Problematik und Lösungsansätze für Transaktionssicherheit und Stabilität bei J2EE Systemen

Oliver Raible, Wilhelm G. Spruth

Abstrakt

Die Java Plattform, Enterprise Edition (Java EE, frühere Bezeichnung: J2EE) wurde entwickelt, um den Einsatz von Java in unternehmenskritischen Transaktionsanwendungen zu ermöglichen. Die Stabilität und Zuverlässigkeit einer J2EE-Anwendung, sowie die unbedingte Gewährleistung von Transaktionssicherheit bei parallel laufenden Transaktionen ist jedoch als problematisch einzustufen. Der vorliegende Beitrag analysiert die existierenden Schwierigkeiten und beschreibt einen neuartigen Lösungsansatz. Dieser ist geeignet um sowohl J2EE-Implementierungen, als auch ganze J2EE-Anwendungssysteme auf Transaktionssicherheit, Stabilität und Zuverlässigkeit hin zu prüfen. Darüber hinaus lassen sich durch die gesammelten Testdaten und Auswertungen früh Fehlerquellen reduzieren sowie Optimierungspotentiale aufzeigen

1 Problematik der Isolation und existierende Lösungsansätze

Unter Transaktionssicherheit wird die qualitative Eigenschaft eines transaktions-verarbeitenden Systems verstanden, die gesicherte und isolierte Verarbeitung von Transaktionen zu ermöglichen. Für ein gutes Leistungsverhalten ist die parallele Ausführung von Transaktionen erforderlich. Die Sun HotSpot JVM unterstützt die Ausführung paralleler Java Threads und wurde spezifisch in Hinblick auf Leistungssteigerungen entwickelt. Dies ermöglicht einen erheblichen Leistungsgewinn bei der Thread Synchronisation. Dabei ist wichtig, dass vorangegangene, parallel durchgeführte oder fehlgeschlagene Transaktionen sich nicht außerhalb der transaktional veränderten Datenbasis der betreffenden Datenquellen beeinflussen. [Borm01] führt zu dieser Isolationsproblematik durch das System die Merkmale *Beeinflussung* und *Zuverlässigkeit* ein. *Beeinflussung* geschieht durch Manipulation des globalen Systemzustands durch eine erste Transaktion (z.B. statische Variable) und Ausführung einer zweiten Transaktion innerhalb dieses geänderten Systemzustands mit Auswirkung auf das Ergebnis. Die *Zuverlässigkeit* adressiert die Fortpflanzung von Fehlern einer Transaktion auf alle momentan parallel ausgeführten Transaktionen (z.B. kritische JNI-Fehler in der JVM oder Speicherknappheit).

Die Nutzung von Java Threads bei der Transaktionsverarbeitung ist wegen der Isolations-Forderung problematisch, siehe z.B. [Cza00] und [San04]. Es existieren einige Lösungsansätze für diese Problematik. Die Isolation mittels Classloader (bzw. Classloader-Hierarchien) wird in fast allen J2EE-Implementierungen eingesetzt, zusammen mit einem Multi-Threading-Ansatz zur parallelen Abarbeitung von Client-Anfragen.

Des Weiteren sind in der EJB-Spezifikation einige Restriktionen bzw. Garantien aufgeführt, welche die Isolation und Stabilität zur Laufzeit sicherstellen sollen. Dazu gehören u. a. keine Verwendung von statischen Klassenvariablen innerhalb von EJBs, keine Thread-Erzeugung oder JNI-Aufrufe innerhalb von EJBs. Kritische Funktionalitäten wie JNI-Aufrufe und Dienste wie Transaktionsverarbeitung werden aus der Anwendung heraus in die

„vertrauenswürdige“ J2EE-Infrastruktur verlegt. Die Vorteile des Classloader-Ansatzes sind in der hohen Performanz und dem verhältnismäßig geringen Ressourcenverbrauch zu sehen. Nachteilig ist die vergleichsweise hohe Verantwortung der Anwendungsentwicklung für die Isolation sowie die durch die höhere Anzahl der LOCs gestiegene Fehlerwahrscheinlichkeit der Isolations-Implementierung.

Der Lösungsansatz, die Isolation mittels einer Erweiterung der JVM sicherzustellen, wird in der *Persistent Reusable JVM* (PRJVM) angewendet [Borm01]. Die Vorteile dieses Ansatzes sind vor allem durch den hohen Grad der Isolation durch das Betriebssystem und durch die *Reset*-Fähigkeit der JVM gegeben, welche damit sowohl die Isolationsproblematik *Beeinflussung* als auch die Isolationsproblematik *Zuverlässigkeit* adressiert. Nachteilig ist, dass die PRJVM bis jetzt nur auf der z/OS Plattform verfügbar ist.

In [Cza00] wird die Isolation durch Modifikationen der JVM sichergestellt. Auf Ebene der JVM werden statische Klassenvariablen pro Thread vervielfältigt und lokal gehalten, sodass eine Beeinflussung von Transaktionen durch dieses Mittel nicht mehr möglich ist. Die Zuverlässigkeit wird wie im J2EE-Fall innerhalb der JVM realisiert. Vorteile gegenüber dem Classloader-Ansatz sind zunächst eine höhere Performanz und ein optimaler Ressourcenverbrauch durch das nur einmalige Laden der Klassen. Dazu kommt die automatische Verfügbarkeit der Isolation (Reduzierung der Verantwortung der Anwendungsentwicklung). Nachteilig ist, dass die Isolation bzgl. Beeinflussung von nachfolgenden Transaktionen (*Zuverlässigkeit)* nicht direkt gegeben ist.

Die kürzlich fertiggestellte Isolate-API (*Java Specification Request* 121) definiert eine Basis API für die Isolation von Komponenten. Java-Anwendungen werden unabhängig voneinander innerhalb von *Isolates* ausgeführt, welche keine Objektinstanzen mit anderen *Isolates* teilen können. Wie Threads, ermöglichen *Isolates* die nebenläufige Ausführung von Anwendungen und lassen sich erzeugen, starten, beenden und managen. Darüber hinaus verfügen sie wie eine JVM über einen eigenen *System-Level*-Kontext.

Die Güte und Art der Isolation kommt dabei stark auf die Implementierung selbst an. Die in [Cza00] vorgestellte Modifikation der JVM und die Separation von statischen Klassenvariablen ist die Basis für die Abbildung aller *Isolates* einer JVM auf einen Betriebssystemprozess. Die Technologie der PRJVM benutzt an dieser Stelle eine als *Enclaves* bezeichnete Technologie, die es ermöglicht, mehrere Kopien der JVM innerhalb des gleichen Betriebssystemprozesses abzubilden.

Auf das Problem der Wiederverwendbarkeit einer JVM für aufeinanderfolgende Transaktionen wird in [Bey05] eingegangen.

Allen Ansätzen ist gemeinsam, dass sie stark zentriert auf die Infrastruktur sind. Die Problematik, stabile und zuverlässige Anwendungskomponenten auf Basis einer komplexen Infrastruktur zu entwickeln, wird dabei vernachlässigt. Im Folgenden wird deshalb ein alternativer testzentrierter Ansatz untersucht und vorgestellt.

2 Transaktionssicherheit durch Testsimulationen

Der testzentrierte Ansatz soll die Stabilität und Zuverlässigkeit sowohl der J2EE-Infrastruktur als auch der Anwendungskomponenten hinsichtlich Transaktionsverarbeitung sicherstellen. Aus Ermangelung einer existierenden Testinfrastruktur für die gestellten Anforderungen,

wurde hierzu das *Transaction Testsystem (TTS)* entworfen und implementiert. Das TTS simuliert Unternehmensprozesse und komplexe Fehlersituationen bei Nebenläufigkeit. Es ist in der Lage, verschiedene Transaktionskombinationen mit frei zu definierender Basislast unter Fehlereinwirkung (bis hin zu fatalen Systemfehlern) reproduzierbar zu machen, kontrolliert durchzuführen und auszuwerten. Die Testszenarien und Fehlerfälle können dem System beliebig hinzugefügt werden um damit einem realen Anwendungsszenario beliebig nahe zu kommen. Die zu testende J2EE-Anwendung ist austauschbar; somit kann das TTS für Anwendungstests in jedem Softwareprojekt eingesetzt werden. Es wird damit möglich, reale Problematiken (aus Transaktionsverarbeitung oder genereller Nebenläufigkeit bedingt) kontrolliert und reproduzierbar innerhalb eines Testsystems zu untersuchen. Darüber hinaus kann unter der Prämisse von stabilen und zuverlässigen Anwendungskomponenten die Zuverlässigkeit der verwendeten J2EE-Infrastruktur überprüft werden.

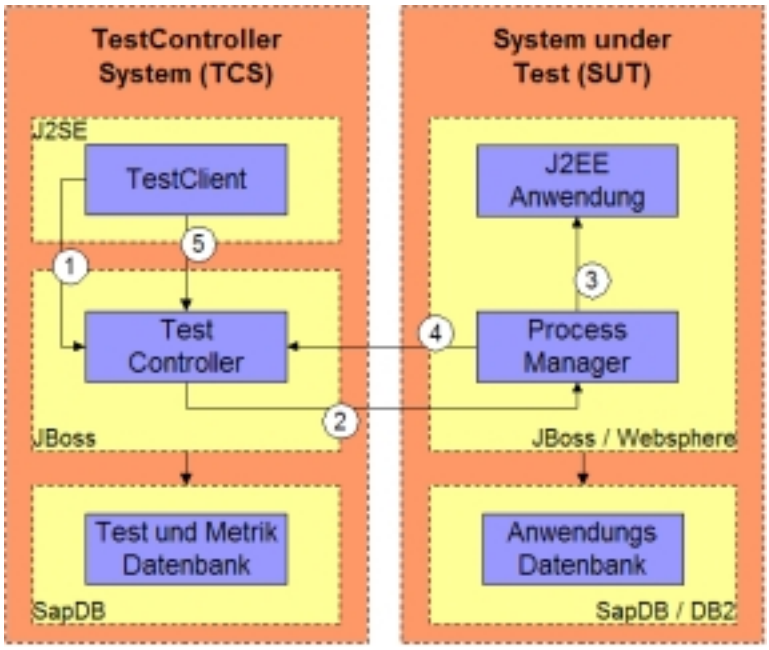

Abbildung 1: Architektur des Transaction Testsystem (TTS)

Das Transaction Testsystem (TTS) besteht aus dem *TestController-System (TCS),* dem *System under Test (SUT)* und einer Reihe von Test- und Fehlerfällen [Ora04].

Das TCS beinhaltet den `TestClient`, welchen ein Tester benutzt um eine Testsuite, bestehend aus mehreren Testfällen, zu erstellen und dem `TestController` zu übergeben (Abbildung 1, 1). Der `TestController` ist ein Dienst innerhalb eines J2EE-Containers und stellt Schnittstellen zum Ausführen und Interagieren mit einer Testsuite bereit. Der `TestController` verwaltet die Testprozesse zu einem Testfall und übergibt diese dem `ProcessManager` zur Ausführung auf dem SUT (Abbildung 1, 2). Ebenfalls steuert er die Basislast gegen das SUT während der Ausführung eines Testprozesses.

Der `ProcessManager` ist die Integrationsschicht zwischen dem TTS und einer beliebigen zu testenden J2EE-Anwendung (Abbildung 1, 3). Während der Ausführung des Testprozesses kommuniziert der `ProcessManager` an bestimmten Synchronisationspunkten mit dem `TestController` (Abbildung 1, 4), um sich mit den parallel ausgeführten Testprozessen desselben Testfalls zu synchronisieren. Die Synchronisation wird vom `TestController` durchgeführt. Wenn ein Testprozess den für ihn vorgesehenen Synchronisationspunkt erreicht

hat, wird der Methodenaufruf vom `TestController` intern blockiert, bis die restlichen Testprozesse ebenfalls ihre Synchronisationspunkte erreicht haben.

Wenn alle Testprozesse ihren Synchronisationspunkt erreicht haben, kehren die Testprozesse in einer definierten Reihenfolge zurück. Bei der Rückkehr der Testprozesse können Fehlerfälle im SUT ausgelöst werden. Bei Fehlern der Fehlerkategorie ERROR (z.B. `NullPointerException`), darf nur der fehlerauslösende Testprozess fehlschlagen und ein Rollback durchführen, alle anderen Testprozesse müssen erfolgreich sein. Bei der Fehlerkategorie FATAL (z.B. *Systemcrash*), laufen alle aktiven Testprozesse auf einen Fehler. Das SUT muss evtl. neu gestartet und auf dem *Resourcemanager* ein *Recovery* durchgeführt werden.

Nach der Ausführung eines Testprozesses, startet das TCS die Überprüfung der Korrektheit der Ergebnisse des Testprozesses sowie die Konsistenzprüfung der Anwendungsdatenbank. Wenn das SUT durch die Ausführung eines FATAL-Fehlers zeitweise nicht verfügbar ist, wird die Validierung solange verzögert. Zuletzt werden die Daten für die Testmetrik und die Vergleichsmetrik gesammelt und in die Metrik-Datenbank des TCS geschrieben.
Der `TestClient` kann jederzeit beim TCS den Status und die Ergebnisse der übergebenen Testsuite abfragen (Abbildung 1, 5).

3 Testdurchführung und Ergebnisse

Um die Eignung des TTS sowohl für die Auswahl einer für das Anwendungsszenario geeigneten J2EE-Infrastruktur als auch für die Sicherstellung von stabilen und verlässlichen Anwendungskomponenten zu untersuchen, wurde das TTS mit folgenden Testsetups auf dem SUT getestet (siehe Abbildung 1):

Testsetup 1, bestehend aus einer Opensource Implementierung mit dem JBoss Applicationserver 3.2.3 und der Datenbank MaxDB 7.4.3.30.

Testsetup 2, bestehend aus einer kommerziellen J2EE-Implementierung mit dem IBM WebSphere Applicationserver 5.1 und der Datenbank DB2 8.1, ebenfalls von IBM.

JBoss als Opensource-Implementierung der J2EE-Spezifikation besteht aus knapp 4.300 Klassen und belegt installiert 50 MB. WebSphere besteht aus über 20.000 Klassen und belegt installiert 460 MB. MaxDB (früher SapDB) wurde ursprünglich von der Software AG entwickelt (Adabas D). Die Datenbank wurde von SAP übernommen und als Opensource-Projekt freigegeben. Seit kurzem ist die Firma MySql AB für die Weiterentwicklung zuständig.

Die JVM (IBM), das Betriebssystem (Windows 2000 Advanced Server) und die Hardware wurden über die gesamte Testdurchführung als statisch betrachtet, um eine Vergleichbarkeit der Performanzmessungen sicherzustellen.

Der TestController des TCS wurde für die Tests auf einer JBoss-Infrastruktur mit Nutzung von optimistischen Synchronisationsverfahren und mit der Persistierung über die MaxDB betrieben. Auf dem SUT wurde ein pessimistisches Sperrverfahren eingesetzt.

Um Informationen über das Verhalten der J2EE-Infrastrukturen unter steigender Last und verschiedenen Konfigurationsoptionen zu erhalten, wurden alle Testszenarien wiederholt ausgeführt, unter Verwendung von unterschiedlichen Rahmenparametern.

4 Testergebnisse zu Transaktionssicherheit

Die Testergebnisse zur Transaktionssicherheit sind für WebSphere in Abbildung 2 und für JBoss in Abbildung 3 grafisch aufbereitet.

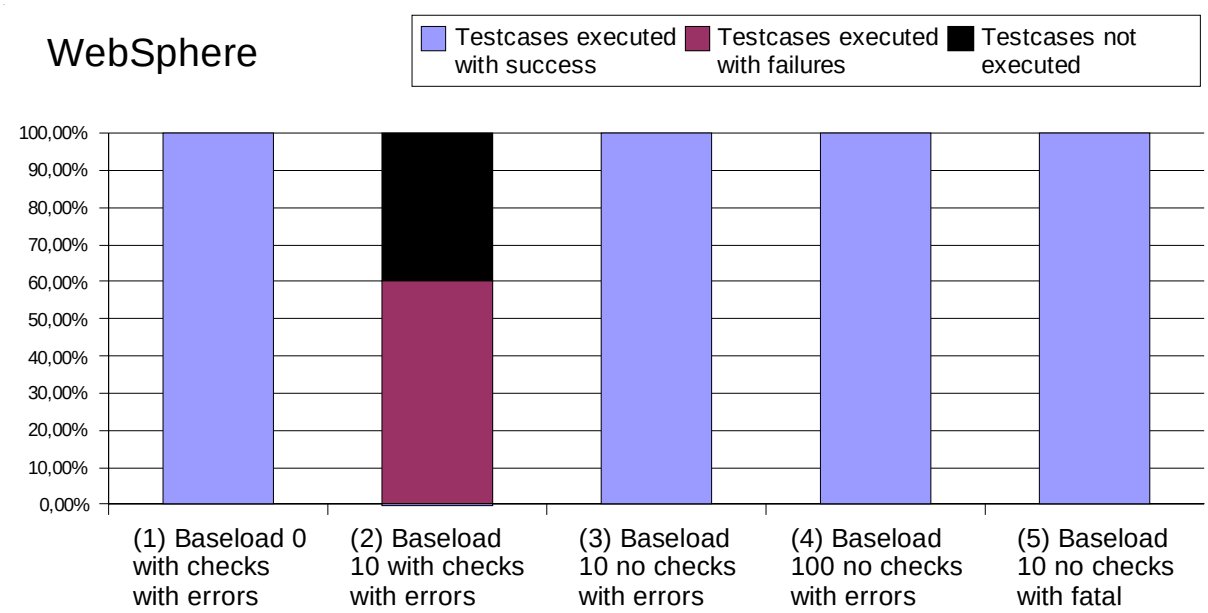

Abbildung 2: Testergebnisse zur Transaktionssicherheit. Dargestellt ist die prozentuale Verteilung der durchgeführten Testfälle (y-Achse) pro Testkategorie (x-Achse).
Die Bestandteile der Testkategorien sind: No Baseload = ohne Basislast, Baseload 10(0) = Basislast von zehn (hundert) parallelen Transaktionen, with checks = Überprüfung aller Transaktionen, without checks = ohne Überprüfung der Basislast-Transaktionen, with errors = Fehler der Kategorie ERROR, with fatal = nur Fehler der Kategorie FATAL.

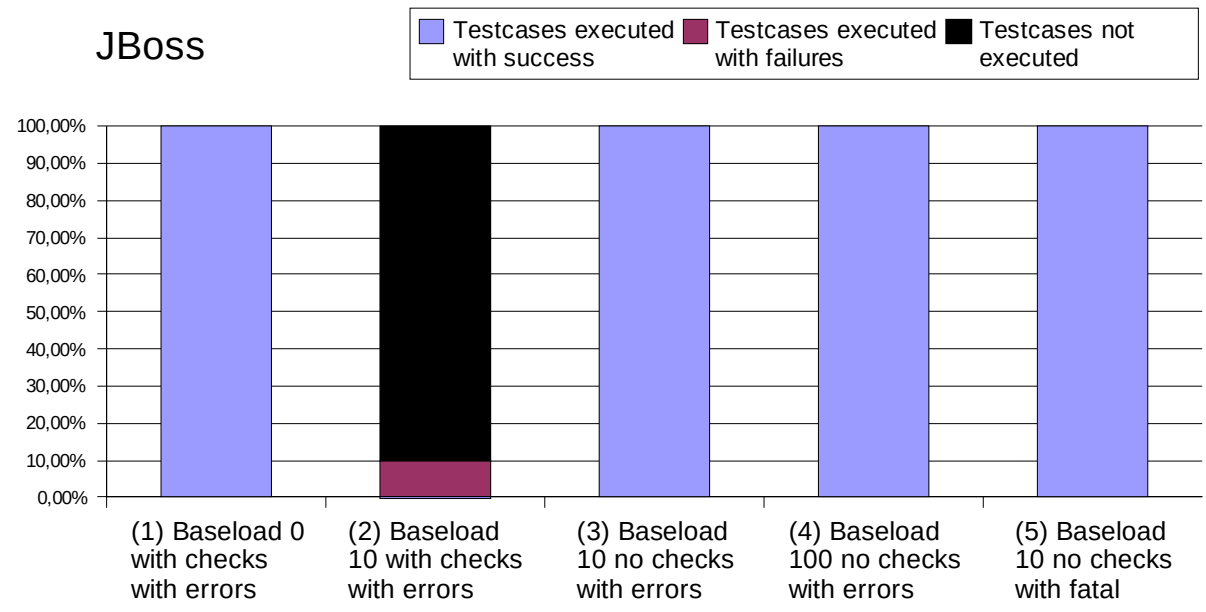

Abbildung 3: Testergebnisse zur Transaktionssicherheit. Dargestellt ist die prozentuale Verteilung der durchgeführten Testfälle (y-Achse) pro Testkategorie (x-Achse). Die Bestandteile der Testkategorien sind identisch mit Abb. 2

JBoss und WebSphere erwiesen sich im Rahmen der durchgeführten Testfälle beide als stabil und für die Transaktionsverarbeitung geeignet. Bei allen Testkategorien, bis auf Kategorie (2), wurden alle Testszenarien mit dem erwarteten Ergebnis durchgeführt sowie mit einem positiven Konsistenztest der Datenbasis abgeschlossen. Die Testkategorie (2) zeigt das Verhalten der Systeme bei zu vielen blockierenden Sperren. Diese wurden durch die Validierung des Testergebnisses und der Durchführung des Konsistenztests sowohl der Transaktionen der Testfälle als auch der Basislast-Transaktionen ausgelöst. Das Endergebnis für diese Testkategorie gegenüber einem Anwender stellt sich in beiden Fällen gleich dar: kein Testfall ist erfolgreich durchgeführt worden. WebSphere (mit DB2) verfügt jedoch über einen besseren und stabileren Locking-Algorithmus als JBoss (mit MaxDB), da WebSphere mehr fehlerhafte Testfälle selbständig abbricht, bevor das System komplett sperrt. Die Testkategorie (3) ist identisch mit (2), jedoch werden nur noch die Transaktionen der Testfälle der Validierungs- und Konsistenzprüfung unterzogen, nicht mehr die Basislast-Transaktionen.

Aus den Problemen bei Testkategorie (2) sowie dem Erfolg von (3) und (4) lässt sich schließen, dass die Validierungs- und Konsistenzprüfungslogik eine enorme Anzahl von Datenbank-Ressourcen durch Sperren blockiert, obwohl nur Daten gelesen wurden. Bei genauerer Betrachtung wird recht schnell klar, dass die Ausführung der Validierungslogik über dieselbe *Datasource*-Definition wie die CMP-Zugriffe diese Probleme verursacht haben. Sowohl der Applicationserver als auch die Datenbank haben die CMP-Transaktionen sowie die *Readonly*-Validierungs-Transaktionen gleich behandelt, als würden Daten verändert werden. Dadurch wurden exklusive Sperren vielfach auf denselben Ressourcen gesetzt, sodass der Durchsatz der Abarbeitung der Transaktionen drastisch einbrach, da die Transaktionen sequentiell ausgeführt werden mussten. Oft stellte sich auch eine *Deadlock*-Situation ein, sodass ganze Testfälle sich gegenseitig blockierten und entweder durch *Deadlock*-Erkennung oder durch die Überschreitung der Transaktionszeit (*Timeout*) rückgängig gemacht wurden.

Als Folgerung aus dieser Problematik lässt sich ziehen, dass Leseoperationen über separate *Datasource*-Definitionen ausgeführt werden sollten, welche als *Readonly*-Verbindungen konfiguriert sind und idealerweise mit einem geringeren Isolationsgrad auskommen (*Read Commited* anstelle von *Repeatable Read*). So kann sowohl der Applicationserver Datenbankzugriffe optimieren (bei *Readonly*-Transaktionen ist beispielsweise kein Aufruf von `ejbStore` notwendig) als auch die Datenbank das Setzen von Sperren effizienter handhaben (weniger Sperren und für die *Readonly* Transaktionen nur *Readonly*-Sperren). Dies führt zu einem erheblichen Anstieg der Parallelität und damit des Gesamtdurchsatzes an Transaktionen, welche das J2EE-System in einem bestimmten Zeitfenster durchführen kann.

Weitere Einzelheiten zu dieser Untersuchung sind unter [Ora04] zu finden.

5 Diskussion der Testergebnisse

WebSphere als marktführende Implementierung der J2EE-Spezifikation sowie DB2 haben die Erwartungen an Stabilität und Transaktionssicherheit auch bei fatalen Fehlern im vollen Umfang erfüllt. Überraschender war die Feststellung, dass die Opensource-Variante aus JBoss und MaxDB in den Tests ebenfalls selbst mit fatalen Fehlern zurecht kam und die Stabilität und Transaktionssicherheit im selben Maße wie die WebSphere-Implementierung

gewährleistete. Auch die Performanz und der Durchsatz waren während der Tests durchaus vergleichbar, wenn auch WebSphere in allen Kategorien bessere Werte gezeigt hat. Vor allem bei steigender Nebenläufigkeit und dem Setzen von Sperren auf denselben Ressourcen skaliert WebSphere besser, was wohl hauptsächlich auf die statische Verarbeitung (CMP-Code wird bei Deployment generiert, nicht zur Laufzeit interpretiert) und der effektiveren Ressourcennutzung von WebSphere liegen dürfte.

Obwohl die Tests damit zu demselben positiven Ergebnis für die Gewährleistung von Transaktionssicherheit auf beiden J2EE-Infrastrukturen gelangen, wird diese bei genauerer Betrachtung auf vollkommen unterschiedliche Weise realisiert. Bei JBoss liegt die Verantwortung bei der Datenbank. Somit sind die guten Testergebnisse der guten Implementierung der MaxDB zu verdanken. Würde ein anderes DBMS eingesetzt werden, welches Probleme bei der Gewährleistung der Transaktionssicherheit hat (beispielsweise beim *Recovery* nach einem Systemfehler), hätten die Testergebnisse wahrscheinlich anders ausgesehen. Die akzeptablen Performanzdaten sind dagegen ein Verdienst von JBoss, welche jedoch durch die leichtgewichtige Realisierung des Transaktionsdienstes begünstigt werden. Bei WebSphere liegt die Verantwortung bei dem eigenen Transaktionsdienst, da WebSphere selbst *Recovery*, erweitertes *Locking* und ausgefeilte Fehlerbehandlungen realisiert und damit dem DBMS viele Aufgaben abnimmt.

6 Schlussfolgerung

In Java können durch die Verwendung von statischen Klassenvariablen und Multi-Threading Isolationsprobleme auftreten. Trotz der derzeitig noch existierenden Isolationsprobleme lassen sich durch nachhaltige Tests und Qualitätssicherungsmaßnahmen stabile und verlässliche Anwendungen erreichen.

Die Qualität von J2EE-Anwendungen lässt sich durch den Einsatz des TTS deutlich steigern, bei gleichzeitiger Senkung der Testkosten durch Benutzung einer vorhandenen Infrastruktur. Die Gesamtkosten für den Lebenszyklus einer J2EE-Software lassen sich somit effektiv reduzieren.

Mit der Verfügbarkeit von J2EE-Produkten auf Basis der Isolate-API ist eine weitere Reduzierung der Isolationsprobleme Beeinflussung und Zuverlässigkeit zu erwarten.

7 Abkürzungsverzeichnis

CMP	Container Managed Persistence
JCA	Java Connector Architecture
JMX	Java Management Extension
JNI	Java Native Interface
LOC	Lines of Code
PRJVM	Persistent Reusable Java Virtual Machine
SUT	System under Test
TCS	Test Controller System
TTS	Transaction Testsystem

8 Literaturverzeichnis

[Bey05] Marc Beyerle, Joachim Franz, Wilhelm G. Spruth: Persistent Reuseable Java Virtual Machine unter z/OS und Linux.. Inform., Forsch. Entwickl. 20(1-2): 102-111 (2005). http://www-ti.informatik.uni-tuebingen.de/~spruth/publish.html

[Borm01] S. Borman, S. Paice, M. Webster, M. Trotter, R. McGuire, A. Stevens,B. Hutchison, R. Berry: A Serially Reusable Java(tm) Virtual Machine Implementation forHigh Volume, Highly Reliable, Transaction Processing. TR 29.3406, IBM, 2001.

[Cza00] G. Czajkowski: Application Isolation in the Java™ Virtual Machine. http://java.sun.com/j2se/jcp/AppIsolationAPI/oopsla00-czajkowski-final.pdf, Sun, 2000.

[IW04] Evers: JBoss Application Server gets J2EE-certified. Info World, July 16, 2004. http://www.infoworld.com/article/04/07/16/HNjbosscert_1.html

[JTA99] S. Cheung, V. Matena: Java Transaction API (JTA), Version 1.0.1. Sun, 1999. http://java.sun.com/products/jta/index.html

[Ora04] Oliver Raible: Transaktionsverarbeitung in J2EE-Systemen, Kapitel 7. ab S. 128.. http://www.byteacademy.com/download/ora04.pdf

[San04] Bo Sandén: Coping with Java Threads. IEEE Computer, Vol. 37, Nr. 4, April 2004, p. 20.