Isolation in Java Transaktionen

Robert Harbach, Wolfgang Rosenstiel, Wilhelm G. Spruth
Wilhelm-Schickard-Institut für Informatik, Eberhard Karls Universität Tübingen
robert.harbach@gmail.com, {rosenstiel, spruth}@informatik.uni-tuebingen.de

Abstract

Using Threads the Java Virtual Machine (JVM) is able to support several applications in parallel. Since, however, the JVM does not enable real application isolation, hosting several applications in one JVM can be critical if these applications represent transactions. Therefore, the common practice found in most Java application servers implies to run one JVM per application. Due to the fact, that this implies an overhead of computational resources, several, most often proprietary, solutions for application isolation in JVMs have been proposed in the past. This publication outlines isolation properties of a new JVM type, called a JVM Server that is integrated into the Customer Information Control System (CICS) [1]. It implements the OSGi framework [2] and leads to major improvements in terms of application isolation. However, it is also shown that several known problems remain unsolved. Within this paper, solutions to these isolation exposures are proposed. Some of them are not based on the official Java release; others use existing CICS services for program control and are thus not applicable across the board.

Zusammenfassung

Die Java Virtual Machine (JVM) ist in der Lage, mit Hilfe von Threads mehrere Anwendungsprogramme gleichzeitig zu betreuen. Dies kann kritisch sein, wenn es sich dabei um Transaktionen handelt, da die JVM keine echte Applikationsisolation ermöglicht. Das Hosting eines einzigen Anwendungsprogramms pro JVM ist deshalb eine übliche Vorgehensweise, was häufig einen ineffizienten Verbrauch von Rechenleistung zur Folge hat. Aus diesem Grund wurden bereits mehrere, meist proprietäre, Lösungen zur Applikationsisolation in JVMs vorgestellt. Diese Veröffentlichung beschreibt die Isolationseigenschaften eines neuen Typs der JVM, welche in das Customer Information Control System (CICS) [1] integriert ist. Sie implementiert das OSGi Framework [2] und erreicht damit wichtige Verbesserungen. Es wird aber gezeigt, dass viele bekannte Isolationsprobleme weiterhin bestehen. Hierzu werden praktische Lösungsansätze als Abhilfe vorgeschlagen. Einige sind nicht Bestandteil des existierenden Java Standards. Andere nutzen bestehende CICS Services zur Kommunikation zwischen Programmen und sind somit nicht universell einsetzbar. Auf die Notwendigkeit einer langfristigen Lösung wird hingewiesen.

1. Einleitung

Unternehmen und staatliche Organisationen verfügen heute über eine komplexe IT Infrastruktur. In großen Unternehmen besteht diese häufig aus Tausenden von Servern, zuständig für Aufgaben wie Sicherung des Netzwerks und der Applikationen, Routing und Switching, sowie das Hosting von (Web-)Applikationen. Dabei erfolgt die zentrale Datenhaltung auf einem oder mehreren Mainframe Servern (deutsche Bezeichnung Großrechner) mit dem z/OS Betriebssystem. Es besteht eine kritische Abhängigkeit der Unternehmen von ihren Datenbeständen. Deshalb haben Anwendungsprogramme, welche die Datenbestände einer Organisation abändern, meistens transaktionale Eigenschaften. Ein Anwendungsprogramm ist dann eine Transaktion, wenn es die ACID Eigenschaften (Atomicity, Consistency, Isolation, Durability) erfüllt [3].

Transaktionen werden in den meisten Fällen mittels einer Middleware ausgeführt, die als Transaktionsmonitor oder Transaktions-Server bezeichnet wird. Diese stellt unter anderem mit Hilfe von Scheduling- und Synchronisationsmechanismen sicher, dass jede Transaktion strikt die ACID Eigenschaften implementiert. Zudem stellen die meisten Transaktionsmonitore bestimmte Services zur Kommunikation zwischen

Anwendungen zur Verfügung. Darüber hinaus wird die Wiederherstellung des Datenbestands nach fehlerhaft ausgeführten Transaktionen ermöglicht (Stichwort Rollback).

Verbreitete Transaktionsserver sind Tuxedo (BEA/Oracle), CICS (IBM), MTS (Microsoft) und SAP R/3. In den letzten 12 Jahren entstanden zusätzlich Web Application Server, welche für Transaktionen in der Programmiersprache Java entwickelt wurden. Beispiele sind Weblogic (BEA/Oracle), WebSphere (IBM) und NetWeaver (SAP) sowie Open Source Produkte wie Geronimo, Glassfish und JBoss.

Transaktionen sind häufig „unternehmenskritische“ Anwendungen. Wir bezeichnen damit solche Anwendungen, deren fehlerhafte Ausführung zu nicht tolerierbaren Konsequenzen führt. Ein Beispiel ist ein Rundungsfehler in der letzten Dezimalstelle bei der Ausführung einer finanziellen Transaktion. Falls dieser Fehler auftritt, muss er, notfalls in tagelanger manueller Arbeit, identifiziert und rückgängig gemacht werden.

2. Problemdefinition

Ein transaktionales Anwendungsprogramm besteht häufig aus den zwei Komponenten: Präsentationslogik und Geschäftslogik (Business Logic). Während die Geschäftslogik die eigentliche Arbeit verrichtet, dient die Präsentationslogik dazu, die Ergebnisse der Geschäftslogik auf eine attraktive Art auf dem Bildschirm darzustellen. In der populären Model View Controller Triade entspricht das Modell der Geschäftslogik und der View der Präsentationslogik [4].

Seit der Einführung von Java im Jahr 1995 existiert ein Interesse, die Sprache auch für transaktionale Anwendungen einzusetzen. Hierzu erschien 1999 die als Java Enterprise Edition (JEE) bezeichnete Erweiterung. Während Java für die Implementierung der Präsentationslogik seitdem einen Siegeszug angetreten hat, erfolgte der Einsatz für die Geschäftslogik unternehmenskritischer Anwendungen bisher nur sehr zögerlich.

Hierfür existieren mehrere Gründe. Ein wichtiger Grund sind unzureichende Eigenschaften, welche die Isolation (I in ACID) der Transaktionen sicherstellen.

Isolationsprobleme der JVM sind seit Einführung des JEE-Standards bekannt, siehe z. B.:

„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 [5]“.

Oder:

„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 [6]“.

3. Bisherige Lösungsansätze

In der Vergangenheit sind eine Reihe von Lösungsvorschlägen entwickelt worden. Diese wurden jedoch häufig wieder verworfen.

Die meisten Isolationsprobleme einer JVM basieren auf der Tatsache, dass die JVM über keine Ressourcen-Management Funktionen verfügt. Spezifisch wurde die JVM nicht für den Betrieb mehrerer Applikationen entworfen. Aus diesem Grund läuft in vielen Applikationsservern pro Anwendung eine virtuelle Maschine. Dadurch wird sichergestellt, dass Anwendungen sich gegenseitig nicht beeinflussen. Dieses Vorgehen führt jedoch zu einer ineffizienten Nutzung der Ressourcen, insbesondere im Zusammenhang mit der Erstellung und dem Abbau der einzelnen JVMs. Ein vielversprechender Ansatz zur Lösung der Isolationsprobleme war die Application Isolation API [7]. Obwohl dieser Ansatz die meisten Isolationsprobleme beseitigte, wurde die Application Isolation API bisher in keinem offiziellen Java Release veröffentlicht.

Ein weiterer interessanter Ansatz zur Lösung bestehender Isolationsprobleme in JVMs war die Persistent Reusable Java Virtual Machine (PRJVM) [8], die speziell für den CICS Transaktions-Server entwickelt wurde. Die PRJVM implementierte einen Mechanismus für die Wiederherstellung des Status der JVM vor etwaigen Programmausführungen. Obwohl dieser Ansatz einen Ausgleich zwischen effizientem Ressourcenverbrauch und Applikationsisolation darstellte, wurde die PRJVM von einer neuen JVM-Implementierung, dem „JVM Server", ersetzt. Ein JVM Server ist eine Laufzeitumgebung der JVM innerhalb von CICS, die eine Standard JVM mit einem OSGi Framework enthält. Anwendungen laufen innerhalb dieses Frameworks als so genannte Bundles. Ein Bundle besteht aus einem Java Archive (jar) und einer Inhaltsbeschreibung (MANIFEST.MF).

Die vorliegende Veröffentlichung beschreibt im folgenden einige der heute existierenden Isolationsprobleme und diskutiert Lösungsmöglichkeiten. Manche Lösungsansätze benötigen hierzu Funktionen der Middleware, in der die JVM eingesetzt wird, und sind deshalb nicht portierbar. Unsere Untersuchungen basieren auf die Version der JVM, welche in dem „JVM Server" des CICS Transaction Servers Version 4.2 seit 2011 verfügbar ist.

CICS (Customer Information Control System) ist ein Transaktionsmonitor der Firma IBM mit Applikationsserver Funktionalität. Die wichtigsten Gründe für den Einsatz von CICS als Applikationsserver sind, neben seiner weiten Verbreitung in Enterprise Systemen, die zahlreichen verfügbaren Services, die unter anderem Transaktionssicherheit ermöglichen. Darüber hinaus agiert CICS als Middleware und ermöglicht dadurch den Betrieb unterschiedlicher Anwendungen ungeachtet der verwendeten Programmiersprache. Damit ist eine einheitliche Kommunikation zwischen Anwendungen möglich.

4. Java Integration in CICS

CICS unterstützt seit 1998 das Hosting von Java Applikationen [9]. Typisch basiert die Java Unterstützung auf einem Pool von JVMs. Diese Lösung wird in Zukunft durch den JVM Server ersetzt. Im Folgenden werden die zwei verschiedenen Modelle der Java Unterstützung beschrieben.

4.1 Pooled JVMs

Ein Pool von JVMs ist eine Laufzeitumgebung für eine vordefinierte Anzahl an JVMs innerhalb vom CICS. Diese Laufzeitumgebung stellt sicher, dass in jeder JVM nur eine Anwendung, bzw. Transaktion, gleichzeitig aktiv ist. Die JVMs werden von den einzelnen Transaktionen seriell verwendet (siehe Abbildung 1).

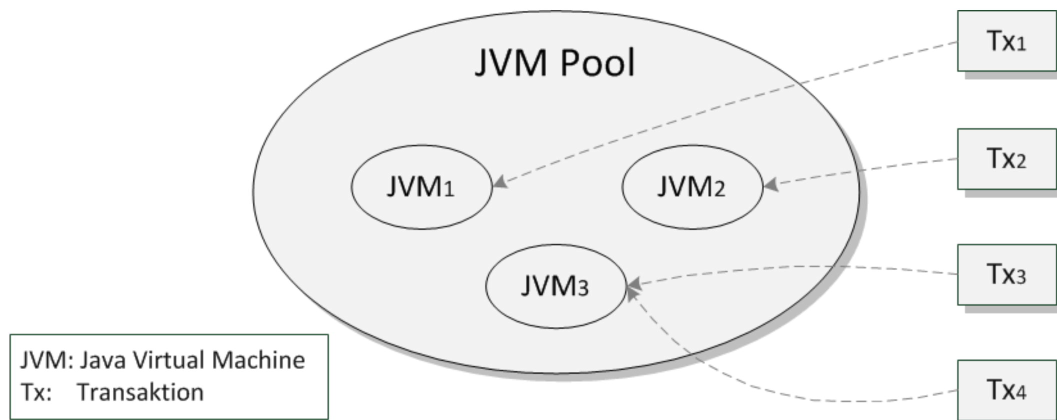

Abb. 1: Grafische Darstellung der seriellen Verwendung von Pooled JVMs durch mehrere Transaktionen. Nach [10].

Wie in [10] beschrieben haben die JVMs innerhalb des Pools drei verschiedene Betriebsmodi:

- Single Use
  - Neuerstellung der JVM nach erfolgter Transaktion
- Reusable
  - Wiederverwendung der JVM für mehrere aufeinanderfolgende Transaktionen
- Resettable (auch Persistent Reusable Virtual Machine genannt, siehe [8])
  - Wiederherstellung des Zustandes der JVM nach erfolgter Transaktion

Auf einem Hochleistungsserver werden mehrere hundert oder tausend Transaktionen pro Sekunde verarbeitet. Trotz der Isolationsvorteile des „Single Use“ Modus, ist dieser für viele Anwendungen nicht geeignet, da die Dauer der Erstellung einer neuen JVMs im hohen Millisekundenbereich angesiedelt ist. Auch der Reusable Modus birgt Nachteile. Beispielsweise kann eine Anwendung durch die Modifikation einer statischen Systemvariable einen direkten Einfluss auf nachfolgende Anwendungen haben, falls diese nicht wieder zurückgesetzt wird. Unter Berücksichtigung der Eigenschaft, dass der Resettable Mechanismus ebenfalls Zeit in Anspruch nimmt, kann auch der Resettable Modus nur bedingt als geeigneter Lösungsansatz der Isolationsprobleme betrachtet werden.
Die Ressourcenprobleme in Pooled JVMs werden durch den neuen CICS JVM Server verbessert.

4.2 JVM Server

Ein JVM Server ist eine Laufzeitumgebung für die Ausführung von CICS Java Applikationen. Diese beinhaltet nur eine JVM und wird von mehreren Anwendungen parallel verwendet. Zudem enthält der JVM Server ein OSGi Framework (siehe Abbildung 2), welches die Isolation und die Modularität von Anwendungen ermöglicht. Dabei wird die Isolation mit Hilfe von Class Loadern bewerkstelligt, eine Methode, die bereits 1997 von Balfanz und Gong [11] beschrieben wurde und auch in JEE verwendet wird [12]. Dieser Ansatz basiert auf der Eigenschaft, dass Objekte, die von unterschiedlichen Class Loadern erstellt wurden, sich in unterschiedlichen Namensräumen befinden und somit keinerlei Kenntnis voneinander besitzen. Wie jedoch in [13] beschrieben, besteht kein Schutz gegen eine Überschneidung der Namensräume (siehe Kapitel 5).

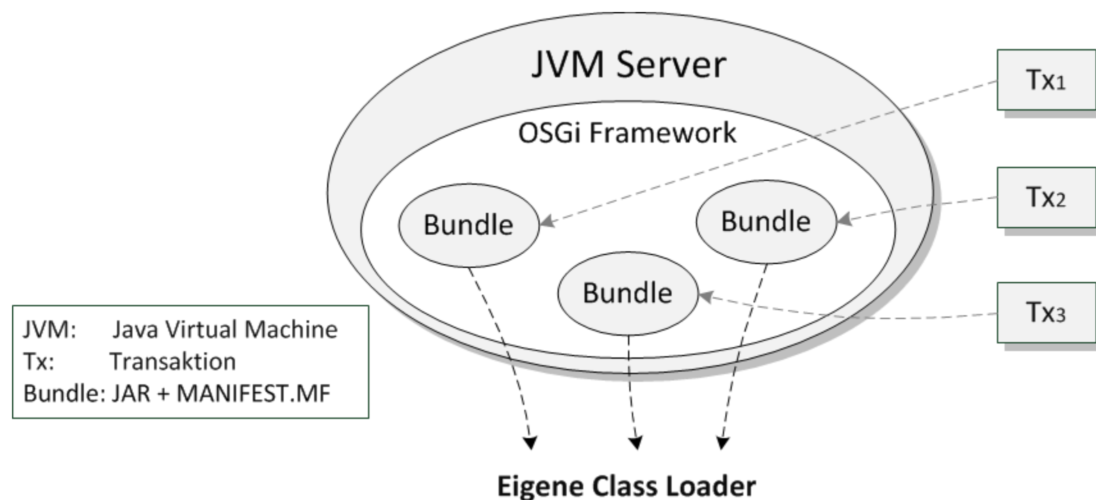

Abb. 2: Grafische Darstellung der JVM Server Laufzeitumgebung in CICS

Das OSGi Framework kapselt Anwendungen bzw. Funktionen (je nach gewünschter Granularität) in sogenannte Bundles, die laut [10] auch als Plain Old Java Objects (POJOs) [14] betrachtet werden können. Ein Bundle hat mehrere Zustände, darunter Uninstalled, Installed, Resolved und Active [2]. Die Administration der Zustände der Bundles innerhalb einer aktiven JVM erfolgt mit Hilfe des OSGi Frameworks. Dies ermöglicht die Installation einer neuen Anwendung in den JVM Server ohne Neustart der JVM. Im Gegensatz dazu ist bei den meisten Java Applikationsservern ein Neustart für ein erfolgreiches Deployment einer neuen Anwendung notwendig.

Die Zustände der Bundles werden in der Regel über ein Interface gesteuert – die OSGi Konsole. Im Falle des JVM Servers wird dieses Interface durch den CICS Explorer, eine auf Eclipse basierende Anwendung, repräsentiert.

Da jedes Bundle einen eigenen Class Loader besitzt, sind Objekte standardmäßig voneinander isoliert. Diese Art der Isolation ist jedoch unvollständig [15]. Im folgenden Abschnitt wird dies näher erläutert.

5. Offene Isolationsprobleme in JVMs

In unseren Untersuchungen fanden wir mehr als ein Dutzend Schwachstellen, die bei der Erstellung von transaktionalen Anwendungen zu Verletzungen der Isolationseigenschaften führen können. Diese Schwachstellen gliedern sich in die folgenden Bereiche:

- Probleme mit Systemklassen
- Überschneidende Namensräume
- Probleme durch das Java Native Interface
- Übermäßiger Ressourcenverbrauch
- Probleme mit Activator Klassen
- Unrechtmäßige Kontrolle

Um Abhilfe unter Einbindung bestehender real existierender Möglichkeiten im Bezug auf diese offenen Isolationsprobleme zu schaffen, wurden praktische Lösungsansätze untersucht. Dazu gehören

- statische Programmanalyse Tools, die den möglichen Status einer Applikation ohne spezifische Eingabedaten überprüfen [16].
- Programmkontrolle mit Hilfe des Java Security Managers.
- Monitoring Tools, die den Zustand der JVM in Echtzeit überwachen, z. B. das unter CICS verfügbare Java Health Center [17].
- Einsatz von plattformabhängigem Code [18].

Der Einsatz dieser Lösungsansätze ist jedoch insgesamt problematisch. Sie sind nicht Bestandteil des Java Standards, arbeiten nicht zuverlässig, verschlechtern das Leistungsverhalten und/oder verletzen die Plattformneutralität.

Eine Diskussion der offenen Isolationsprobleme und der von uns vorgeschlagenen Lösungsansätze würde den Rahmen dieser Veröffentlichung sprengen. Einzelheiten sind in [18] oder in einer parallel im Java Spektrum erscheinenden Veröffentlichung [19] beschrieben.

6. Stand der Technik

6.1 Transaktionseigenschaften

Java hat seit der Einführung 1995 einen Siegeszug angetreten. Die JSE und die JME sind universell akzeptiert und gelten als bevorzugte Programmiersprache vor allem bei der jungen Generation von Programmierern.

Für Enterprise Anwendungen wurde JEE 1999 eingeführt. Der wesentliche Unterschied zu der JSE sind die Enterprise Java Beans (EJB), welche über standardisierte Schnittstellen des EJB Containers Dienste wie JNDI (Java Naming directory Interface), JMS (Java Messaging Service) oder JIDL (Java Interface Definition Language) zur Verfügung stellen. Von zentraler Bedeutung ist der Java Transaction Service (JTS), eine Schnittstelle für einen JEE Transaktionsmanager, der ebenfalls die X/Open XA Interface bereitstellt.

Transaktionseigenschaften sind eine Voraussetzung für die meisten Enterprise Anwendungen, und hierbei ist die Isolation von kritischer Wichtigkeit. An dieser Stelle hinterlässt die JEE den Eindruck einer Baustelle. Dies gilt besonders für den 2006 erfolgten Übergang von EJB 2.0 nach EJB 3.0. Letzteres beinhaltete wesentlich Änderungen, wie z.B. die Eliminierung der Entity Beans oder eine größere Rolle für POJOs (Plain Old Java Objects).

6.2 Lebensdauer

Ein wichtiger Unterschied zwischen der JEE und anderen Java Editionen ist die erwartete Lebensdauer der damit entwickelten Anwendungen. Bei manchen Smartphones und Tablets wird akzeptiert, dass Software für ältere Modelle aufgrund deren begrenzten Langzeitsupports nicht mehr verfügbar ist. Der ideale Kunde eines Smartphone Herstellers ersetzt sein Mobiltelefon alle 24 Monate durch ein neues, verbessertes Modell, obwohl das alte Modell noch voll funktionsfähig ist. Inkompatibilitäten zwischen aufeinander folgenden Software Versionen sind in dieser Situation wegen des schnelllebigen Marktes kein gravierendes Problem.

Im Gegensatz dazu hat Enterprise Software eine Lebensdauer, die in Jahrzehnten gemessen wird. Ein Testfall waren die „Jahr 2000“ (y2k) Umstellungen von zweistelligen auf vierstellige Jahreszahlen. Obwohl der erforderliche Umstellungsaufwand von weltweit 300 Mrd. $ ein guter Anlass gewesen wäre, veraltete Software durch neuere, moderne Versionen zu ersetzen, ist dies fast nirgendwo geschehen [20].

In der Vergangenheit verstand man unter dem Begriff „Legacy Software“ Anwendungen, die auf Mainframe Rechnern ausgeführt wurden. In zunehmendem Maße existieren in den Unternehmen jedoch Legacy Anwendungen, die auf Windows (in unterschiedlichen Versionen), Linux, AIX, HP_UX, Solaris, Tru64 UNIX und vielen weiteren Betriebssystemen laufen.

Es wird geschätzt, dass Unternehmen etwa 50 % ihres jährlichen Budgets für Anwendungssoftware in Neuentwicklungen und 50 % in die Administration und Pflege von Legacy Software investieren [21].

6.3 Java

Inkompatibilitäten zwischen aufeinander folgenden Java Versionen sind ein Problem, wenn die Lebensdauer von Anwendungen in Jahrzehnten gemessen wird. Mehrfache Entwicklungsumgebungen müssen unterhalten werden. Doug Lyon berichtet, dass die Anzahl von Deprecations zwischen JDK 1.1 und JDK 1.6 von 41 auf 1 868 angestiegen ist, was einer jährlichen Wachstumsrate von 46 % entspricht [22]. Ebenso sind zahlreiche Frameworks heute nicht mehr verfügbar oder werden in Zukunft möglicherweise nicht mehr verfügbar sein.

6.4 Cobol

Von Edsger Dijkstra existiert die berühmte Aussage „The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence“[23].

Das war 1968. Seitdem ist die Benutzung von Cobol weiter gewachsen, bei weitem nicht so stark wie Java, oder auch Visual Basic und C#, aber dennoch gewachsen.

Die weltweit derzeitig existierenden und regelmäßig genutzten Cobol Programme haben einen Umfang von 250 Milliarden Lines of Code, und der Bestand wächst jährlich um etwa 5 Milliarden [24]. Ein Grund hierfür ist vermutlich, dass man Cobol Code einen besonders geringen Wartungsaufwand unterstellt. Die Jahr-2000 Umstellungskosten pro Function Point betrugen im Durchschnitt für alle Sprachen 45 $, für Cobol Programme jedoch nur 28 $ [25].

Im Unterschied zu allen anderen Sprachen kann Cobol Code durch Phoneme ausgedrückt werden. Nach E. Arranga ist dies eine der Erklärungen für die überlegene Wartbarkeit von Cobol Anwendungen [26].

7. Zusammenfassung und Ausblick

7.1 Zusammenfassung

Seit Jahren setzt sich die Firma IBM sehr stark für den Einsatz von Java bei der Entwicklung von neuen unternehmenskritischen transaktionalen Anwendungen ein; bisher jedoch nur mit begrenztem Erfolg.

Die fortschreitende Entwicklung von Java wird stark durch die Bedürfnisse des Internets getrieben, bei denen man eine kurze Lebensdauer für neue Anwendungen unterstellt. Für Anwendungen im Enterprise Bereich wird es erforderlich sein, die Wartungsfreundlichkeit von Java für einen Zeitraum von Jahrzehnten zu überprüfen, und ggf. mittels geeigneter Erweiterungen zu verbessern. Langfristig wird sich Java an der Wartungsfreundlichkeit von Cobol orientieren müssen.

Die in diesem Aufsatz aufgezeigten Isolationsprobleme bedürfen einer grundsätzlichen Verbesserung; die ebenfalls hier vorgeschlagenen Lösungen sind eher als temporäre Fixes zu betrachten. Bemerkenswert ist, dass ein seit 10 Jahren bekanntes Problem bis heute nicht zufriedenstellend gelöst wurde. Zu hoffen ist, dass die zu erwartende Erweiterung/Verbesserung des JEE Standards ohne größere Inkompatibilitäten mit den bisherigen Versionen erreicht werden kann.

7.2 Ausblick

Es ist interessant zu spekulieren, wie die zukünftige Entwicklung der JVM aussehen könnte.

Die heutige JVM ist ein Zwitter: Sie implementiert auf der einen Seite einen Emulator, der einen Betrieb auf unterschiedlichen Hardware Architekturen ermöglicht. Auf der anderen Seite stellt sie die Funktionen eines sehr einfachen Betriebssystems zur Verfügung, welches über einen einzigen Adressenraum verfügt.

Denkbar wäre es, eine saubere Trennung zwischen Emulator und Betriebssystem Funktionen einzuführen. Letztere könnten nach dem Vorbild des OSGi Frameworks als eine Art Middleware innerhalb der JVM implementiert werden.

Dies sind einige Ideen hierzu:

Das JVM Betriebssystem könnte ein Single Address Space Multithreaded Betriebssystem sein, welches sein eigenes Prozess/Thread Management und sein eigenes Resource Management durchführen würde. Nach dem Vorbild des z/OS CICS Transaktionsservers könnte der Nucleus dieses JVM Betriebssystems die folgenden Eigenschaften haben:

- Die wichtigsten Komponenten des JVM Nucleus wären Communication Management (RMI), Thread Management, Main Storage Management, Class Loader Management und External Storage Management (Persistence Framework).
- Java Threads werden vom JVM Nucleus und nicht vom darunter liegenden Betriebssystem Kernel verwaltet, so lange run-to-completion garantiert ist.
- In Situationen in welchen dies nicht gewährleistet werden kann, wird analog zu CICS ein QR TCB versus Open TCB Mechanismus eingeführt [27].
- Der Open TCB Mechanismus würde gleichzeitig die parallele Ausführung von Java Threads innerhalb einer JVM auf mehreren CPUs ermöglichen.
- Hauptspeicherplatz ist realer Speicher aus der Sicht des JVM Nucleus, aber virtueller Speicherplatz aus der Sicht des Betriebssystem Kernels. Nach dem Vorbild des CICS Storage Managers würde der JVM Nucleus seinen Hauptspeicherplatz dynamisch verwalten [28].
- Ebenfalls nach dem Vorbild von CICS würde die Application Isolation mit Hilfe eines z/OS-ähnlichen Enclave Mechanismus sichergestellt. Problematisch ist an dieser Stelle, dass Dokumentation über Enclaves kaum verfügbar ist.
- Die Firmen Intel und AMD werden motiviert, nach dem Vorbild der S/360 Architektur (Baujahr 1964!) Storage Protection Keys einzuführen. Damit ist es zunächst möglich, den JVM Nucleus vor Java Threads zu schützen. Zusätzlich könnten aber damit auch die Heaps und Stacks der einzelnen Threads gegeneinander geschützt werden.
  Wir gehen davon aus, dass Intel/AMD ohnehin zur Verbesserung der Buffer Overflow Problematik nicht umhin kommen werden, Storage Protection Keys als Bestandteil der x86 Architektur einzuführen.
- Auch eine Synthese moderner Virtualisierungs-Technologien mit den derzeitigen JVM Eigenschaften ist eine interessante Spekulation.

Die hier vorgestellte Vision hätte einen Vorteil: Sie hat das Potential, ohne größere „Deprecate“ und Downward Kompatibilitäts-Probleme implementiert werden zu können. Die zusätzlichen Stabilitätseigenschaften würden auch einen Beitrag zur Verbesserung der Wartbarkeit von Java Anwendungen leisten.

8. Danksagung

Die Autoren danken Isabel Arnold, Philipp Breitbach, Uwe Denneler, Frank Güttler, Wilfried van Hecke, Tobias Leicher und Ulrich Seelbach für ihre tatkräftige Unterstützung.

9. Abkürzungen

ACID Atomicity, Consistency, Isolation, Durability
CICS Customer Information Control System
EJB Enterprise Java Bean

JSE	Java Standard Edition
JEE	Java Enterprise Edition
JME	Java Micro Edition
JVM	Java Virtual Machine
OSGi	Open Services Gateway initiative
PRJVM	Persistent Reusable Java Virtual Machine
Y2K	Year 2000

Literatur

[1] Rayns C, Bertram S, Bogner G, Carlin C, Clark A, Ferrell A, Keehn G, Klein P, Lee R, Woerner E. CICS Transaction Server from Start to Finish. IBM International Technical Support Organization (ITSO), Dezember 2011. http://www.redbooks.ibm.com/redbooks/pdfs/sg247952.pdf

[2] OSGi Alliance. OSGi Service Platform Core Specification Release 4 Version 4.3, April 2011. http://www.osgi.org/download/r4v41/r4.core.pdf

[3] Gray J, Reuter A. Transaction Processing. Morgan, Kaufman, 1993

[4] Mössenböck H. Objektorientierte Programmierung. Springer, 1993, ISBN 3-540-55690-7

[5] 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. ACM Press, New York, 2000, S. 354–366.
http: //research.sun.com/projects/barcelona/papers/oopsla00.pdf

[6] Sandén B. Coping with Java Threads. IEEE Computer, Vol. 37, Nr. 4, April 2004, S. 20

[7] Java Community Process. Sr-000121 Application Isolation API Specification.
Webseite, Februar 2006. http://jcp.org/aboutJava/communityprocess/final/jsr121/index.html

[8] IBM Corp. New IBM Technology featuring Persistent Reusable Java Virtual Machines, Oktober 2001. http://www-03.ibm.com/systems/resources/servers_eserver_zseries_software_java_pdf_prjvm13.pdf

[9] IBM Corp. Techreport: History of CICS. Webseite, 2011.
https://www-304.ibm.com/support/docview.wss?uid=swg21025234

[10] IBM Corp. Java Applications in CICS (Version 4 Release 2), 2011.
http://publib.boulder.ibm.com/infocenter/cicsts/v4r2/topic/com.ibm.cics.ts.java.doc/dfhpj_pdf.pdf

[11] Balfanz D, Gong L. Experience with Secure Multi-Processing in Java. Technical report, Department of Computer Science, Princeton University, September 1997. http://sip.cs.princeton.edu/pub/icdcs.pdf

[12] Chinnici R, Shannon B. Java Platform, Enterprise Edition (JavaEE) Specification v6, Oktober 2009. http://download.oracle.com/otndocs/jcp/javaee-6.0- fr-oth-JSpec/

[13] McGraw G, Felden E. Securing Java. John Wiley & Sons, Inc., Januar 1999.
http://www.securingjava.com/

[14] Fowler M. Webseite, 2000. http://www.martinfowler.com/bliki/POJO.html

[15] Mueller J. Anwendungs- und Transaktionsisolation unter Java, Mai 2005. http://www-ti.informatik.uni-tuebingen.de/~spruth/DiplArb/jmueller.pdf

[16] Reps T, Sagiv M, Wilhelm R. Static Program Analysis via 3-Valued Logic. In Computer Aided Verification, volume 3114 of Lecture Notes in Computer Science, S. 401–404. Springer Berlin / Heidelberg, 2004. http://groups.csail.mit.edu/cag/crg/papers/reps04shape.pdf

[17] IBM Corp. IBM Monitoring and Diagnostic Tools for Java - Health Center Version 2.0. Webseite. http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/

[18] Harbach R. CICS JVM Server Application Isolation. Masterarbeit am Wilhelm-Schickard-Institut für Informatik der Universität Tübingen, März 2012.
http://www-ti.informatik.uni-tuebingen.de/~spruth/DiplArb/harbach.pdf

[19] JAVA SPEKTRUM Veröffentlichung erscheint demnächst.

[20] Wikipedia. Year 2000 problem. http://en.wikipedia.org/wiki/Year_2000_problem

[21] Sneed H. Offering Software Maintenance as an Offshore Service. 24th International conference on Software Maintenance, ISCM, Beijing, September 2008.
http://www.icsm2008.org/downloads/sneed.pdf

[22] Lyon D. The Java Tree Withers. IEEE Computer, Januar 2012, S.83-85

[23] Dijkstra E. How do we tell truths that might hurt?, 18 Juni 1975
http://www.cs.virginia.edu/~evans/cs655/readings/ewd498.html

[24] MicroFocus. How COBOL has stood the test of time.
http://blog.microfocus.com/how-cobol-has-stood-the-test-of-time/1285/

[25] Kappelman L. Some strategic Y2K blessings. Journal IEEE Software,
http://dl.acm.org/citation.cfm?id=624636.626107, Volume 17 Issue 2, März 2000, S. 42 – 46

[26] Arranga E. Fresh from Y2K, What's Next for Cobol?
Journal IEEE Software, Volume 17 Issue 2, März 2000, S. 16 – 20

[27] Rayns C, Bogner G, Hale P, James E, Klein P, Tilling J. Threadsafe Considerations for CICS. IBM International Technical Support Organization (ITSO).
http://www.redbooks.ibm.com/redbooks/pdfs/sg246351.pdf

[28] Horswill J. Designing and Programming CICS Applications. O'Reilly, 2000

Alle in digitaler Form verfügbaren Aufsätze sind zusätzlich in einem Mirror unter
http://www-ti.informatik.uni-tuebingen.de/~spruth/Mirror/Informatik_Spektrum/index.html
abrufbar.