**Jens Müller · Wilhelm G. Spruth**

# Anwendungs- und Transaktionsisolation unter Java

**Zusammenfassung** Die Java Platform, Enterprise Edition (Java EE, frühere Bezeichnung: J2EE) wurde entwickelt, um den Einsatz von Java in unternehmenskritischen Transaktionsanwendungen, wie etwa Finanzbuchhaltungs- oder Warenwirtschaftssystemen, zu ermöglichen. Die Programmierung ist jedoch immer noch sehr aufwendig, wenn es darum geht, die Isolation parallel laufender Transaktionen zu gewährleisten. Die Einhaltung der Transaktionsisolation ist eine Grundvoraussetzung, um die zugrunde liegenden Daten in einem konsistenten Zustand zu halten. Der vorliegende Beitrag analysiert die existierenden Probleme und gibt einen Überblick über die derzeitig verfügbaren oder diskutierten Lösungsansätze.

**Schlüsselwörter** ACID · Application Isolation API · Enterprise JavaBeans · Isolation · J2EE · Java EE · Java Virtual Machine · JSR-121 · Multi-Tasking Virtual Machine · Persistent Reusable Java Virtual Machine · Virtual Machine Container · Transaktion

**Abstract** The Java Platform, Enterprise Edition (Java EE, previous term: J2EE) has been designed to permit the use of Java for mission critical transactional business applications such as financial accounting or enterprise resource planning systems. When programming such applications it is still difficult to assure the isolation between transactions executing in parallel. Compliance with transaction isolation is a basic requirement to keep underlying data in a consistent state. This paper analyses existing problems and gives an overview of the solutions that are presently available or under discussion.

J. Müller (✉) · W.G. Spruth
Wilhelm-Schickard-Institut für Informatik,
Arbeitsbereich Technische Informatik, Universität Tübingen,
Sand 13,
72076 Tübingen, Deutschland
E-Mail: mail@jensmueller.com

W.G. Spruth
Institut für Informatik, Abteilung Computersysteme,
Universität Leipzig,
Johannisgasse 26,
04103 Leipzig, Deutschland
E-Mail: spruth@informatik.uni-tuebingen.de

**Keywords** ACID · Application Isolation API · Enterprise JavaBeans · Isolation · J2EE · Java EE · Java Virtual Machine · JSR-121 · Multi-Tasking Virtual Machine · Persistent Reusable Java Virtual Machine · Virtual Machine Container · Transaction

**CR Subject Classification** A.1 · D.3.2 · D.3.4 · D.4.1

**Abkürzungsverzeichnis**

| | |
|---|---|
| ABAP | Advanced Business Application Programming |
| ACID | Atomicity, Consistency, Isolation, Durability |
| AWT | Abstract Windowing Toolkit |
| EJB | Enterprise JavaBeans |
| Java EE | Java Platform, Enterprise Edition |
| JNI | Java Native Interface |
| JSR | Java Specification Request |
| JVM | Java Virtual Machine |
| MVM | Multi-Tasking Virtual Machine |
| PRJVM | Persistent Reusable Java Virtual Machine |
| SAP JVM | SAP Java Virtual Machine |
| XML | Extensible Markup Language |

## 1 Einführung

Java wurde von der Firma Sun Microsystems im Mai 1995 offiziell angekündigt und ist heute eine der am häufigsten eingesetzten Programmiersprachen. Die Java Platform, Enterprise Edition wurde erstmalig im Juni 1999 ausgeliefert.

Der Schwerpunkt der Java 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. Das Herzstück eines Transaktionsverarbeitungssystems ist der Transaktionsmonitor. Dabei handelt es sich um eine Reihe von Diensten, die den eingehenden Transaktionsfluss verwalten und koordinieren,

eine hohe Skalierbarkeit des Systems sicherstellen und Ausfallsicherheitsmechanismen zur Gewährleistung der Hochverfügbarkeit bereitstellen. Die Verbreitung objekt- und komponentenorientierter Entwicklungskonzepte führte zur Entwicklung von Objektransaktionsmonitoren [12]. Mit Einführung der Java Platform, Enterprise Edition und der serverseitigen Komponententechnologie Enterprise JavaBeans entstanden neue, auf Java basierende Transaktionsverarbeitungssysteme.

Für den Einsatz in unternehmenskritischen Transaktionsanwendungen ist die strikte Einhaltung der ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability) absolut erforderlich. In diesem Beitrag wird diesbezüglich zwischen zwei Arten der Isolation unterschieden. Der klassische Isolationslevel bei Transaktionen bestimmt üblicherweise, 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 laufender Transaktionsthreads. Im weiteren Verlauf steht der Begriff Isolation bei Transaktionen immer für Request Isolation. In der Literatur ist immer wieder auf Defizite der Java Virtual Machine (JVM) [26] hinsichtlich der Isolation hingewiesen worden:

> The existing application isolation mechanisms, such as class loaders, do not guarantee that two arbitrary applications executing in the same instance of the JVM will not interfere with one another. Such interference can occur in many places. For instance, mutable parts of classes can leak object references and can allow one application to prevent the others from invoking certain methods. The internalized strings introduce shared, easy to capture monitors. Sharing event and finalization queues and their associated handling threads can block or hinder the execution of some application. Monopolizing of computational resources, such as heap memory, by one application can starve the others. (Grzegorz Czajkowski, Laurent Daynès: Multitasking without Compromise: a Virtual Machine Evolution [5])

> Java gives the virtuoso thread programmer considerable freedom, but it also presents many pitfalls for less experienced programmers, who can create complex programs that fail in baffling ways. (Bo Sandén: Coping with Java Threads [28])

Es ist möglich, sichere Transaktionsanwendungen in Java zu schreiben. Dazu ist eine genaue Kenntnis der Spracheigentümlichkeiten erforderlich, um Isolationsprobleme zu vermeiden. Welche Java-Konstrukte zu Isolationsverletzungen führen können, ist jedoch bisher noch nicht zusammenfassend dokumentiert worden.

Der vorliegende Beitrag beschreibt, wie Anwendungen und Transaktionen unter Java parallel ausgeführt werden können. Anschließend werden die Defizite von Java in verschiedenen Bereichen diskutiert, die im Zusammenhang mit der parallelen Ausführung relevant sind. Abschließend werden die derzeitig verfolgten Lösungsansätze der Firmen IBM und SAP sowie die im Rahmen des Java Community Process spezifizierte Java Application Isolation API und deren Referenzimplementierung der Firma Sun Microsystems vorgestellt.

## 2 Parallele Ausführung und Isolation unter Java

Die JVM bietet keine explizite Möglichkeit, mehrere Anwendungen parallel auszuführen. Dennoch kann die parallele Ausführung von Anwendungen und Transaktionen durch die Verwendung von Threads bewerkstelligt werden.

Java [11] wird als sichere Sprache bezeichnet. Diese Sicherheit basiert auf vier grundlegenden Techniken: Bytecode-Verifikation, Typsicherheit, automatische Speicherbereinigung (Garbage Collection) und Speicherschutz. Darüber hinaus verwendet Java statische und dynamische Kontrollmechanismen, um über den Zugriff im Zusammenhang mit Objekten zu entscheiden. Das Sicherheitskonzept von Java bietet Threads, die parallel innerhalb derselben JVM ausgeführt werden, eine gewisse Isolation. Beispielsweise ist es nicht möglich, dass ein Thread durch Fälschung einer Objektreferenz Daten eines anderen Threads verändert.

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 JVM parallel zu verarbeiten, um einen akzeptablen Durchsatz zu erzielen.

Java ist jedoch nicht als betriebssystemähnliche Laufzeitumgebung konzipiert worden. Das Sicherheitskonzept von Java stellt den inneren Schutz von Anwendungen sicher, es bietet aber nicht die Dienstgüte des Prozessmodells bei Betriebssystemen und in Folge dessen die damit verbunden Mechanismen: Isolation, die sichere Beendigung von Anwendungen und Ressourcenmanagement. Dennoch wurde durch die Verwendung der Klassenlader (Class Loader) eine Technik entwickelt, die zumindest die Isolation garantieren sollte. Diese ist jedoch unzureichend und stellt 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 Verarbeitung von Transaktionen ist deren Isolation einer der wichtigsten Aspekte.

### 2.1 Klassenlader

Bei der Übersetzung von Java-Klassen entsteht plattformunabhängiger Bytecode. Jede Klasse wird dabei in einer eigenen class-Datei gespeichert. Bei der Ausführung eines Java-

Programms werden aber nicht alle dazu benötigten Klassen 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 Klassenlader zuständig. Klassenlader sind Objekte, die von der abstrakten Klasse ClassLoader abgeleitet sind. Jede JVM verfügt über einen Standardklassenlader (Primordial Class Loader), der nur Klassen aus dem lokalen Dateisystem laden kann. Benutzerdefinierte Klassenlader ermöglichen das Laden von Klassen von beliebigen Quellen. Beispielsweise verwenden Applets Instanzen der Klasse AppletClassLoader, die Klassen unter Benutzung des Hypertext-Transferprotokolls (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 Klassenlader verwaltet die bereits geladenen Klassen. Bevor eine neue Klasse geladen wird, überprüft ein benutzerdefinierter Klassenlader 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 dafür zuständigen Standardklassenlader delegiert.

### 2.2 Klassennamensräume

Im Folgenden wird eine Java-Anwendung als eine Menge von Threads definiert [23], die in einer Threadgruppe verwaltet werden. Der Klassennamensraum einer Java-Anwendung wird durch den Klassenlader definiert, der die Hauptklasse der Anwendung lädt (z.B. die Klasse, welche die main-Methode enthält). Der Klassennamensraum eines Klassenladers enthält Klassen, die er selbst geladen hat und alle Klassen (oder eine Teilmenge der Klassen), die der übergeordnete Klassenlader geladen hat. Ist im weiteren Verlauf von Anwendungen die Rede, so sind damit Java-Anwendungen gemeint. Im Kontext der Java Platform, Standard Edition wäre dies ein gewöhnliches Java-Programm, im Kontext der Enterprise Edition eine EJB-Anwendung.

Ein Thread kann nur auf Objekte zugreifen, die sich in der Objekthülle der mit ihm in derselben Threadgruppe verwalteten Threads befinden. Die Objekthülle einer Anwendung beinhaltet sämtliche Objekte, die während der Ausführung ihrer Threads erzeugt wurden und über deren Stacks 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 der Anwendung referenziert werden.

Der Klassennamensraum enthält also Klassen, die in einer Datenstruktur des Klassenladers gespeichert sind und als Vorlage für Objekte 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 der Anwendung erreichbar sind.

### 2.3 Unsichere statische Felder und unsichere Klassen

In Bezug auf die Isolation kann man zwischen sicheren und unsicheren statischen Feldern unterscheiden. Sichere statische Felder können weder dazu benutzt werden, Objektreferenzen zwischen Anwendungen auszutauschen, noch dazu, den Zustand einer Anwendung zu modifizieren. Konstante statische Felder primitiven Typs sind sicher, z.B. das Feld MAX_VALUE der Systemklasse Integer. Inkonstante statische Felder sind generell unsicher. Statische Felder konstanten Klassentyps sind je nach Semantik sicher oder unsicher. Dies hängt davon ab, ob die Klasse über einen statischen Zustand verfügt, der von außerhalb direkt oder indirekt veränderbar ist und etwaige Änderungen einer Anwendung Auswirkungen auf andere Anwendungen haben können. Konstante Felder des Klassentyps Long, Double oder String sind beispielsweise sicher. Der Standardausgabestrom – das Feld out der Systemklasse System – ist im Gegensatz dazu unsicher, da eine Anwendung den Strom schließen könnte und er dadurch für alle anderen Anwendungen ebenfalls geschlossen wäre.

Klassen, die unsichere statische Felder enthalten, sind potentiell unsichere Klassen. Auch sind Klassen unsicher, die Methoden enthalten, bei deren Ausführung Nebenwirkungen mit Auswirkungen auf andere Anwendungen auftreten können. Dies ist beispielsweise bei den Klassen Runtime und System der Fall, da sie Methoden enthalten, deren Aufruf die JVM und damit alle darin ausgeführten Anwendungen beenden.

### 2.4 Multitasking unter Verwendung eines Klassenladers

Teilen sich Anwendungen innerhalb derselben JVM denselben Klassenlader, so teilen sie sich auch denselben Klassennamensraum. Die Schnittmenge ihrer Objekthüllen enthält sämtliche Objekte, die von statischen Feldern der gemeinsamen Klassen referenziert werden. Objekte innerhalb dieser Schnittmenge und alle inkonstanten statischen Felder können von jeder Anwendung manipuliert werden, was eventuell die Integrität einer Anwendung verletzt.

### 2.5 Multitasking unter Verwendung mehrerer Klassenlader

Um dieses und weitere Probleme bei der Ausführung mehrerer Anwendungen innerhalb derselben JVM zu lösen, wird üblicherweise jede Anwendung und die von ihr verwendeten Klassen von einem eigenen Klassenlader geladen [25]. Dadurch ändert sich die Semantik statischer Felder von globalen Variablen auf JVM-Ebene zu globalen Variablen auf Anwendungsebene. Wird beispielsweise eine Klasse von zwei

verschiedenen Anwendungen mittels getrennter Klassenlader geladen, so haben Manipulationen der einen Anwendung an statischen Feldern dieser Klasse keinen Einfluss auf die korrespondierenden statischen Felder dieser Klasse der jeweils anderen Anwendung. Diese Methode wird auch von Java EE-Applikationsservern verwendet, um EJB-Anwendungen zu isolieren.

## 3 Isolationsdefizite der Java Virtual Machine

Durch die Verwendung mehrerer Klassenlader wird ein relativ hoher Isolationsgrad erreicht. Dennoch weist auch diese Technik Defizite auf, die darauf zurückzuführen sind, dass die JVM nicht für die parallele Ausführung von Anwendungen und Transaktionen konzipiert wurde. Die daraus resultierenden Probleme lassen sich folgendermaßen klassifizieren: spezifische Probleme bei Verwendung mehrerer Klassenlader, Probleme aufgrund systemweiter Datenstrukturen und Mechanismen, Probleme bei der Beendigung von Anwendungen und Probleme aufgrund fehlenden Ressourcenmanagements. Eine detaillierte Diskussion ist unter [27] zu finden.

### 3.1 Spezifische Probleme bei Verwendung mehrerer Klassenlader

#### *3.1.1 Unsichere statische Felder von Systemklassen und unsichere Klassen*

Die Isolation bei Verwendung mehrerer Klassenlader ist unvollständig, da statische Felder und synchronisierte Klassenmethoden von Systemklassen nicht repliziert werden.

Eine Untersuchung der Standardpakete der Java Platform, Standard Edition 5.0 ergab, dass keine direkt zugänglichen unsicheren statischen Felder existieren, der Zugriff aber bei über 40 Feldern durch Klassenmethoden möglich ist. Bei manchen dieser Methoden ist eine Wertzuweisung nur einmal möglich. Falls eine Anwendung einem derartigen Feld einen Wert zugewiesen hat, führen weitere Zuweisungsversuche anderer Anwendungen zu Ausnahmen. Des Weiteren ist die Ausführung der meisten Methoden abhängig vom Sicherheitsmanager und wurde unter WebSphere bei aktivierter Java 2-Sicherheit bei allen zurückgewiesen.

Bei Java-Anwendungen im Kontext der Java Platform, Standard Edition sind gemeinsam verwendete unsichere statische Felder von Systemklassen nicht akzeptabel. Durch den bei Applikationsservern üblicherweise aktivierten Sicherheitsmanager ist die Problematik bei EJB-Anwendungen weitgehend entschärft, die Klassenmethoden ohne Sicherheitsüberprüfung scheinen unkritisch zu sein. Um zu einem endgültigen Ergebnis zu kommen, müssten aber die statischen Felder aller Klassen der Java Platform, Enterprise Edition und deren Semantik untersucht werden, was den Rahmen dieses Beitrags gesprengt hätte.

#### *3.1.2 Synchronisierte Klassenmethoden von Systemklassen*

Bei synchronisierten Klassenmethoden kann ein Problem auftreten, falls ein Thread, der sich innerhalb einer solchen Methode befindet, von einem anderen Thread derselben Anwendung ausgesetzt wird und sie in Folge dessen von anderen Anwendungen nicht mehr ausgeführt werden kann. Der Grund dafür ist, dass es sich bei synchronisierten Klassenmethoden um Monitore handelt. Ein Monitor [31] ist ein Konstrukt, mit dessen Hilfe Prozesse oder 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.

Auch wenn das Aussetzen von Threads innerhalb von Enterprise Beans verboten ist, gelang es im Test mit wenigen Zeilen Code, Deadlocks zu erzeugen. Instanzen einer fehlerhaft programmierten Session Bean, die nicht der Spezifikation entspricht, könnten sich auf diese Weise gegenseitig blockieren.

#### *3.1.3 Replikation von Anwendungsklassen*

Ein weiterer Nachteil entsteht durch die Replikation von Anwendungsklassen durch verschiedene Klassenlader, die eigentlich der Isolation zugutekommt. Diese Vervielfachung ist aber bei der Verwendung von Just In Time-Übersetzern 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 [4].

### 3.2 Probleme aufgrund systemweiter Datenstrukturen und Mechanismen

#### *3.2.1 Internalisierte Strings*

Bestimmte Datenstrukturen der Laufzeitumgebung werden von allen Anwendungen gemeinsam verwendet, z.B. gibt es nur eine einzige Datenbasis für 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, dass sie mit dem Operator == schneller

auf Gleichheit überprüft werden können. String-Literale und konstante Strings werden automatisch internalisiert.

Werden internalisierte Strings als Sperrobjekte in synchronized-Abschnitten verschiedener Anwendungen verwendet, so kann dies zu unerwarteten Interaktionen zwischen den Anwendungen führen, sofern es sich um identische Zeichenketten handelt. Um an dieser Stelle eine vollständige Isolation zu erreichen, müsste die JVM für jede Anwendung eine eigene Datenbasis für internalisierte Strings unterhalten.

Darüber hinaus handelt es sich bei der zur Internalisierung verwendeten synchronisierten Klassenmethode String.intern() um einen Monitor (siehe Abschnitt 3.1.2), der von allen innerhalb der JVM 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.

#### *3.2.2 Finalisierer und Ereignisverarbeitung*

Finalisierer und Ereignisse des Abstract Windowing Toolkit (AWT) – die Standardschnittstelle für grafische Benutzeroberflächen – werden jeweils von einem eigenen Thread verarbeitet. Diese Verarbeitung findet sequentiell statt, was die Isolation von Anwendungen gefährden kann.

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 und damit die Freigabe von Ressourcen einer anderen Anwendung verhindern.

Ereignisse der grafischen Benutzeroberfläche werden ebenfalls 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 Anwendungen mit grafischer Benutzeroberfläche, die zu diesem Zeitpunkt von der JVM ausgeführt werden, nicht mehr reagieren. Im Kontext von Enterprise JavaBeans sind derartige Probleme irrelevant, da die EJB-Spezifikation den Zugriff auf das AWT verbietet.

#### *3.2.3 Plattformabhängiger Code*

Die Einbindung plattformabhängigen Codes durch die Java Native Interface (JNI) kann ebenfalls zu Problemen führen. Plattformabhängiger Code wird im Adressraum der JVM ausgeführt und hat somit uneingeschränkten Zugriff auf alle Anwendungsdaten. Enterprise Beans dürfen wiederum keine plattformabhängigen Bibliotheken laden, Java EE-Applikationsserver greifen aber über die J2EE Connector Architecture [36] häufig auf plattformabhängigen Code zu. Im Experiment führte die Verwendung einer fehlerhaften plattformabhängigen Bibliothek innerhalb eines Ressourcenadapters zum Absturz des Applikationsservers. In solch einem Fall werden alle laufenden Transaktionen abgebrochen.

### 3.3 Probleme bei der Beendigung von Anwendungen

#### *3.3.1 Explizite Beendigung der Java Virtual Machine*

Verwendet eine EJB-Anwendung die unsichere Runtime- oder System-Klasse zur Beendigung der JVM, werden alle parallel ausgeführten Anwendungen ebenfalls beendet, auch wenn dies seitens der EJB-Spezifikation verboten ist. Ein aktivierter und korrekt konfigurierter Sicherheitsmanager kann dies zwar verhindern, zumindest bei WebSphere ist dies in der Standardeinstellung aber nicht der Fall.

#### *3.3.2 Beendigung von Threads*

Java bietet keine sichere Möglichkeit, Threads zu beenden. Von der Verwendung der Primitiva Thread.stop(), Thread.suspend(), Thread.resume() und Runtime.runFinalizersOnExit(boolean) wird seit Version 1.2 abgeraten [34]. Dafür gibt es mehrere Gründe [13]. Bei Aufruf von Thread.stop() werden z.B. alle vom Thread gesperrten Monitore freigegeben. Wenn ein von einem solchen Monitor zuvor geschütztes Objekt dadurch in einem inkonsistenten Zustand hinterlassen wird, könnten andere Threads wieder auf dieses als beschädigt bezeichnete Objekt zugreifen, was zu unvorhersehbarem Verhalten führen könnte. Die nicht vorhandene Möglichkeit einer sicheren Beendigung ist gerade bei Anwendungen innerhalb eines Java EE-Applikationsservers problematisch. Ist eine Anwendung fehlerhaft programmiert und kann nicht erfolgreich beendet werden, muss der gesamte Applikationsserver neu gestartet werden, um sie aus dem Speicher zu entfernen.

### 3.4 Probleme aufgrund des fehlenden Ressourcenmanagements

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 stehen würde.

Um auch in diesem Fall praktische Ergebnisse zu erzielen, wurde eine EJB-Anwendung sowie ein externer Java-Client entwickelt, um solche Angriffe wenigstens ansatzweise zu simulieren. Dabei führten Threads im Hintergrund permanent Transaktionen im Rahmen der EJB-Anwendung durch, deren Durchsatz gemessen wurde. Parallel zur Transaktionsverarbeitung wurde ein Denial-of-Service-Angriff durchgeführt, der den Großteil der CPU-Zeit für sich beanspruchte. Dieser Angriff führte je nach Einstellung seiner Parameter zu einem völligen Stillstand der Transaktionsverarbeitung.

### 3.5 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 dargestellten Probleme kann sie aber nur unzureichend lösen. Die Probleme im 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 Java EE-Applikationsserver 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 [22]. Aber auch wenn nur eine Anwendung ausgeführt wird, bestehen dieselben Probleme bei der parallelen Ausführung von Transaktionen dieser Anwendung.

## 4 Lösungsansätze

Die Probleme der JVM 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.

### 4.1 Verwendung einer eigenen Java Virtual Machine für jede Transaktion

Eine Möglichkeit zur Lösung der im letzten Kapitel analysierten Probleme scheint die Verwendung einer eigenen JVM für jede Transaktion, um sowohl Isolation als auch Parallelität zu erreichen. Diese vermeintliche Lösung zur Umgehung der geschilderten Probleme weist aber weitgehend dieselben Defizite auf, da es grundsätzlich möglich ist, dass eine Transaktion die Ausführung der Folgetransaktion beeinflusst, indem sie den Zustand der JVM verändert. Beispiele für eventuell sicherheitskritische Hinterlassenschaften der vorhergehenden Transaktion sind überschriebene statische Felder, gestartete Threads oder geladene plattformabhängige Bibliotheken. Auch aufgrund der langen Startzeit und des hohen Speicherverbrauchs der JVM scheidet diese Vorgehensweise aus.

Zwei Lösungsansätze, die in den nächsten zwei Abschnitten vorgestellt werden, setzen dennoch auf die Verwendung einer eigenen JVM für jede Transaktionen, vermeiden jedoch die aufgezeigten Probleme und Nachteile durch proprietäre Erweiterungen der JVM. Sie können jedoch nur in speziellen Hard- und Softwareumgebungen eingesetzt werden und stellen damit keine allumfassende Lösung der Isolationsproblematik dar.

### 4.2 Persistent Reusable Java Virtual Machine (IBM)

Die Persistent Reusable Java Virtual Machine (PRJVM) ([10], [3], [17], [2]) ist eine JVM mit erweiterter Funktionalität, die unter dem z/OS Betriebssystem verfügbar ist. z/OS (frühere Bezeichnung: OS/390) ist ein Großrechnerbetriebssystem der Firma IBM, welches in großen Unternehmen als zentraler Server für betriebswirtschaftliche Anwendungen eingesetzt wird [15]. Die Entwicklung der PRJVM hatte zum Ziel, unter allen Umständen die Einhaltung der Isolation bei der Verarbeitung von Transaktionen zu gewährleisten.

Da eine vollständige Isolation von Transaktionen innerhalb derselben JVM nicht möglich ist, werden getrennte virtuelle Maschinen für die parallele Verarbeitung von Transaktionen eingesetzt. Zur Leistungsverbesserung laufen alle virtuellen Maschinen in isolierten Laufzeitumgebungen (Run Time Units) innerhalb eines einzigen virtuellen Adressraums. Die Run Time Units werden als Enclaves bezeichnet und stellen einen Thread-ähnlichen Mechanismus dar [18]. Eingehende Transaktionen werden von einer Master Virtual Machine auf die einzelnen Instanzen verteilt. Sie ist auch dafür zuständig, die jeweilige Instanz nach Abschluss einer Transaktion wieder instand zu setzen. Durch die Verwendung mehrerer virtueller Maschinen benötigt das Transaktionsverarbeitungssystem zwar mehr Speicher, bietet aber vollständige Isolation. Das Problem des erhöhten Speicherbedarfs wird durch eine Unterteilung des Heaps in getrennte Bereiche teilweise gelöst. Diese haben jeweils unterschiedliche Eigenschaften:

- Der System Heap besteht aus dem Main System Heap, der unter anderem die Systemklassen enthält, und dem Application-Class System Heap. Einzelheiten sind in [17] zu finden.
- Der Middleware Heap enthält Objekte, die von Middleware-Klassen erzeugt werden. Middleware-Klassen haben höhere Privilegien als Anwendungsklassen, beispielsweise können Klassen und Bibliotheken in einen Cache geladen und von mehreren Anwendungen benutzt werden.
- Der Transient Heap enthält Objekte, die von Anwendungsklassen erzeugt werden.

Die Unterteilung in mehrere Heaps ermöglicht eine Aufteilung in transaktionsbezogene (und somit kurzlebige) und langlebigere Daten. Dies ermöglicht es, die weiter unten erwähnte Instandsetzung zwischen zwei Transaktionen besonders schnell durchzuführen. Die Lebensdauer der verschiedenen Objekte wird durch eine eigene Garbage Collection Policy für jeden Abschnitt des Heaps festgelegt [16].

Die unterteilte Heap-Struktur der PRJVM ähnelt dem in einigen virtuellen Maschinen realisierten Ansatz Generational Garbage Collection [1]. Bei diesem wird der Heap ebenfalls in verschiedene Bereiche unterteilt, die als Generations bezeichnet werden. Der in der Regel Nursery genannte Teil enthält dabei neu instanziierte Objekte, die nach einiger Zeit in die älteren Abschnitte des Heaps verlagert werden. Objekte, die über einen sehr langen Zeitraum existieren, kommen in einen speziellen Teil des Heaps, der nicht mehr einer regelmäßigen Garbage Collection unterzogen wird. Im Vergleich hierzu hat die unterteilte Heap-Struktur des PRJVM-Ansatzes den Vorteil, dass man die erwartete Lebensdauer der Objekte a priori bestimmen kann. Somit wird ein Verschieben der Objekte im Speicher unnötig. Auch fehlt in der Technik Generational Garbage Collection der hohe Grad an Isolation, den die PRJVM bietet.

Wie bereits erwähnt sollte eine JVM nach Verarbeitung einer Transaktion nicht wiederverwendet werden, um höchste Sicherheit zu garantieren. Daher muss für jede Transaktion eine neue JVM erzeugt werden. Die Dauer des Startvorgangs einer JVM ist aber inakzeptabel, da zwischen 20 und 100 Millionen Maschineninstruktionen ausgeführt werden müssen [3]. Um dies zu vermeiden, wurden mit der PRJVM neue Konzepte eingeführt, die einen Neustart im Normalfall überflüssig machen. Das wichtigste dieser Konzepte ist die Möglichkeit, eine PRJVM, die bereits eine Transaktion verarbeitet hat, in ihren Ursprungszustand zu versetzen. Diese Funktionalität wird durch eine neue JNI-Funktion namens ResetJavaVM implementiert. Die Instandsetzung einer PRJVM 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 ein Aufruf der ResetJavaVM-Funktion false zurück.

Die PRJVM wird unter dem z/OS-Betriebssystem vom Transaktionsmonitor CICS (Customer Information Control System) Transaction Server sowie von gespeicherten Prozeduren (Stored Procedures) des Datenbankmanagementsystems DB2 unterstützt und ist seit dem Jahr 2000 verfügbar.

### 4.3 Virtual Machine Container (SAP)

Javas Defizite im Bereich der Isolation veranlassten auch die Firma SAP, Transaktionsverarbeitung unter Java sicherer zu gestalten. Dies führte zur Entwicklung des Virtual Machine Containers ([24], [32], [30]), der seit Anfang 2006 Bestandteil der Integrations- und Anwendungsplattform NetWeaver 7.0 (frühere Bezeichnung: 2004s) ist. Die NetWeaver-Plattform besteht unter anderem aus dem Web Application Server ABAP und dem J2EE-konformen Web Application Server Java. Der Web Application Server ABAP konnte bisher lediglich in der Programmiersprache ABAP geschriebene Transaktionen verarbeiten, deren Isolation auf prozessbasierter Sicherheit beruht. Ein Dispatcher verteilt eingehende Anfragen auf eine Reihe von sich bereits im Hauptspeicher befindlichen Arbeiterprozessen. 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.

Der Virtual Machine Container (VMC) ist eine in den SAP Web Application Server ABAP integrierte Komponente, mit dessen Einführung eine Einbindung einer JVM in den ABAP-Arbeiterprozess ermöglicht wird. Dadurch können auch in diesem Teil der NetWeaver-Plattform Java-Funktionen ausgeführt werden. Der VMC ist für Anwendungen optimiert, welche sowohl in ABAP als auch in Java implementierte Funktionen verwenden und schnell und zuverlässig miteinander kommunizieren müssen.

Die Integration einer JVM bietet eine Reihe von Vorteilen, vor allem eine strenge Isolation zwischen aktiven Benutzersitzungen. Analog zu ABAP-Transaktionen verarbeitet die JVM innerhalb eines Arbeiterprozesses zu einem Zeitpunkt höchstens eine Anfrage, selbst deren Absturz beeinträchtigt im schlimmsten Fall den gerade aktiven Benutzer. Um den Speicherverbrauch zu minimieren, können die virtuellen Maschinen innerhalb der einzelnen Arbeiterprozesse auf einen gemeinsamen Speicherbereich zugreifen, in den gemeinsam verwendete Klassen geladen werden. Des Weiteren können auch Objekte und alle von ihnen aus erreichbaren Objekte (transitive Hülle), sogenannte Shared Closures, von mehreren virtuellen Maschinen gemeinsam verwendet werden. Dies kann entweder durch eine schnelle Einblendung der Objekte in den Adressraum der jeweiligen JVM oder eine Kopie bewerkstelligt werden. Eine eingeblendete Shared Closure kann jedoch nur gelesen und nicht verändert werden.

Die Zustandsdaten einer Benutzersitzung – auch Benutzerkontext genannt – werden nach jeder verarbeiteten Anfrage ebenfalls in einen von allen virtuellen Maschinen gemeinsam verwendeten Speicherbereich kopiert. Auf diese

Weise kann der Benutzerkontext bei der nächsten Anfrage einem beliebigen, nicht verwendeten Arbeiterprozess zugeordnet werden. Dadurch wird verhindert, dass Arbeiterprozesse zwischen den einzelnen Anfragen innerhalb einer Sitzung brach liegen. Ein Konzept namens Process Attachable Virtual Machines ermöglicht darüber hinaus, die JVM vom korrespondierenden Arbeiterprozess zu trennen und ihr Speicherabbild schnell in den Adressraum eines anderen Prozesses einzublenden. Mit Hilfe dieser Technik können in einem Pool vorrätig gehaltene virtuelle Maschinen Arbeiterprozessen dynamisch zugeordnet werden. Die Trennung einer JVM ist beispielsweise sinnvoll, falls die Verarbeitung einer Anfrage aufgrund gewisser Operationen (z.B. Netzwerkein- und Ausgabe) keine Prozessorzeit in Anspruch nimmt. In diesem Fall kann sie temporär durch eine JVM aus dem Pool ausgetauscht werden, die neue Anfragen verarbeiten kann.

Der VMC ist allerdings kein Ersatz für den Web Application Server Java und steht zudem ausschließlich ausgewählten Java-Komponenten von SAP zur Verfügung – von Kunden entwickelte Java-Funktionen für den VMC werden nicht unterstützt. Allerdings wird mit NetWeaver 7.1 zum ersten Mal die SAP Java Virtual Machine (SAP JVM) [29] ausgeliefert werden, die auf der Sun HotSpot Java Virtual Machine basiert und das Konzept der Shared Closures vom VMC übernimmt. Die SAP JVM wird als Bestandteil des Web Application Server Java dann auch dort die Auslagerung des Benutzerkontexts unterstützen. Da normalerweise nur etwa zehn 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 erheblich verringert.

### 4.4 Application Isolation API

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.

Die Application Isolation API ermöglicht der JVM im Gegensatz zur bisherigen Verfahrensweise die vollständig isolierte Ausführung von Anwendungen. Zentraler Bestandteil der API ist die Klasse Isolate, welche 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.

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. Weiterhin können unterschiedliche Implementierungen einen unterschiedlichen Isolationsgrad zur Verfügung stellen. Alle konformen Implementierungen müssen jedoch Anwendungen gewisse Voraussetzungen garantieren. Die endgültige Version der Spezifikation wurde am 13. Juni 2006 veröffentlicht [19].

### 4.5 Multi-Tasking Virtual Machine (Sun Microsystems)

In Form eines Forschungsprojekts arbeitet die Firma Sun Microsystems seit einigen Jahren an der JSR-121-Referenzimplementierung, der Multi-Tasking Virtual Machine (MVM) ([5], [33], [14]).

Das Design der MVM wurde durch drei Zielvorgaben bestimmt. Erstens sollten sich Anwendungen an keiner Stelle innerhalb der JVM beeinflussen können. Zweitens sollte jeder Anwendung der Eindruck vermittelt werden, als würden keine weiteren Anwendungen innerhalb derselben JVM ausgeführt. Drittens sollte das Design zu einer guten Leistung und Skalierbarkeit führen. Die MVM ist eine ganzheitliche Lösung, welche die derzeitigen Defizite der JVM adressiert und beheben soll.

Die Untersuchung aller Komponenten der JVM und eine anschließende Entscheidung, ob diese gemeinsam von Anwendungen genutzt werden können oder nicht, stellt das Grundprinzip des Designs dar [7]. Beispielsweise kann die Laufzeitrepräsentation von Klassen größtenteils gemeinsam verwendet werden und nur ein Teil muss pro Isolate repliziert werden, z.B. der statische Zustand. Durch die Verwendung von Isolates entfallen die spezifischen Probleme der Klassenlader und die Probleme bei der Beendigung von Anwendungen. Die Probleme im Zusammenhang mit systemweiten Finalisierungs- und Ereigniswarteschlangen werden ebenfalls durch Replikation gelöst. Plattformabhängiger Code wird sicher in einem separaten Prozess ausgeführt [6]. In Folge dessen stellt auch ein fehlerhafter Ressourcenadapter kein allzu großes Problem mehr dar. Insgesamt werden so die Probleme aufgrund systemweiter Datenstrukturen und Mechanismen beseitigt. Da die MVM Mechanismen zum Ressourcenmanagement beinhaltet ([8], [9], [21]), sind schließlich alle Problembereiche der bisherigen JVM abgedeckt. Die zugrunde liegende Spezifikation der Resource Consumption Management API wird im Rahmen des JSR-284 [20] vorangetrieben.

Die MVM scheint daher eine umfassende Lösung zu werden, um Anwendungen und Transaktionen unter Java parallel, sicher und effizient ausführen zu können. Die Forscher experimentierten auch mit dem Einsatz in J2EE-Applikationsservern [22] und beschreiben zwei Einsatzszenarien der Application Isolation API: Architekturbasierte und anwendungsbasierte Isolation.

Bei architekturbasierter Isolation wird jede Komponente eines Java EE-Applikationsservers (z.B. der EJB-Container) in einem eigenen Isolate ausgeführt. Der Vorteil ist, dass an

der bisherigen Java EE-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. Beide Ansätze können miteinander kombiniert werden. Auch sind verschiedene Stufen der Granularität denkbar, z.B. könnten einzelne Servlets und Enterprise Beans innerhalb des Web- bzw. EJB-Containers in einem eigenen Isolate ausgeführt werden. Eine dahingehende Modifikation bestehender Applikationsserver wäre aber sehr aufwendig.

Einige Konzepte der MVM sind bereits in die Entwicklung der Java Platform, Micro Edition eingeflossen. Eine Vorabversion der MVM für Solaris auf der SPARC-Plattform ist derzeit im Rahmen von Forschungszwecken erhältlich [35].

## 5 Zusammenfassung

Der Einsatz von Java EE für die Entwicklung größerer unternehmenskritischer Transaktionsanwendungen ist bisher nur sehr zögerlich erfolgt. Der existierende Standard hat zwei Schwachstellen: Die gegenseitige Isolation parallel laufender Threads und die Wiederverwendung einer JVM für aufeinanderfolgende Transaktionen. Auch durch Verwendung mehrerer Klassenlader weist die JVM Defizite im Bereich der Isolation auf. Statische Felder und statische Klassenmethoden, internalisierte Strings, systemweite Ereignis- und Finalisierungswarteschlagen sowie plattformabhängiger Code können zur gegenseitigen Beeinflussung von Anwendungen und Transaktionen führen. Das fehlende Ressourcenmanagement und die Probleme bei der Beendigung von Anwendungen machen deutlich, dass Java nicht für die parallele Ausführung von Anwendungen und Transaktionen konzipiert wurde. Es existiert keine befriedigende Dokumentation über programmtechnische Maßnahmen zur Vermeidung von Problemen.

Zwei bereits verfügbare Eigenentwicklungen der Firmen IBM und SAP, die Persistent Reusable Java Virtual Machine bzw. der Virtual Machine Container, sind geeignet, die bestehenden Probleme zumindest im Kontext unternehmenskritischer Transaktionsanwendungen zu lösen. Diese können bisher allerdings nur in einigen Hard- und Softwareumgebungen eingesetzt werden. IBM verwendet für jede Transaktion eine eigene PRJVM, die nach abgeschlossener Verarbeitung normalerweise in den Ursprungszustand versetzt und für nachfolgende Transaktionen sicher wiederverwendet werden kann. Der Virtual Machine Container von SAP ermöglicht die Einbindung einer JVM in einen ABAP-Arbeiterprozess, wobei jede Transaktion von einer eigenen JVM verarbeitet wird, die von dem ihr zugeordneten Benutzerkontext getrennt und damit für nachfolgende Transaktionen ebenfalls wiederverwendet werden kann. Da der Virtual Machine Container nicht für von Kunden entwickelte Java-Funktionen vorgesehen ist und die Multi-Tasking Virtual Machine der Firma Sun Microsystems sich noch im Prototypenstadium befindet, stellt die PRJVM bis dato die einzige Möglichkeit dar, sichere Transaktionsverarbeitung mit Java und unter Produktionsbedingungen durchzuführen. Es bleibt zu hoffen, dass die innerhalb des Java Community Process entwickelte Application Isolation API als Bestandteil in den Java EE-Standard aufgenommen wird und die bestehenden Probleme auf diese Weise zufriedenstellend gelöst werden.

**Danksagung** Wir danken Herrn Prof. Dr. Rosenstiel, Wilhelm-Schickard-Institut für Informatik, Eberhard Karls Universität Tübingen, für seine Unterstützung der vorliegenden Arbeit.

## Literatur

1. Appel AW (1989) Simple generational garbage collection and fast allocation. Source Software – Practice & Experience 19(2):171–183, URL `http://www.cs.princeton.edu/~appel/papers/143.ps`
2. Beyerle M, Franz J, Spruth WG (2005) Persistent Reusable Java Virtual Machine unter z/OS und Linux. Informatik – Forschung Entwicklung 20(1-2):102–111, URL `http://www-ti.informatik.uni-tuebingen.de/~spruth/Mirror/pers28.pdf`
3. Borman S, Paice S, Webster M, Trotter M, McGuire R, Stevens A, Hutchison B, Berry R (2000) A Serially Reusable Java Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing. Tech. Rep. TR 29.3406, IBM Hursley, URL `http://www.ibm.com/servers/eserver/zseries/software/java/pdf/29.3406.pdf`
4. Cramer T, Friedman R, Miller T, Seberger D, Wilson R, Wolczko M (1997) Compiling Java Just in Time. IEEE Micro 17(3):36–43, DOI 10.1109/40.591653
5. Czajkowski G, Daynès L (2001) Multitasking without Compromise: a Virtual Machine Evolution. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, pp 125–138, URL `http://research.sun.com/projects/barcelona/papers/oopsla01.pdf`
6. Czajkowski G, Daynès L, Wolczko M (2001) Automated and Portable Native Code Isolation. Tech. Rep. TR-2001-96, Sun Microsystems, URL `http://research.sun.com/techrep/2001/abstract-96.html`
7. Czajkowski G, Daynès L, Nystrom N (2002) Code Sharing among Virtual Machines. In: Proceedings of the 16th European Conference on Object-Oriented Programming, Springer-Verlag, Berlin Heidelberg New York, pp 155–177, URL `http://research.sun.com/projects/barcelona/papers/ecoop02.pdf`
8. Czajkowski G, Hahn S, Skinner G, Soper P (2002) Resource Consumption Interfaces for Java Application Programming – a Proposal. In: Proceedings of the 16th European Conference on Object-Oriented Programming, Springer-Verlag, Berlin Heidelberg New York, pp 6–7, URL `http://www.ovmj.org/workshops/resman/resman02.tar.gz`
9. Czajkowski G, Hahn S, Skinner G, Soper P, Bryce C (2003) A Resource Management Interface for the Java Platform. Tech. Rep. TR-2003-124, Sun Microsystems, URL `http://research.sun.com/techrep/2003/abstract-124.html`
10. Dillenberger D, Bordawekar R, Clark III CW, Durand D, Emmes D, Gohda O, Howard S, Oliver MF, Samuel F, St John RW (2000) Building a Java virtual machine for server applications: The Jvm on OS/390. IBM Systems Journal 39(1):194–

210, URL http://www.research.ibm.com/journal/sj/391/dillenberger.pdf
11. Gosling J, Joy B, Steele G, Bracha G (2005) The Java Language Specification, 3rd edn. Addison-Wesley, Boston
12. Hammerschall U (2002) Applikations-Server – gestern, heute, morgen? JavaSPEKTRUM 09/2002:22–26
13. Hawblitzel C, von Eicken T (1999) Tasks and Revocation for Java (or, Hey! You got your Operating System in my Language!). URL http://www.cs.cornell.edu/Info/People/hawblitz/PLDI2000-submit/luna-99-11-13.ps
14. Heiss JJ (2005) The Multi-Tasking Virtual Machine: Building a Highly Scalable JVM. Sun Developer Network, URL http://java.sun.com/developer/technicalArticles/Programming/mvm/
15. Herrmann P, Kebschull U, Spruth WG (2003) Einführung in z/OS und OS/390, 2nd edn. Oldenbourg Wissenschaftsverlag, München
16. IBM Corp (2002) Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.2, 2nd edn. IBM ITSO
17. IBM Corp (2003) Persistent Reusable Java Virtual Machine User's Guide, 5th edn. IBM ITSO, URL http://www.ibm.com/servers/eserver/zseries/software/java/pdf/prjvm14.pdf
18. IBM Corp (2005) Language Environment Concepts Guide, seventh edn. IBM ITSO, URL http://publibz.boulder.ibm.com/epubs/pdf/ceea8160.pdf
19. Java Community Process (2006) JSR-000121 Application Isolation API Specification (Final Release). URL http://jcp.org/aboutJava/communityprocess/final/jsr121/
20. Java Community Process (2006) JSR-000284 Resource Consumption Management API (Public Review Draft). URL http://jcp.org/aboutJava/communityprocess/pr/jsr284/
21. Jordan M, Czajkowski G, Kouklinski K, Skinner G (2003) Extending a J2EE Server with Dynamic and Flexible Resource Management. In: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag, Berlin Heidelberg New York, pp 439–458, URL http://research.sun.com/projects/barcelona/papers/middleware04.pdf
22. Jordan M, Daynès L, Czajkowski G, Jarzab M, Bryce C (2004) Scaling J2EE Application Servers with the Multi-Tasking Virtual Machine. Tech. Rep. TR-2004-135, Sun Microsystems, Inc., URL http://research.sun.com/techrep/2004/abstract-135.html
23. Krause J, Plattner B (2000) Safe Class Sharing among Java Processes. Tech. Rep. 3230, IBM Research, Zurich Research Laboratory, URL http://www.zurich.ibm.com/pdf/rz3230.pdf
24. Kuck N, Kuck H, Lott E, Rohland C, Schmidt O (2002) SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers. Work-in-Progress Report, 2nd Java Virtual Machine Research and Technology Symposium, URL http://www.bitser.net/isolate-interest/papers/PAVM.pdf
25. Liang S, Bracha G (1998) Dynamic Class Loading in the Java Virtual Machine. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, pp 36–44, URL http://www.bracha.org/classloaders.ps
26. Lindholm T, Yellin F (1999) The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Boston
27. Müller J (2005) Anwendungs- und Transaktionsisolation unter Java. Diplomarbeit, Wilhelm-Schickard-Institut für Informatik, Eberhard Karls Universität Tübingen, URL http://www-ti.informatik.uni-tuebingen.de/~spruth/DiplArb/jmueller.pdf
28. Sandén B (2004) Coping with Java Threads. IEEE Computer 37(4):20–27
29. SAP AG (2006) Application Server's Robustness and High Availability. URL https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/73a16f1d-0e01-0010-95a6-a4f66ca1a65e
30. SAP AG (2007) Documentation for SAP NetWeaver Release 7.0. URL http://help.sap.com/saphelp_nw70/helpdata/en/
31. Silberschatz A, Galvin PB, Gagne G (2004) Operating System Concepts, seventh edn. John Wiley & Sons, New York
32. Smits T (2004) Unbreakable Java – A Java server that never goes down. JDJ 9(12):54–56, URL http://pdf.sys-con.com/Java/JDJDecember2004.pdf
33. Sun Microsystems, Inc (2004) The Multitasking Virtual Machine. Sun Inner Circle Newsletter, URL http://www.sun.com/emrkt/innercircle/newsletter/0404cto.html
34. Sun Microsystems, Inc (2006) JDK 6 Documentation: Java Thread Primitive Deprecation. URL http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html
35. Sun Microsystems, Inc (2006) Multi-Tasking Virtual Machine Download. URL http://research.sun.com/projects/barcelona/mvm/
36. Sun Microsystems, Inc (2003) J2EE Connector Architecture Specification Version 1.5. URL http://java.sun.com/j2ee/connector/download.html