ein Benutzer mehrere Jlogin-Instanzen ausführen und so an mehreren Sitzungen teilnehmen. Bei Start eines Jlogin-Prozesses verbindet sich dieser über einen Socket mit dem Mserver, einem Isolate innerhalb der MVM, und überträgt den Benutzerkontext. Anschließend wird der Mserver veranlasst, das gewünschte Isolate zu starten und dessen Eingabe-, Ausgabe- und Fehlerströme mit dem Jlogin-Prozess zu verbinden. Abbildung 6.9 zeigt 3 Benutzer, die jeweils mit Hilfe eines Jlogin-Prozesses ein Isolate gestartet haben.

Der letzten Schritt hin zu einer betriebssystemähnlichen Laufzeitumgebung ist das Ressourcenmanagement der MVM ([CHSS02], [CHS+03], [JCKS04]), womit schließlich alle Problembereiche der herkömmlichen Virtual Machine abgedeckt sind.

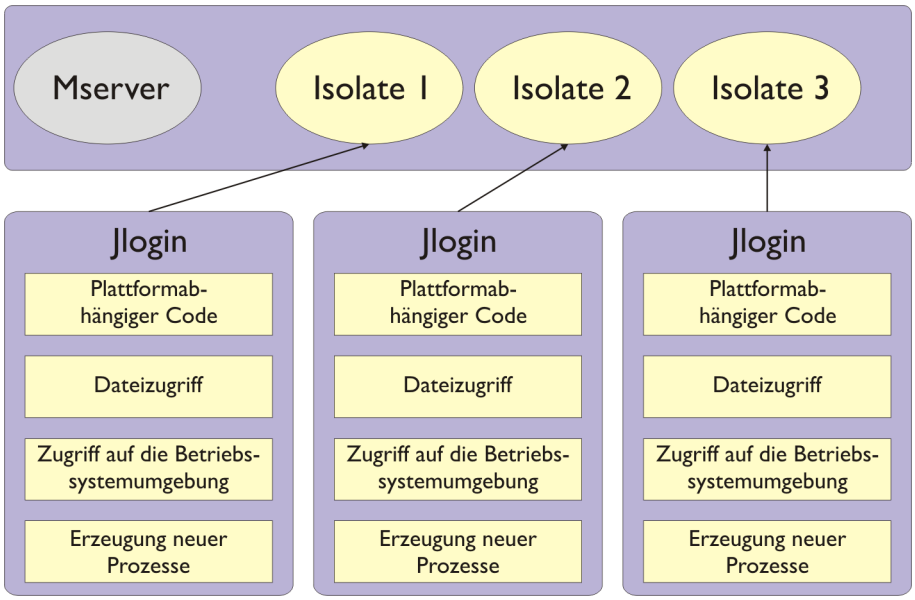

Abbildung 6.9: Drei Benutzer führen Isolates in der MVM aus

Die Multi-Tasking Virtual Machine scheint daher eine umfassende Lösung zu werden, um Anwendungen und Transaktionen unter Java parallel, sicher und effizient ausführen zu können. Die Forscher des Barcelona-Projekts experimentieren auch mit dem Einsatz in J2EE-Applikationsservern [JDC+04] und beschreiben zwei Einsatzszenarien der Application Isolation API: Architekturbasierte und anwendungsbasierte Isolation.

Bei architekturbasierter Isolation wird jede Komponente eines J2EE-Applikationsservers (z.B. der EJB-Container), in einem eigenen Isolate ausgeführt. Der Vorteil ist, dass an der bisherigen J2EE-Architektur nur geringe Änderungen vorgenommen werden müssen. Nachteilig sind dabei hohe Kommunikationskosten und die Schwierigkeit, Ressourcen auf Anwendungsebene zu verwalten. Bei anwendungsbasierter Isolation werden gesamte Anwendungen voneinander isoliert. Von Vorteil sind hierbei niedrige Kommunikationskosten und das Ressourcenmanagement auf Anwendungsebene. Der große Nachteil ist die Replikation der Serverkomponenten für jede Anwendung.

Bei beiden Ansätzen sind verschiedene Stufen der Granularität denkbar, z.B. könnte jedes Servlet innerhalb des Web-Containers in einem eigenen Isolate ausgeführt werden. Eine dahingehende Modifikation bestehender Applikationsserver ist schwierig, wie die Forscher des Barcelona-Projekts erläutern [JDC+04]:

> This experiment clearly demonstrated that it is not trivial to take a system that was designed in the context of a shared address space and restructure it into multiple address spaces. Approximately 7000 lines of code were required to implement iservlets (Anmerkung des Autors: Servlets in separaten Isolates) without modifying the Catalina codebase. The experiment discouraged us from pursuing the use of small-grain isolates in other areas of J2EE, such as individual EJBs.

Unbeantwortet bleibt, ob eine solche Isolation nötig ist, um die ACID-Eigenschaften zu 100 Prozent einzuhalten. Die meisten Probleme werden durch die MVM zwar gelöst, eine gegenseitige Beeinflussung von Transaktionen durch statische Variablen von Anwendungsklassen könnte aber dennoch stattfinden — abgesehen davon, dass deren Benutzung durch die EJB-Spezifikation verboten ist. Jedoch ist unklar, ob dies überhaupt als Bruch der Isolation gewertet werden kann. Schließlich vollzieht sich die Kommunikation innerhalb derselben Anwendung. Diese Frage stellt sich weder bei der Persistent Reusable Java Virtual Machine noch beim SAP VM Container und es ist abzuwarten, ob die MVM höchsten Sicherheitsanforderungen gerecht werden kann. Mit endgültiger Gewissheit kann dies natürlich erst nach ihrer Veröffentlichung gesagt werden.

Zum Zeitpunkt der Abgabe dieser Arbeit (Mai 2005) stand die Veröffentlichung der MVM zu Forschungszwecken kurz bevor. Laut Angaben des Chefentwicklers Grzegorz Czajkowski ([38], [39]) basiert die MVM auf einer eingeschränkten Erweiterung des J2SE Development Kit 5.0 (JDK). Die Implementierung der Application Isolation API fußt auf frühen Entwürfen der Spezifikation, die jedoch an die MVM angepasst wurden. Nach Fertigstellung der Schnittstelle wird die MVM daran angepasst. Die Verwaltung von Ressourcen ist noch nicht möglich, aber in folgenden Versionen vorgesehen. Um über die Veröffentlichung zu Forschungszwecken informiert zu werden, kann man sich unter [40] registrieren.

Kapitel 7

Zusammenfassung

Mit Einführung der Java 2 Platform, Enterprise Edition hat sich Java auch im Bereich serverseitiger Geschäftsanwendungen etabliert. Spätestens ab diesem Zeitpunkt wurde es zur Notwendigkeit, Anwendungen und Transaktionen innerhalb derselben Virtual Machine parallel auszuführen. Dadurch wandelte sich die Java Plattform allmählich von einer einfachen virtuellen Maschine zu einer betriebssystemähnlichen Laufzeitumgebung. Java wurde jedoch nie als virtuelles Betriebssystem konzipiert. Diese Tatsache macht sich durch die Probleme in den Bereichen Isolation, Interprozesskommunikation, Ressourcenmanagement und der Beendigung von Prozessen bemerkbar.

Die meisten Defizite sind im Bereich der Isolation anzutreffen. Die Probleme in Zusammenhang mit statischen Feldern und synchronisierten Klassenmethoden von Systemklassen, internalisierten Strings, systemweiten Ereignis- und Finalisierungswarteschlangen und plattformabhängigem Code können zur gegenseitigen Beeinflussung von Anwendungen und Transaktionen führen. Gemeinsam verwendete Objektreferenzen lassen die Grenzen von Prozessen verschwimmen und machen deren saubere Beendigung unmöglich. Das gänzlich fehlende Ressourcenmanagement bietet eine Angriffsfläche für Denial-of-Service-Angriffe.

Die Isolation herkömmlicher Java-Anwendungen innerhalb derselben Virtual Machine ist daher unzureichend. Etwas anders sieht es bei EJB-Anwendungen aus. Die EJB-Spezifikation verringert den Handlungsspielraum von transaktionalen Objekten. Dadurch werden einige der bei herkömmlichen Java-Anwendungen möglichen Isolationsprobleme irrelevant. Durch die strikte Einhaltung der Spezifikation wird die Gefahr einer gegenseitigen Beeinflussung zwar minimal, die praktische Untersuchung möglicher Probleme ergab jedoch, dass Enterprise Beans, die gegen Verbote der Spezifikation verstoßen, zumindest von WebSphere nicht beanstandet werden. Die Beendigung sämtlicher Transaktionen durch fehlerhaften plattformabhängigen Code und Denial-of-Service-Angriffe können aber auch durch korrekte Programmierung nicht verhindert werden. Die Probleme bei parallel ausgeführten EJB-Anwendungen lassen sich auf parallel ausgeführte Transaktionen derselben EJB-Anwendung übertragen. An dieser Stelle müsste der Begriff Isolation jedoch genauer definiert werden. Beispielsweise ist unklar, ob die unzulässige Verwendung statischer Felder in Anwendungsklassen die ACID-Eigenschaften verletzt.

In vielen Umgebungen mag die Sicherheit der EJB-Architektur ausreichend sein. Soll das Transaktionsverarbeitungssystem aber absolute Transaktionssicherheit bieten, so ist derzeit vom Einsatz der EJB-Technologie abzuraten.

Die zahlreichen Probleme veranlassten einige Universitäten und Firmen, eigene Lösungen zu finden. Keiner dieser Lösungsansätze deckt jedoch alle Problembereiche vollständig ab. Für absolut sichere Transaktionsverarbeitung kommt derzeit nur die Persistent Reusable Java Virtual Machine in Frage, auch dabei handelt es sich aber nicht um eine ganzheitliche Lösung. Ein weiterer Nachteil ist, dass sie nur unter z/OS zur Verfügung steht.

Angesichts dieser Tatsachen nahm im April 2001 eine Expertengruppe innerhalb des Java Community Process ihre Arbeit an der Spezifikation der Application Isolation API (JSR-121) auf, die die bestehenden Defizite beseitigen sollte. Bis zum heutigen Tag wurde sie aus diversen Gründen allerdings nicht fertig gestellt.

Ein Prototyp, der auf Entwürfen der Spezifikation aufbaut, ist die JanosVM der Universität von Utah. Auch SAP sah sich zum Handeln veranlasst und entwickelt derzeit den SAP VM Container, der allem Anschein nach konform zu JSR-121 ist. Wie die Persistent Reusable Java Virtual Machine zielt er aber insbesondere auf sichere Transaktionsverarbeitung ab. Parallel zur Arbeit an der Application Isolation API wird in den Forschungslabors von Sun seit Jahren an der Multi-Tasking Virtual Machine gearbeitet, die das Ziel einer ganzheitlichen Lösung verfolgt. Ihre Veröffentlichung zu Forschungszwecken stand zum Zeitpunkt der Abgabe dieser Arbeit kurz bevor.

Wann die endgültige Spezifikation der Application Isolation API verabschiedet wird, bzw. die Multi-Tasking Virtual Machine Marktreife erreicht, ist noch nicht absehbar, vermutlich aber nicht vor 2006.

Anhang A

Quellcode

A.1 Die Denial-of-Service-Unternehmensanwendung

A.1.1 Der Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar id="ejb-jar_ID" version="2.1"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
    <display-name>
    Thesis</display-name>
    <enterprise-beans>
        <entity id="Account">
            <ejb-name>Account</ejb-name>
            <home>ejbs.AccountHome</home>
            <remote>ejbs.Account</remote>
            <local-home>ejbs.AccountLocalHome</local-home>
            <local>ejbs.AccountLocal</local>
            <ejb-class>ejbs.AccountBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>ejbs.AccountKey</prim-key-class>
            <reentrant>false</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>Account</abstract-schema-name>
            <cmp-field id=...>
                <field-name>id</field-name>
            </cmp-field>
            <cmp-field id=...>
                <field-name>value</field-name>
            </cmp-field>
        </entity>
        <session id="Transfer">
            <ejb-name>Transfer</ejb-name>
            <home>ejbs.TransferHome</home>
            <remote>ejbs.Transfer</remote>
            <local-home>ejbs.TransferLocalHome</local-home>
            <local>ejbs.TransferLocal</local>
            <ejb-class>ejbs.TransferBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <ejb-ref id=...>
                <description>
                </description>
                <ejb-ref-name>ejb/Account</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>ejbs.AccountHome</home>
                <remote>ejbs.Account</remote>
                <ejb-link>Account</ejb-link>
            </ejb-ref>
        </session>
        <session id="Test">
            <ejb-name>Test</ejb-name>
            <home>ejbs.TestHome</home>
            <remote>ejbs.Test</remote>
            <ejb-class>ejbs.TestBean</ejb-class>
```

```
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
    <ejb-client-jar>ThesisClient.jar</ejb-client-jar>
</ejb-jar>
```

A.1.2 Enterprise Bean: Account

Account.java

```
package ejbs;

/**
 * Remote interface for Enterprise Bean: Account
 */
public interface Account extends javax.ejb.EJBObject {

    public int getValue() throws java.rmi.RemoteException;

    public void setValue(int newValue) throws java.rmi.RemoteException;

    public void deposit(int value) throws java.rmi.RemoteException;

    public void withDraw(int value) throws java.rmi.RemoteException;
}
```

AccountBean.java

```
package ejbs;

/**
 * Bean implementation class for Enterprise Bean: Account
 */
public abstract class AccountBean implements javax.ejb.EntityBean {

    private javax.ejb.EntityContext myEntityCtx;

    public void setEntityContext(javax.ejb.EntityContext ctx) {
        myEntityCtx = ctx;
    }

    public javax.ejb.EntityContext getEntityContext() {
        return myEntityCtx;
    }

    public void unsetEntityContext() {
        myEntityCtx = null;
    }

    public ejbs.AccountKey ejbCreate(int id) throws
        javax.ejb.CreateException {
        setId(id);
        return null;
    }

    public void ejbPostCreate(int id) throws javax.ejb.CreateException {
    }

    public void ejbActivate() {
    }

    public void ejbLoad() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() throws javax.ejb.RemoveException {
    }

    public void ejbStore() {
    }

    public abstract int getId();

    public abstract void setId(int newId);

    public abstract int getValue();

    public abstract void setValue(int newValue);

    public void deposit(int value) {
        setValue(getValue() + value);
    }

    public void withDraw(int value) {
        setValue(getValue() - value);
    }
}
```

AccountHome.java

```
package ejbs;

/**
 * Home interface for Enterprise Bean: Account
 */
public interface AccountHome extends javax.ejb.EJBHome {

    public ejbs.Account create(int id) throws javax.ejb.CreateException,
        java.rmi.RemoteException;

    public ejbs.Account findByPrimaryKey(ejbs.AccountKey primaryKey) throws
        javax.ejb.FinderException, java.rmi.RemoteException;
}
```

AccountKey.java

```
package ejbs;

/**
 * Key class for Entity Bean: Account
 */
public class AccountKey implements java.io.Serializable {

    static final long serialVersionUID = ...;

    public int id;

    public AccountKey() {
    }

    public AccountKey(int id) {
        this.id = id;
    }

    public boolean equals(java.lang.Object otherKey) {
        if (otherKey instanceof ejbs.AccountKey) {
            ejbs.AccountKey o = (ejbs.AccountKey) otherKey;
            return ((this.id == o.id));
        }
        return false;
    }

    public int hashCode() {
        return ((new java.lang.Integer(id).hashCode()));
    }
}
```

AccountLocal.java

```
package ejbs;

/**
 * Local interface for Enterprise Bean: Account
 */
public interface AccountLocal extends javax.ejb.EJBLocalObject {

    public int getValue();

    public void setValue(int newValue);

    public void deposit(int value);

    public void withDraw(int value);
}
```

AccountLocalHome.java

```
package ejbs;

/**
 * Local Home interface for Enterprise Bean: Account
 */
public interface AccountLocalHome extends javax.ejb.EJBLocalHome {

    public ejbs.AccountLocal create(int id) throws
        javax.ejb.CreateException;

    public ejbs.AccountLocal findByPrimaryKey(ejbs.AccountKey primaryKey)
        throws javax.ejb.FinderException;
}
```

A.1.3 Enterprise Bean: Transfer

Transfer.java

```
package ejbs;

/**
 * Remote interface for Enterprise Bean: Transfer
 */
public interface Transfer extends javax.ejb.EJBObject {

    public void transfer(int primaryKey1, int primaryKey2, int value) throws
        Exception;
}
```

TransferBean.java

```
package ejbs;

import com.ibm.etools.service.locator.ServiceLocatorManager;
import ejbs.Account;
import ejbs.AccountHome;
import ejbs.AccountKey;
import java.rmi.RemoteException;
import javax.ejb.*;
import javax.naming.*;

/**
 * Bean implementation class for Enterprise Bean: Transfer
 */
public class TransferBean implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext mySessionCtx;

    private final static String STATIC_AccountHome_REF_NAME = "ejb/Account";

    private final static Class STATIC_AccountHome_CLASS = AccountHome.class;

    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }

    public void setSessionContext(javax.ejb.SessionContext ctx) {
        mySessionCtx = ctx;
    }

    public void ejbCreate() throws javax.ejb.CreateException {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }

    protected Account find_AccountHome_findByPrimaryKey(AccountKey
        primaryKey) {
        AccountHome anAccountHome = (AccountHome) ServiceLocatorManager
                .getRemoteHome(STATIC_AccountHome_REF_NAME,
                        STATIC_AccountHome_CLASS);
        try {
            if (anAccountHome != null)
                return anAccountHome.findByPrimaryKey(primaryKey);
        } catch (javax.ejb.FinderException fe) {
            fe.printStackTrace();
        } catch (RemoteException re) {
            re.printStackTrace();
        }
        return null;
    }

    public void transfer(int primaryKey1, int primaryKey2, int value)
        throws Exception {
        Account account1 = find_AccountHome_findByPrimaryKey(new
            AccountKey(primaryKey1));
        Account account2 = find_AccountHome_findByPrimaryKey(new
            AccountKey(primaryKey2));

        account1.withDraw(value);
        account2.deposit(value);
    }
}
```

TransferHome.java

```
package ejbs;

/**
 * Home interface for Enterprise Bean: Transfer
 */
public interface TransferHome extends javax.ejb.EJBHome {

    public ejbs.Transfer create() throws javax.ejb.CreateException,
        java.rmi.RemoteException;
}
```

A.1.4 Enterprise Bean: Test

Test.java

```
package ejbs;

import java.rmi.RemoteException;

/**
 * Remote interface for Enterprise Bean: Test
 */
public interface Test extends javax.ejb.EJBObject {

    public void allocateMemory(int megabytes, int millisecondsToWait,
        boolean store) throws RemoteException;
}
```

TestBean.java

```
package ejbs;

import java.util.*;

/**
 * Bean implementation class for Enterprise Bean: Test
 */
public class TestBean implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext mySessionCtx;

    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }

    public void setSessionContext(javax.ejb.SessionContext ctx) {
        mySessionCtx = ctx;
    }

    public void ejbCreate() throws javax.ejb.CreateException {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }

    public void allocateMemory(int megabytes, int millisecondsToWait,
        boolean store)
    {
        Vector garbage = new Vector();

        for (int i = 0; i < megabytes; i++)
        {
            try {
                if (store)
                    garbage.addElement(new Garbage());
                else
                    new Garbage();
            }
            catch (OutOfMemoryError e) {
                System.out.println("TestBean.allocateMemory: " +
                    " OutOfMemoryError -  clearing garbage!");

                garbage.clear();

                return;
            }

            try {
                Thread.sleep(millisecondsToWait);
            }
            catch (InterruptedException e) {}
        }

        garbage.clear();
    }
}
```

TestHome.java

```
package ejbs;

/**
 * Home interface for Enterprise Bean: Test
 */
public interface TestHome extends javax.ejb.EJBHome {

    public ejbs.Test create() throws javax.ejb.CreateException,
        java.rmi.RemoteException;
}
```

Garbage.java

```
package ejbs;

public class Garbage {

    private static boolean finalization = false;
    private static int counter = 0;

    private int id;

    Byte[] megabyte = new Byte[1048576];

    Garbage() {
        id = counter++;

        System.out.println("Erzeugung von Objekt " + id);
    }

    public static void setFinalization(boolean finalization) {
        Garbage.finalization = finalization;
    }

    protected void finalize() {
        while (finalization) {
            System.out.println("Ausführung des Finalisierers von Objekt " +
                id);

            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {}
        }
    }
}
```

A.2 Der Denial-of-Service-Testclient

A.2.1 ThesisEJBClient.java

```
import ejbs.*;
import java.awt.*;
import java.rmi.RemoteException;
import java.util.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import javax.swing.*;
import javax.transaction.TransactionRolledbackException;

public class ThesisEJBClient extends JFrame {

    private InitialContext initialContext = null;
    private Integer txs = new Integer(0);
    private JFrame selfReference = this;
    private MemoryThread memoryThread = null;
    private ReadOnlyDefaultTableModel tableModel = null;
    private TestHome testHome = null;
    private ThreadManager threadManager = new ThreadManager();
    private TransferHome transferHome = null;
    private Vector accounts = new Vector();
    private java.util.Timer timer = null;

    // GUI-Elemente

    private JButton jButton = null;
    private JButton jButton1 = null;
    private JButton jButton2 = null;
    private JButton jButton3 = null;
    private JCheckBox jCheckBox = null;
    private JComboBox jComboBox = null;
    private JComboBox jComboBox1 = null;
    private JLabel jLabel = null;
    private JLabel jLabel1 = null;
    private JLabel jLabel2 = null;
    private JLabel jLabel3 = null;
    private JLabel jLabel4 = null;
    private JLabel jLabel5 = null;
    private JLabel jLabel6 = null;
    private JMenu jMenu = null;
    private JMenu jMenu1 = null;
    private JMenu jMenu2 = null;
    private JMenuBar jJMenuBar = null;
    private JMenuItem jMenuItem = null;
    private JMenuItem jMenuItem1 = null;
    private JMenuItem jMenuItem2 = null;
    private JPanel jContentPane = null;
    private JScrollPane jScrollPane = null;
    private JScrollPane jScrollPane1 = null;
    private JTable jTable = null;
    private JTextArea jTextArea = null;
    private PaintPanel jPanel = null;

    // Klasse UpdateTableTask

    private class UpdateTableTask extends TimerTask {

        public void run() {
            updateTable(true);
        }
    }

    // Klasse ThreadManager

    public class ThreadManager {

        private Vector threads = new Vector();
        private boolean running = false;

        public int getThreadAccountNumber(int index) {
            return ((TransferThread)
                threads.elementAt(index)).getAccountNumber();
        }

        public void startThreads() {
```

```
            threads.add(new TransferThread(0, 1000));
            threads.add(new TransferThread(1, 1001));
            threads.add(new TransferThread(2, 1002));
            threads.add(new TransferThread(3, 1003));

            for (int i = 0; i < 2; i++) {
                ((TransferThread) threads.elementAt(i)).start();
                appendText("THREAD " + ((TransferThread)
                    threads.elementAt(i)).getAccountNumber() + " START\n");
            }

            running = true;
        }

        public void stopThreads() {
            if (!running)
                return;

            for (int i = 0; i < 2; i++)
                 ((TransferThread) threads.elementAt(i)).terminate();

            for (int i = 0; i < 2; i++)
                try {
                    ((TransferThread) threads.elementAt(i)).join();
                    appendText("THREAD " + ((TransferThread)
                        threads.elementAt(i)).getAccountNumber() +
                            " STOP\n");
                } catch (InterruptedException e) {
                    System.out.println(
                        "Exception in ThreadManager.stopThreads: " +
                            e.getMessage());

                    System.exit(0);
                }

            threads.clear();

            running = false;
        }
    }

    // Klasse MemoryThread

    private class MemoryThread extends Thread {

        private Test test;
        private int megabytes;
        private int millisecondsToWait;

        MemoryThread(int megabytes, int millisecondsToWait) {
            this.megabytes = megabytes;
            this.millisecondsToWait = millisecondsToWait;
        }

        public void run() {

            try {
                test = testHome.create();
            } catch (Exception e) {
                System.out.println("Exception in MemoryThread.run: " +
                    e.getMessage());

                System.exit(0);
            }

            try {
                appendText("THREAD MEMORY START\n");
                test.allocateMemory(megabytes, millisecondsToWait,
                    jCheckBox.isSelected());
                appendText("THREAD MEMORY STOP\n");
            } catch (Exception e) {
                System.out.println("Exception in MemoryThread.run: " +
                    e.getMessage());

                System.exit(0);
            }
        }
    }

    // Klasse TransferThread
```

```
private class TransferThread extends Thread {

    private Random randomAccount = new Random();
    private Random randomValue = new Random();
    private Transfer transfer;
    private boolean termination = false;
    private int accountIndex;
    private int accountNumber;

    TransferThread(int accountIndex, int accountNumber) {
        this.accountIndex = accountIndex;
        this.accountNumber = accountNumber;

        try {
            transfer = transferHome.create();
        } catch (Exception e) {
            System.out.println(
                "Exception in TransferThread.TransferThread: " +
                    e.getMessage());

            System.exit(0);
        }
    }

    public void run() {
        while (!termination) {
            try {
                transfer.transfer(accountNumber,
                    threadManager.getThreadAccountNumber(accountIndex +
                        2), (randomValue.nextInt(10) + 1) * 10);
            } catch (TransactionRolledbackException e) {
                appendText("THREAD " + accountNumber +
                    " TransactionRolledbackException\n");
            } catch (Exception e) {
                System.out.println("Exception in TransferThread.run: " +
                    e.getMessage());

                break;
            }

            synchronized (txs) {
                txs = new Integer(txs.intValue() + 1);
            }
        }

        termination = false;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public void terminate() {
        termination = true;
    }
}

public static void main(String[] args) {
    ThesisEJBClient thesisEJBClient = new ThesisEJBClient();
    thesisEJBClient.show();
}

/**
 * This is the default constructor
 */
public ThesisEJBClient() {
    super();
    initialize();

    String serverName = System.getProperty("serverName", "");

    if (serverName.equals("")) {
        JOptionPane.showMessageDialog(selfReference,
            "Die Systemeigenschaft 'serverName' ist nicht gesetzt!",
                "Hinweis", JOptionPane.ERROR_MESSAGE);

        System.exit(0);
    }
```

```
// InitialContext

Properties props = new Properties();
props.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.ibm.websphere.naming.WsnInitialContextFactory");
props.put(Context.PROVIDER_URL, "iiop://" + serverName + ":2809");

try {
    initialContext = new InitialContext(props);
} catch (NamingException e) {
    System.out.println(
        "Exception in ThesisEJBClient.ThesisEJBClient: " +
            e.getMessage());

    System.exit(0);
}

Object ref1 = null;
Object ref2 = null;
Object ref3 = null;

try {
    ref1 = initialContext.lookup("ejb/ejbs/AccountHome");
    ref2 = initialContext.lookup("ejb/ejbs/TestHome");
    ref3 = initialContext.lookup("ejb/ejbs/TransferHome");
} catch (NamingException e) {
    JOptionPane.showMessageDialog(selfReference,
        "Der Applikationsserver ist nicht verfügbar!", "Hinweis",
            JOptionPane.ERROR_MESSAGE);

    System.exit(0);
}

AccountHome accountHome = (AccountHome)
    PortableRemoteObject.narrow(ref1, AccountHome.class);
testHome = (TestHome) PortableRemoteObject.narrow(ref2,
    TestHome.class);
transferHome = (TransferHome) PortableRemoteObject.narrow(ref3,
    TransferHome.class);

try {
    accounts.add(accountHome.findByPrimaryKey(
        new AccountKey(1000)));
    accounts.add(accountHome.findByPrimaryKey(
        new AccountKey(1001)));
    accounts.add(accountHome.findByPrimaryKey(
        new AccountKey(1002)));
    accounts.add(accountHome.findByPrimaryKey(
        new AccountKey(1003)));
} catch (Exception e) {
    System.out.println(
        "Exception in ThesisEJBClient.ThesisEJBClient: " +
            e.getMessage());

    System.exit(0);
}

appendText("EJB-Initialisierung erfolgreich!\n");

// GUI

tableModel = new ReadOnlyDefaultTableModel();
tableModel.addColumn("Kontonummer");
tableModel.addColumn("Saldo");

for (int i = 0; i < accounts.size(); i++) {
    Account account = (Account) accounts.elementAt(i);
    Vector row = new Vector();

    try {
        row.add(String.valueOf(((AccountKey)
            account.getPrimaryKey()).id));
        row.add(String.valueOf(account.getValue()));
    } catch (RemoteException e) {
        System.out.println(
            "Exception in ThesisEJBClient.ThesisEJBClient: " +
                e.getMessage());

        System.exit(0);
    }
```

```
            tableModel.addRow(row);
        }

        jTable.setModel(tableModel);
    }

    private void appendText(String text) {
        jTextArea.append("- " + text);

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                jTextArea.setCaretPosition(jTextArea.getText().length());
            }
        });
    }

    private void stopThreads() {
        threadManager.stopThreads();
    }

    private void updateTable(boolean updatePanel) {
        UpdateGUIRunnable updateGUIRunnable = null;

        if (updatePanel)
        {
            int value = txs.intValue();

            synchronized (txs) {
                txs = new Integer(0);
            }

            updateGUIRunnable = new UpdateGUIRunnable(tableModel, jLabel2,
                jPanel, value);
        }
        else
            updateGUIRunnable = new UpdateGUIRunnable(tableModel);

        for (int i = 0; i < accounts.size(); i++) {
            Account account = (Account) accounts.elementAt(i);
            String value = "";

            try {
                value = String.valueOf(account.getValue());
            } catch (RemoteException e) {
                System.out.println(
                    "Exception in ThesisEJBClient.updateTable: " +
                        e.getMessage());

                System.exit(0);
            }

            updateGUIRunnable.addTableDate(new TableDate(value, i, 1));
        }

        SwingUtilities.invokeLater(updateGUIRunnable);
    }

    /**
     * This method initializes this
     *
     * @return void
     */
    private void initialize() {
        Dimension dimension = Toolkit.getDefaultToolkit().getScreenSize();

        this.setSize(630, 650);
        this.setContentPane(getJContentPane());
        this.setJMenuBar(getJJMenuBar());
        this.setIconImage(
            java.awt.Toolkit.getDefaultToolkit().getImage(
                System.getProperties().getProperty("user.dir") +
                    "/java.gif"));
        this.setTitle(
            "Diplomarbeit Jens Müller - Denial-of-Service-Testclient");
        this.setResizable(false);
        this.setLocation(
            (dimension.width - this.getSize().width) / 2,
            (dimension.height - this.getSize().height) / 2);
        this.addWindowListener(new java.awt.event.WindowAdapter() {
```

```
            public void windowClosing(java.awt.event.WindowEvent e) {
                if (memoryThread != null)
                {
                    try {
                        memoryThread.join();
                    }
                    catch (InterruptedException exception) {}
                }

                stopThreads();

                System.exit(0);
            }
        });
    }
    /**
     * This method initializes jPanel
     *
     * @return javax.swing.JPanel
     */
    private PaintPanel getJPanel() {
        if (jPanel == null) {
            jPanel = new PaintPanel();
            jPanel.setSize(290, 151);
            jPanel.setLocation(300, 230);
            jPanel.setFont(
                new java.awt.Font("Courier New", java.awt.Font.PLAIN, 12));
        }
        return jPanel;
    }
    /**
     * This method initializes jButton
     *
     * @return javax.swing.JButton
     */
    private javax.swing.JButton getJButton() {
        if (jButton == null) {
            jButton = new javax.swing.JButton();
            jButton.setSize(250, 20);
            jButton.setLocation(25, 50);
            jButton.setText("Überweisungsthreads starten");
            jButton.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent e) {
                    jButton.setEnabled(false);
                    jButton2.setEnabled(false);

                    timer = new java.util.Timer();
                    timer.schedule(new UpdateTableTask(), 0, 5000);

                    threadManager.startThreads();

                    jButton1.setEnabled(true);
                }
            });
        }
        return jButton;
    }
    /**
     * This method initializes jButton1
     *
     * @return javax.swing.JButton
     */
    private javax.swing.JButton getJButton1() {
        if (jButton1 == null) {
            jButton1 = new javax.swing.JButton();
            jButton1.setSize(250, 20);
            jButton1.setLocation(25, 80);
            jButton1.setText("Überweisungsthreads stoppen");
            jButton1.setEnabled(false);
            jButton1.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent e) {
                    jButton1.setEnabled(false);

                    stopThreads();

                    timer.cancel();
                    updateTable(false);

                    jLabel2.setText("-");
                    jPanel.reset();
```

```
                jButton.setEnabled(true);
                jButton2.setEnabled(true);
            }
        });
    }
    return jButton1;
}
/**
 * This method initializes jButton2
 *
 * @return javax.swing.JButton
 */
private javax.swing.JButton getJButton2() {
    if (jButton2 == null) {
        jButton2 = new javax.swing.JButton();
        jButton2.setSize(250, 20);
        jButton2.setLocation(25, 110);
        jButton2.setText("Saldi zurücksetzen");
        jButton2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                for (int i = 0; i < accounts.size(); i++) {
                    Account account = (Account) accounts.elementAt(i);

                    try {
                        account.setValue(10000000);
                    } catch (RemoteException exception) {
                        System.out.println(
                            "Exception in getJButton2(): " +
                                exception.getMessage());

                        System.exit(0);
                    }

                    updateTable(false);
                }
            }
        });
    }
    return jButton2;
}
/**
 * This method initializes jButton3
 *
 * @return javax.swing.JButton
 */
private javax.swing.JButton getJButton3() {
    if (jButton3 == null) {
        jButton3 = new javax.swing.JButton();
        jButton3.setSize(250, 20);
        jButton3.setLocation(26, 360);
        jButton3.setText("Denial-of-Service-Angriff starten");
        jButton3.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                int v1 = Integer.parseInt((String)
                    jComboBox.getSelectedItem());
                int v2 = Integer.parseInt((String)
                    jComboBox1.getSelectedItem());

                memoryThread = new MemoryThread(v1, v2);
                memoryThread.start();
            }
        });
    }
    return jButton3;
}
/**
 * This method initializes jCheckBox
 *
 * @return javax.swing.JCheckBox
 */
private javax.swing.JCheckBox getJCheckBox() {
    if (jCheckBox == null) {
        jCheckBox = new javax.swing.JCheckBox();
        jCheckBox.setSize(250, 21);
        jCheckBox.setText("Objekte zwischenspeichern");
        jCheckBox.setLocation(26, 320);
    }
    return jCheckBox;
}
```

```
    /**
     * This method initializes jComboBox
     *
     * @return javax.swing.JComboBox
     */
    private javax.swing.JComboBox getJComboBox() {
        if (jComboBox == null) {
            jComboBox = new javax.swing.JComboBox();
            jComboBox.setSize(120, 24);
            jComboBox.setLocation(155, 230);
        }

        jComboBox.addItem("10");
        jComboBox.addItem("20");
        jComboBox.addItem("40");
        jComboBox.addItem("60");
        jComboBox.addItem("80");
        jComboBox.addItem("100");
        jComboBox.addItem("200");
        jComboBox.addItem("300");
        jComboBox.addItem("400");
        jComboBox.addItem("500");
        jComboBox.addItem("1000");

        return jComboBox;
    }
    /**
     * This method initializes jComboBox1
     *
     * @return javax.swing.JComboBox
     */
    private javax.swing.JComboBox getJComboBox1() {
        if (jComboBox1 == null) {
            jComboBox1 = new javax.swing.JComboBox();
            jComboBox1.setSize(120, 24);
            jComboBox1.setLocation(155, 280);
        }

        jComboBox1.addItem("50");
        jComboBox1.addItem("25");
        jComboBox1.addItem("10");
        jComboBox1.addItem("5");
        jComboBox1.addItem("0");

        return jComboBox1;
    }
    /**
     * This method initializes jLabel
     *
     * @return javax.swing.JLabel
     */
    private javax.swing.JLabel getJLabel() {
        if (jLabel == null) {
            jLabel = new javax.swing.JLabel();
            jLabel.setSize(100, 10);
            jLabel.setText("Menü");
            jLabel.setLocation(25, 30);
        }
        return jLabel;
    }
    /**
     * This method initializes jLabel1
     *
     * @return javax.swing.JLabel
     */
    private javax.swing.JLabel getJLabel1() {
        if (jLabel1 == null) {
            jLabel1 = new javax.swing.JLabel();
            jLabel1.setSize(180, 15);
            jLabel1.setText("Transaktionen pro Sekunde:");
            jLabel1.setLocation(300, 200);
        }
        return jLabel1;
    }
    /**
     * This method initializes jLabel2
     *
     * @return javax.swing.JLabel
     */
    private javax.swing.JLabel getJLabel2() {
```

```
        if (jLabel2 == null) {
            jLabel2 = new javax.swing.JLabel();
            jLabel2.setSize(50, 15);
            jLabel2.setText("-");
            jLabel2.setLocation(490, 200);
            jLabel2.setFont(
            new java.awt.Font("Courier New", java.awt.Font.PLAIN, 12));
        }
        return jLabel2;
    }
    /**
     * This method initializes jLabel3
     *
     * @return javax.swing.JLabel
     */
    private javax.swing.JLabel getJLabel3() {
        if (jLabel3 == null) {
            jLabel3 = new javax.swing.JLabel();
            jLabel3.setSize(120, 15);
            jLabel3.setText("Anzufordernde");
            jLabel3.setLocation(26, 230);
        }
        return jLabel3;
    }
    /**
     * This method initializes jLabel4
     *
     * @return javax.swing.JLabel
     */
    private javax.swing.JLabel getJLabel4() {
        if (jLabel4 == null) {
            jLabel4 = new javax.swing.JLabel();
            jLabel4.setSize(120, 15);
            jLabel4.setText("Speichermenge");
            jLabel4.setLocation(26, 245);
        }
        return jLabel4;
    }
    /**
     * This method initializes jLabel5
     *
     * @return javax.swing.JLabel
     */
    private javax.swing.JLabel getJLabel5() {
        if (jLabel5 == null) {
            jLabel5 = new javax.swing.JLabel();
            jLabel5.setSize(120, 15);
            jLabel5.setText("Wartezeit pro");
            jLabel5.setLocation(26, 280);
        }
        return jLabel5;
    }
    /**
     * This method initializes jLabel6
     *
     * @return javax.swing.JLabel
     */
    private javax.swing.JLabel getJLabel6() {
        if (jLabel6 == null) {
            jLabel6 = new javax.swing.JLabel();
            jLabel6.setSize(120, 15);
            jLabel6.setText("MB in Millisekunden");
            jLabel6.setLocation(26, 295);
        }
        return jLabel6;
    }
    /**
     * This method initializes jMenu
     *
     * @return javax.swing.JMenu
     */
    private javax.swing.JMenu getJMenu() {
        if (jMenu == null) {
            jMenu = new javax.swing.JMenu();
            jMenu.add(getJMenuItem());
            jMenu.setText("Datei");
        }
        return jMenu;
    }
    /**
```

```
     * This method initializes jJMenuBar
     *
     * @return javax.swing.JMenuBar
     */
    private javax.swing.JMenuBar getJJMenuBar() {
        if (jJMenuBar == null) {
            jJMenuBar = new javax.swing.JMenuBar();
            jJMenuBar.add(getJMenu());
        }
        return jJMenuBar;
    }
    /**
     * This method initializes jMenuItem
     *
     * @return javax.swing.JMenuItem
     */
    private javax.swing.JMenuItem getJMenuItem() {
        if (jMenuItem == null) {
            jMenuItem = new javax.swing.JMenuItem();
            jMenuItem.setText("Beenden");
            jMenuItem.addActionListener(new
                java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent e) {
                    stopThreads();

                    System.exit(0);
                }
            });
        }
        return jMenuItem;
    }
    /**
     * This method initializes jContentPane
     *
     * @return javax.swing.JPanel
     */
    private javax.swing.JPanel getJContentPane() {
        if (jContentPane == null) {
            jContentPane = new javax.swing.JPanel();
            jContentPane.setLayout(null);
            jContentPane.add(getJButton(), null);
            jContentPane.add(getJButton1(), null);
            jContentPane.add(getJButton2(), null);
            jContentPane.add(getJButton3(), null);
            jContentPane.add(getJCheckBox(), null);
            jContentPane.add(getJComboBox(), null);
            jContentPane.add(getJComboBox1(), null);
            jContentPane.add(getJLabel(), null);
            jContentPane.add(getJLabel1(), null);
            jContentPane.add(getJLabel2(), null);
            jContentPane.add(getJLabel3(), null);
            jContentPane.add(getJLabel4(), null);
            jContentPane.add(getJLabel5(), null);
            jContentPane.add(getJLabel6(), null);
            jContentPane.add(getJPanel(), null);
            jContentPane.add(getJScrollPane(), null);
            jContentPane.add(getJScrollPane1(), null);
        }
        return jContentPane;
    }
    /**
     * This method initializes jScrollPane
     *
     * @return javax.swing.JScrollPane
     */
    private javax.swing.JScrollPane getJScrollPane() {
        if (jScrollPane == null) {
            jScrollPane = new javax.swing.JScrollPane();
            jScrollPane.setViewportView(getJTable());
            jScrollPane.setSize(290, 130);
            jScrollPane.setLocation(300, 50);
        }
        return jScrollPane;
    }
    /**
     * This method initializes jScrollPane1
     *
     * @return javax.swing.JScrollPane
     */
    private javax.swing.JScrollPane getJScrollPane1() {
```

```
        if (jScrollPane1 == null) {
            jScrollPane1 = new javax.swing.JScrollPane();
            jScrollPane1.setViewportView(getJTextArea());
            jScrollPane1.setSize(565, 180);
            jScrollPane1.setLocation(25, 400);
        }
        return jScrollPane1;
    }
    /**
     * This method initializes jTable
     *
     * @return javax.swing.JTable
     */
    private javax.swing.JTable getJTable() {
        if (jTable == null) {
            jTable = new javax.swing.JTable();
            jTable.setRowSelectionAllowed(false);
            jTable.setFont(new java.awt.Font("Courier New",
                java.awt.Font.PLAIN, 12));
        }
        return jTable;
    }
    /**
     * This method initializes jTextArea
     *
     * @return javax.swing.JTextArea
     */
    private javax.swing.JTextArea getJTextArea() {
        if (jTextArea == null) {
            jTextArea = new JTextArea();
            jTextArea.setFont(new java.awt.Font("Courier New",
                java.awt.Font.PLAIN, 12));
        }
        return jTextArea;
    }
}
```

A.2.2 PaintPanel.java

```
import java.awt.*;
import javax.swing.JPanel;

public class PaintPanel extends JPanel {

    private int values[] = new int[30];
    private int y = 0;

    public PaintPanel() {
        super();

        for (int i = 0; i < 30; i++)
            values[i] = 0;
    }

    public void paint(Graphics g) {
        super.paint(g);

        g.setColor(Color.black);
        g.fillRect(0, 0, getWidth() - 1, getHeight() - 1);

        g.setColor(Color.lightGray);
        g.drawLine(0, getHeight() / 2 - 1, getWidth() - 1, getHeight() / 2
            - 1);

        g.setColor(Color.green);

        for (int i = 0; i < 29; i++)
            g.drawLine(i * 10, getHeight() - values[i] - 1, i * 10 + 10,
                getHeight() - values[i + 1] - 1);

        g.setColor(Color.lightGray);

        g.drawString("0", 5, getHeight() - 5);
        g.drawString("150", 5, getHeight() / 2 + 15);
        g.drawString("300", 5, 15);
    }

    public void updatePanel(int y) {
        for (int i = 0; i < 29; i++)
            values[i] = values[i + 1];

        values[29] = y / 2;

        repaint();
    }

    public void reset() {
        for (int i = 0; i < 30; i++)
            values[i] = 0;

        repaint();
    }
}
```

A.2.3 ReadOnlyDefaultTableModel.java

```
import javax.swing.table.*;

public class ReadOnlyDefaultTableModel extends DefaultTableModel {

    public boolean isCellEditable(int arg0, int arg1) {
        return false;
    }
}
```

A.2.4 TableDate.java

```
public class TableDate {

    private String value;
    private int column;
    private int row;

    public TableDate(String value, int row, int column) {
        this.value = value;
        this.row = row;
        this.column = column;
    }

    public String getValue() {
        return value;
    }

    public int getRow() {
        return row;
    }

    public int getColumn() {
        return column;
    }
}
```

A.2.5 UpdateGUIRunnable.java

```
import java.util.*;
import javax.swing.*;

public class UpdateGUIRunnable implements Runnable {

    private JLabel jLabel = null;
    private PaintPanel jPanel = null;
    private ReadOnlyDefaultTableModel tableModel = null;
    private Vector tableData = new Vector();
    private boolean panel;
    private int value;

    public UpdateGUIRunnable(ReadOnlyDefaultTableModel tableModel) {
        this.tableModel = tableModel;

        panel = false;
    }

    public UpdateGUIRunnable(ReadOnlyDefaultTableModel tableModel,
        JLabel jLabel, PaintPanel jPanel, int value) {
        this.tableModel = tableModel;
        this.jLabel = jLabel;
        this.jPanel = jPanel;
        this.value = value;

        panel = true;
    }

    public void addTableDate(TableDate tableDate) {
        tableData.add(tableDate);
    }

    public void run() {
        for (int i = 0; i < tableData.size(); i++) {
            TableDate tableDate = (TableDate)tableData.elementAt(i);
            tableModel.setValueAt(tableDate.getValue(), tableDate.getRow(),
                tableDate.getColumn());
        }

        if (panel) {
            jLabel.setText(new Integer(value / 5).toString());
            jPanel.updatePanel(value / 5);
        }
    }
}
```

Anhang B

DVD zur Diplomarbeit

B.1 Inhalt der DVD

- **Verzeichnis \Diplomarbeit:**
  Enthält die Diplomarbeit als PDF-Dokument.

- **Verzeichnis \Literatur:**
  Enthält den Großteil der benutzten Literatur in Form von PDF- und PostScript-Dokumenten.

- **Verzeichnis \Navigation:**
  Enthält das Inhaltsverzeichnis der DVD als HTML-Dokument (`index.html`).

- **Verzeichnis \Software:**
  Enthält die verwendete Software für Windows (Trial-Versionen):
  - IBM DB2 Universal Database Enterprise Server Edition 8.2
  - IBM Rational Application Developer 6.0
  - IBM WebSphere Application Server 6.0

- **Verzeichnis \Source:**
  Enthält den Quellcode der im Rahmen der Diplomarbeit geschriebenen Software.

Anwendungs- und Transaktionsisolation unter Java
Diplomarbeit

Betreut von Prof. Dr.-Ing. Wilhelm G. Spruth

Eberhard Karls Universität Tübingen
Fakultät für Informations- und Kognitionswissenschaften
Wilhelm-Schickard-Institut für Informatik

Vorgelegt von Jens Müller (Matrikelnummer 2200565)

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Literaturverzeichnis

Alta - K0		
CPU Inheritance Scheduling	Quelle	
Microkernels Meet Recursive Virtual Machines	Quelle	
Nested Java Processes: OS Structure for Mobile Code	Quelle	
Java Operating Systems: Design and Implementation	Quelle	
Drawing the Red Line in Java	Quelle	
The Alta Operating System	Quelle	
Techniques for the Design of Java Operating Systems	Quelle	
Balfanz und Gong		
Experience with Secure Multi-Processing in Java	Quelle	
Class Loader		
Dynamic Class Loading in the Java Virtual Machine	Quelle	
Echidna		
Echidna User Guide For Echidna 0.4.0	Quelle	
Isolates		
Isolates: A New Approach to Multi-Programming in Java Platforms	Quelle	
JanosVM		
Janos: A Java-oriented OS for Active Network Nodes	Quelle	
JanosVM User's Manual and Tutorial	Quelle	
J-Kernel		
Implementing Multiple Protection Domains in Java	Quelle	
J-Kernel: a Capability-Based Operating System for Java	Quelle	
KVM		
J2ME Building Blocks for Mobile Devices	Quelle	
KaffeOS		
Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java	Quelle	
Isolation, Resource Management and Sharing in the KaffeOS Java Runtime System	Quelle	
Luna		
Tasks and Revocation for Java (or, Hey! You got your Operating System in my Language!)	Quelle	
Adding Operating System Structure to Language-Based Protection	Quelle	
Luna: a Flexible Java Protection System	Quelle	
Marmot: An Optimizing Compiler for Java	Quelle	
Type System Support for Dynamic Revocation	Quelle	
Object Management Group		
Transaction Service Specification 1.2.1	Quelle	

Abbildung B.1: Das Literaturverzeichnis auf DVD

Abkürzungsverzeichnis

ABAP	Advanced Business Application Programming
ACID	Atomicity Consistency Isolation Durability
AIX	Advanced Interactive Executive
API	Application Programming Interface
AWT	Abstract Window Toolkit
CICS	Customer Information Control System
COM+	Component Object Model +
CORBA	Common Object Request Broker Architecture
CRM	Communication Resource Manager
DCOM	Distributed Component Object Model
EIS	Enterprise Information System
EJB	Enterprise JavaBeans
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines
IIOP	Internet Inter-Orb Protocol
J2EE	Java 2 Platform, Enterprise Edition
J2ME	Java 2 Platform, Micro Edition
J2SE	Java 2 Platform, Standard Edition
JCA	J2EE Connector Architecture
JDBC	Java Database Connectivity
JDK	J2SE Development Kit
JMS	Java Message Service
JNI	Java Native Interface
JSP	Java Server Pages
JTA	Java Transaction API
JTS	Java Transaction Service
JVM	Java Virtual Machine
KVM	K Virtual Machine
MVM	Multi-Tasking Virtual Machine
ORB	Object Request Broker
OTM	Object Transaction Monitor
OTS	Object Transaction Service
PRJVM	Persistent Reusable Java Virtual Machine
RMI	Remote Method Invocation
RMI-IIOP	Remote Method Invocation over Internet Inter-Orb Protocol
RPC	Remote Procedure Call
SAP	Systeme, Anwendungen, Produkte in der Datenverarbeitung
XIMI	Cross-Isolate Method Invocation
XML	Extended Markup Language

Literaturverzeichnis

[Bac02] Back, G.: *Isolation, Resource Management and Sharing in the KaffeOS Java Runtime System*, University of Utah, Diss., 2002. – http://www.cs.utah.edu/flux/papers/back-thesis-base.html

[BG97] Balfanz, D. ; Gong, L.: Experience with Secure Multi-Processing in Java / Department of Computer Science, Princeton University. 1997 (560-97). – Forschungsbericht. http://www.cs.princeton.edu/sip/pub/icdcs.php3

[BH99] Back, G. ; Hsieh, W.: Drawing the Red Line in Java. In: *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, 1999, S. 116–121. – http://www.cs.utah.edu/flux/papers/redline-hotos7-base.html

[BHL00] Back, G. ; Hsieh, W. ; Lepreau, J.: Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In: *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, 2000, S. 333–346. – http://www.usenix.org/publications/library/proceedings/osdi2000/back.html

[BPW+00] Borman, S. ; Paice, S. ; Webster, M. ; Trotter, M. ; McGuire, R. ; Stevens, A. ; Hutchison, B. ; Berry, R.: A Serially Reusable Java Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing / IBM Hursley. 2000 (TR 29.3406). – Forschungsbericht. http://www.ibm.com/servers/eserver/zseries/software/java/pdf/29.3406.pdf

[Bry04] Bryce, C.: Isolates: A New Approach to Multi-Programming in Java Platforms. In: *OTLand Experts' Corner* (2004). – http://www.bitser.net/isolate-interest/papers/bryce-05.04.pdf

[BTS+98] Back, G. ; Tullmann, P. ; Stoller, L. ; Hsieh, W. ; Lepreau, J.: Java Operating Systems: Design and Implementation / School of Computing, University of Utah. 1998 (UUCS-98-015). – Forschungsbericht. http://www.cs.utah.edu/flux/papers/javaos-tr98015-abs.html

[BTS+00] Back, G. ; Tullmann, P. ; Stoller, L. ; Hsieh, W. ; Lepreau, J.: Techniques for the Design of Java Operating Systems. In: *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000, S. 197–210. – http://www.usenix.org/publications/library/proceedings/usenix2000/general/back.html

[CCH+98] Czajkowski, G. ; Chang, C. ; Hawblitzel, C. ; Hu, D. ; Eicken, T. von: Resource Management for Extensible Internet Servers. In: *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, 1998, S. 33–39. – http://www.cs.cornell.edu/slk/papers/sigops98.pdf

[CD01] Czajkowski, G. ; Daynès, L.: Multitasking without Compromise: a Virtual Machine Evolution. In: *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York : ACM Press, 2001, S. 125–138. – http://research.sun.com/projects/barcelona/papers/oopsla01.pdf

[CDN02] Czajkowski, G. ; Daynès, L. ; Nystrom, N.: Code Sharing among Virtual Machines. In: *Proceedings of the 16th European Conference on Object-Oriented Programming*. London : Springer-Verlag, 2002, S. 155–177. – http://research.sun.com/projects/barcelona/papers/ecoop02.pdf

[CDT03] Czajkowski, G. ; Daynès, L. ; Titzer, B.: A Multi-User Virtual Machine. In: *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003, S. 85–98. – http://www.usenix.org/events/usenix03/tech/full_papers/czajkowski/czajkowski.pdf

[CDW01] Czajkowski, G. ; Daynès, L. ; Wolczko, M.: Automated and Portable Native Code Isolation / Sun Microsystems, Inc. 2001 (TR-2001-96). – Forschungsbericht. http://research.sun.com/projects/barcelona/papers/TR_2001_96.pdf

[CE98] Czajkowski, G. ; Eicken, T. von: JRes: A Resource Accounting Interface for Java. In: *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York : ACM Press, 1998, S. 21–35. – http://www.cs.cornell.edu/slk/papers/oopsla98.pdf

[CFM+97] Cramer, T. ; Friedman, R. ; Miller, T. ; Seberger, D. ; Wilson, R. ; Wolczko, M.: Compiling Java Just in Time. In: *IEEE Micro* 17 (1997), S. 36–43. – http://portal.acm.org/citation.cfm?id=624097

[CHS+03] Czajkowski, G. ; Hahn, S. ; Skinner, G. ; Soper, P. ; Bryce, C.: A Resource Management Interface for the Java Platform / Sun Microsystems, Inc. 2003 (TR-2003-124). – Forschungsbericht. http://research.sun.com/techrep/2003/smli_tr-2003-124.pdf

[CHSS02] Czajkowski, G. ; Hahn, S. ; Skinner, G. ; Soper, P.: *Resource Consumption Interfaces for Java Application Programming - a Proposal*. 2002. – http://www.ovmj.org/workshops/resman/resman02.tar.gz

[Cza00] Czajkowski, G.: Application Isolation in the Java Virtual Machine. In: *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York : ACM Press, 2000, S. 354–366. – http://research.sun.com/projects/barcelona/papers/oopsla00.pdf

[DBC+00] Dillenberger, D. ; Bordawekar, R. ; Clark, C. ; Durand, D. ; Emmes, D. ; Gohda, O. ; Howard, S. ; Oliver, M. ; Samuel, F. ; John, R. S.: Building a Java virtual machine for server applications: The Jvm on OS/390. In: *IBM Systems Journal* 39 (2000), Nr. 1, S. 194–210. – http://www.research.ibm.com/journal/sj/391/dillenberger.pdf

[ECC+99] Eicken, T. von ; Chang, C. ; Czajkowski, G. ; Hawblitzel, C. ; Hu, D. ; Spoonhower, D.: J-Kernel: a Capability-Based Operating System for Java. In: Vitek, J. (Hrsg.) ; Jensen, C. (Hrsg.): *Secure Internet Programming - Security Issues for Mobile and Distributed Objects*. Berlin : Springer-Verlag, 1999, S. 369–394

[FHL+96] Ford, B. ; Hibler, M. ; Lepreau, J. ; Tullmann, P. ; Back, G. ; Clawson, S.: Microkernels Meet Recursive Virtual Machines. In: *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, 1996, S. 137–151. – http://www.usenix.org/publications/library/proceedings/osdi96/hibler.html

[FKR+99] Fitzgerald, R. ; Knoblock, T. ; Ruf, E. ; Steensgaard, B. ; Tarditi, D.: Marmot: An Optimizing Compiler for Java / Microsoft Research. 1999 (MSR-TR-99-33). – Forschungsbericht. http://research.microsoft.com/research/pubs/view.aspx?tr_id=267

[Flu03a] Flux Research Group: *JanosVM User's Manual and Tutorial*, 2003. – http://www.cs.utah.edu/flux/janos/janosvm-1.0/janosvm-1.0-manual/manual.ps

[Flu03b] Flux Research Group: *Pressemitteilung: JanosVM v1.0*. 2003. – http://www.cs.utah.edu/flux/janos/janosvm-1.0/ANNOUNCE

[FS96] Ford, B. ; Susarla, S.: CPU Inheritance Scheduling. In: *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, 1996, S. 91–105. – http://www.usenix.org/publications/library/proceedings/osdi96/ford.html

[GJSB05] Gosling, J. ; Joy, B. ; Steele, G. ; Bracha, G.: *The Java Language Specification*. 3. Auflage. Boston : Addison Wesley, 2005. – http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

[Gor99] Gorrie, Luke: *Echidna User Guide For Echidna 0.4.0*, 1999. – http://www.javagroup.org/echidna/EchidnaUserGuide.pdf

[GR93] Gray, J. ; Reuter, A.: *Transaction Processing: Concepts and Techniques*. 1. Auflage. San Francisco : Morgan Kaufmann, 1993

[Haw00] Hawblitzel, C.: *Adding Operating System Structure to Language-Based Protection*, Cornell University, Diss., 2000. – http://www.cs.dartmouth.edu/~hawblitz/publish/hawblitzel-thesis.pdf

[HCC+98] Hawblitzel, C. ; Chang, C. ; Czajkowski, G. ; Hu, D. ; Eicken, T. von: Implementing Multiple Protection Domains in Java. In: *Proceedings of the 1998 USENIX Annual Technical Conference*, 1998, S. 259–270. – http://www.usenix.org/publications/library/proceedings/usenix98/hawblitzel.html

[HE98] Hawblitzel, C. ; Eicken, T. von: A Case for Language-Based Protection / Department of Computer Science, Cornell University. 1998 (98-1670). – Forschungsbericht. http://www.cs.cornell.edu/slk/papers/TR98-1670.pdf

[HE99a] Hawblitzel, C. ; Eicken, T. von: *Tasks and Revocation for Java (or, Hey! You got your Operating System in my Language!)*. 1999. – http://www.cs.dartmouth.edu/~hawblitz/publish/luna-99-11-13.ps

[HE99b] Hawblitzel, C. ; Eicken, T. von: Type System Support for Dynamic Revocation. In: *Proceedings of the ACM SIGPLAN 1999 Workshop on Compiler Support for System Software*, INRIA, 1999, S. 72–81. – http://www.cs.dartmouth.edu/~hawblitz/publish/Wcsss99.ps

[HE02] Hawblitzel, C. ; Eicken, T. von: Luna: a Flexible Java Protection System. In: *Proceedings of the USENIX 5th Symposium on Operating Systems Design and Implementation*, 2002, S. 391–403. – http://www.usenix.org/events/osdi02/tech/hawblitzel.html

[Hei05] Heiss, J.: The Multi-Tasking Virtual Machine: Building a Highly Scalable JVM. In: *Sun Technical Articles and Tips* (2005). – http://java.sun.com/developer/technicalArticles/Programming/mvm/

[IBM03] IBM Corp.: *Persistent Reusable Java Virtual Machine User's Guide*. 5. Auflage. IBM, 2003. – http://www.ibm.com/servers/eserver/zseries/software/java/pdf/prjvm14.pdf

[IBM04] IBM Corp.: *IBM WebSphere Application Server, Version 6: Securing applications and their environment*. 2004. – ftp://ftp.software.ibm.com/software/webserver/appserv/library/v60/wasv600base_security.pdf

[JCKS04] Jordan, M. ; Czajkowski, G. ; Kouklinski, K. ; Skinner, G.: Extending a J2EE Server with Dynamic and Flexible Resource Management. In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. New York : Springer Verlag, 2004, S. 439–458. – http://research.sun.com/projects/barcelona/papers/middleware04.pdf

[JCP02] JCP (Java Community Process): *JSR-000121 Application Isolation API Specification (Public Review Draft)*. 2002. – http://jcp.org/aboutJava/communityprocess/review/jsr121/

[JDC+04] Jordan, M. ; Daynès, L. ; Czajkowski, G. ; Jarzab, M. ; Bryce, C.: Scaling J2EE Application Servers with the Multi-Tasking Virtual Machine / Sun Microsystems, Inc. 2004 (TR-2004-135). – Forschungsbericht. http://research.sun.com/techrep/2004/abstract-135.html

[KKL+02] Kuck, N. ; Kuck, H. ; Lott, E. ; Rohland, C. ; Schmidt, O.: SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers / SAP AG. 2002. – Forschungsbericht. http://www.bitser.net/isolate-interest/papers/PAVM.pdf

[KP00] Krause, J. ; Plattner, B.: Safe Class Sharing among Java Processes / IBM Research, Zurich Research Laboratory. 2000. – Forschungsbericht. http://www.zurich.ibm.com/pdf/rz3230.pdf

[LB98] Liang, S. ; Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York : ACM Press, 1998, S. 36–44. – http://portal.acm.org/citation.cfm?id=286945

[Lev84] Levy, H.: *Capability-Based Computer Systems*. 1. Auflage. Bedford : Digital Press, 1984. – http://www.cs.washington.edu/homes/levy/capabook/

[OMG01] Object Management Group, Inc.: *Transaction Service Specification 1.2.1*. 2001. – http://www.omg.org/cgi-bin/doc?formal/2001-11-03

[Par04] Parnas, D.: SAP Virtual Machine Container (as seen on TechEd). In: *SAP Developer Network Weblogs* (2004). – https://www.sdn.sap.com/sdn/weblogs.sdn?blog=/pub/wlg/940

[PCDV02] Palacz, K. ; Czajkowskiy, G. ; Daynès, L. ; Vitek, J.: Incommunicado: Efficient Communication for Isolates. In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York : ACM Press, 2002, S. 262–274. – http://research.sun.com/projects/barcelona/papers/oopsla02.pdf

[SGG03] Silberschatz, A. ; Galvin, P. ; Gagne, G.: *Operating System Concepts*. 6. Auflage. New York : John Wiley & Sons, Inc., 2003

[SM97] Sun Microsystems, Inc.: *Pressemitteilung: SUN DEBUTS THE JAVA PLATFORM FOR THE ENTERPRISE*. 1997. – http://www.sun.com/smi/Press/sunflash/1997-04/sunflash.970402.4383.html

[SM99] Sun Microsystems, Inc.: *Java Transaction Service (JTS) Version 1.0*. 1999. – http://java.sun.com/products/jts/

[SM00] Sun Microsystems, Inc.: *J2ME Building Blocks for Mobile Devices*. 2000. – http://java.sun.com/products/cldc/wp/KVMwp.pdf

[SM01] Sun Microsystems, Inc.: *J2EE Connector Architecture Specification Version 1.0*. http://java.sun.com/j2ee/connector/download.html. Version: 2001

[SM02] Sun Microsystems, Inc.: *Java Transaction API (JTA) Version 1.0.1B*. 2002. – http://java.sun.com/products/jta/

[SM03a] Sun Microsystems, Inc.: *Enterprise JavaBeans Specification Version 2.1*. 2003. – http://java.sun.com/products/ejb/docs.html

[SM03b] Sun Microsystems, Inc.: *Java 2 Platform Enterprise Edition Specification Version 1.4*. 2003. – http://java.sun.com/j2ee/1.4/download.html

[SM04] Sun Microsystems, Inc.: The Multitasking Virtual Machine. In: *Sun Inner Circle Newsletter* (2004). – http://www.sun.com/emrkt/innercircle/newsletter/0404cto.html

[SM05] Sun Microsystems, Inc.: *Enterprise JavaBeans Specification Version 3.0 (Early Draft 2)*. 2005. – http://java.sun.com/products/ejb/docs.html

[Smi04a] Smits, T.: Unbreakable Java - A Java server that never goes down. In: *JDJ* 9 (2004), Nr. 12, S. 54–56. – http://pdf.sys-con.com/Java/JDJDecember2004.pdf

[Smi04b] Smits, T.: *Unbreakable Java - The Java Server that Never Goes Down*. 2004. – https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/documents/a1-8-4/Unbreakable%20Java.pdf

[THL01] Tullmann, P. ; Hibler, M. ; Lepreau, J.: Janos: A Java-oriented OS for Active Network Nodes. In: *IEEE Journal on Selected Areas in Communications* 19 (2001), Nr. 3, S. 501–510. – http://www.cs.utah.edu/flux/papers/janos-jsac01-base.html

[TL98] Tullmann, P. ; Lepreau, J.: Nested Java Processes: OS Structure for Mobile Code. In: *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*. New York : ACM Press, 1998, S. 111–117. – http://www.cs.utah.edu/flux/papers/npmjava-esigops98web-abs.html

[Tul99] Tullmann, P.: *The Alta Operating System*, University of Utah, Diplomarbeit, 1999. – http://www.cs.utah.edu/flux/papers/tullmann-thesis-base.html

[Wil92] Wilson, P.: Uniprocessor Garbage Collection Techniques. In: *Proceedings of the International Workshop on Memory Management*. Berlin : Springer Verlag, 1992, S. 1–42. – ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps

Internetseiten/ Mailinglistenbeiträge

[1] *IBM Software - CICS family - Family Overview.* http://www.ibm.com/software/htp/cics/

[2] *IBM: z/OS operating system.* http://www.ibm.com/servers/eserver/zseries/zos/

[3] *Java 2 Platform, Enterprise Edition (J2EE).* http://java.sun.com/j2ee/

[4] *Enterprise JavaBeans Technology.* http://java.sun.com/products/ejb/

[5] *JavaServer Pages Technology.* http://java.sun.com/products/jsp/

[6] *Java Servlet Technology.* http://java.sun.com/products/servlet/

[7] *Welcome To The OMG's CORBA Website.* http://www.corba.org/

[8] *Java RMI over IIOP.* http://java.sun.com/products/rmi-iiop/

[9] *BEA Systems - BEA Tuxedo.* http://www.bea.com/tuxedo/

[10] *COM: Component Object Model Technologies.* http://www.microsoft.com/com/

[11] *IBM WebSphere Software.* http://www.ibm.com/software/websphere/

[12] *Java Transaction API (JTA).* http://java.sun.com/products/jta/

[13] *Java Transaction Service (JTS).* http://java.sun.com/products/jts/

[14] *IBM AIX 5L: UNIX operating system - an open UNIX solution.* http://www-1.ibm.com/servers/aix/

[15] *The Linux Home Page at Linux Online.* http://www.linux.org/

[16] *Microsoft Windows Family Home Page.* http://www.microsoft.com/windows/

[17] *String (Java 2 Platform SE 5.0).* http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html#intern()

[18] *J2EE Connector Architecture.* http://java.sun.com/j2ee/connector/

[19] *Java Thread Primitive Deprecation.* http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

[20] *Echidna - A free multiprocess system in Java.* http://www.javagroup.org/echidna/

[21] http://www.cs.cornell.edu/slk/JKernel/Default.html

[22] *Java Remote Method Invocation (Java RMI).* http://java.sun.com/products/jdk/rmi/

[23] *Kaffe.org.* http://www.kaffe.org/

[24] *Fluke: Flux μ-kernel Environment.* http://www.cs.utah.edu/flux/fluke/html/

[25] *The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 121.* http://jcp.org/en/jsr/detail?id=121

[26] *Java Isolation JSR-121 Interest Site.* http://www.bitser.net/isolate-interest/

[27] *JavaOne 2003 JSR-121 BOF Presentations.* http://www.bitser.net/isolate-interest/JavaOne2003/

[28] *JavaOne 2004 JSR-121 BOF Presentations.* http://www.bitser.net/isolate-interest/JavaOne2004/

[29] *[Isolate-interest] JSR-121 update.* http://altair.cs.oswego.edu/pipermail/isolate-interest/2003-July/000074.html

[30] *[Isolate-interest] JSR 121 status?* http://altair.cs.oswego.edu/pipermail/isolate-interest/2004-March/000099.html

[31] *[Isolate-interest] JSR 121 status?* http://altair.cs.oswego.edu/pipermail/isolate-interest/2004-March/000102.html

[32] *[Isolate-interest] any isolation API in J2SE.next will NOT be N:1 style.* http://altair.cs.oswego.edu/pipermail/isolate-interest/2004-October/000216.html

[33] *[Isolate-interest] any isolation API in J2SE.next will NOT be N:1 style.* http://altair.cs.oswego.edu/pipermail/isolate-interest/2004-October/000219.html

[34] *JSR-121 Expert Group Meeting 27 October 2004.* http://www.bitser.net/isolate-interest/egmeetings/m20041027.html

[35] *[Isolate-interest] JSR-121 status update.* http://altair.cs.oswego.edu/pipermail/isolate-interest/2004-December/000226.html

[36] http://www.cs.utah.edu/flux/janos/janosvm.html

[37] *The Barcelona Project.* http://research.sun.com/projects/barcelona/

[38] *[Isolate-interest] MVM for research.* http://altair.cs.oswego.edu/pipermail/isolate-interest/2005-April/000229.html

[39] *[Isolate-interest] MVM for research.* http://altair.cs.oswego.edu/pipermail/isolate-interest/2005-April/000230.html

[40] *The Barcelona Project.* http://research.sun.com/projects/barcelona/mvm/